Graph Convolutional Branch and Bound

v1.0.0

Generated by Doxygen 1.9.6

Chapter 1

Graph Convolutional Branch and Bound TSP Solver

This repository contains the implementation of the Graph Convolutional Branch and Bound solver for the Traveling Salesman Problem. It combines a 1Tree branch and bound proposed by `Held and Karp` with the Graph Convolutional Network proposed by `Joshi, Laurent, and Bresson`. In the `src` folder, you can also find a Cplex TSP solver that I developed to verify the correctness of the hybrid one.

1.1 Ideas

The Graph Conv Net is used to preprocess the input Graph to create a distance matrix file. Each entry in this file will be a pair (w_{ij}, p_{ij}), where w_{ij} is the weight of the Edge between nodes i and j, computed as the euclidean distance, and $p_{ij} \in [0,1]$ is the probability, obtained by the neural network, that the corresponding Edge is part of the optimal tour. I will leverage this probabilistic information to expedite the exploration of the branch and bound tree.

1.2 1-Tree Branch and Bound

To improve efficiency, the original 1-Tree Branch and Bound approach proposed by Held and Karp was not implemented. Instead, a modified version, well described in the `Valenzuela and Jones` paper, was used.

1.3 Graph Convolutional Network

I used the pre-trained Graph Conv Nets that Joshi released in the `official repository` of the paper. These networks were trained on one million instances of Euclidean TSP, with cities sampled from the range $[0,1] \times [0,1]$ and sizes of 20, 50, and 100 nodes. The edge embeddings from the last convolutional layer were transformed into a **probabilistic adjacency matrix** using a multi-layer perceptron with softmax. I refer you to this repository to download the trained models and to build the correct Python environment for the forward step.

1.4 Neural Grafting

The hybrid solver obtains the probabilities for each Edge of being in the solution using a Graph Conv Net, it then assigns to a 1-Tree the probability of being the optimal tour by averaging the probabilities of its edges. It then uses these values as follows:

1. **Starting vertex**: to construct a 1Tree, and so solving the relaxed version of the TSP a **starting Vertex** must be chosen. The algorithm tries all verticies and then select the one that yields the best lower bound. If multiple verticies produce the same lower bound, the one with the highest probability is chosen;
2. **Probabilistic nearest neighbor**: the algorithms needs an initial feasible solution. In the classical solver, this is accomplished by executing the nearest neighbor algorithm with each vertex as the starting city and then selecting the lowest tour found as the initial tour. The hybrid solver also uses a prob-nearest-neighbor algorithm. Starting from each city, it selects at every step the unvisited city that is linked to the current one by the edge with the highest probability. The tour found with this algorithm is then compared with the one returned by the nearest neighbor, and the best one is used as the initial feasible solution;
3. **Node Selection**: all subproblems generated by the branching steps are stored and sorted from lowest to highest. In the Hybrid Solver when two subproblems have the same value, the one with the highest probability is selected first in a **Best-First-Prob** manner. This procedure is extremely flexible, as it provides meta-parameters that allow for the modification of the subproblems sorting criterion, enabling any desired trade-off between the probability and the value of 1Trees.
4. **Variable Selection**: hen a 1Tree is not a correct tour but provides a lower bound on the current best solution found, a branching step must be taken. The selection of the edge to be fixed as mandatory or forbidden in the new branch and bound nodes is accomplished by integrating the `Shutler's method` with the edge probabilities.

1.5 Code Documentation

All code documentation was completed using `Doxygen`, and is accessible in both `online` and `PDF` formats.

1.6 Results

Below are some of the results obtained:

Chapter 2

An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem

>**:rocket: Update:** If you are interested in this work, you may be interested in **`our latest paper`** and **`up-to-date codebase`** bringing together several architectures and learning paradigms for learning-driven TSP solvers under one pipeline.

This repository contains code for the paper `**"An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem"**` by Chaitanya K. Joshi, Thomas Laurent and Xavier Bresson.

We introduce a new learning-based approach for approximately solving the Travelling Salesman Problem on 2D Euclidean graphs. We use deep Graph Convolutional Networks to build efficient TSP graph representations and output tours in a non-autoregressive manner via highly parallelized beam search. Our approach outperforms all recently proposed autoregressive deep learning techniques in terms of solution quality, inference speed and sample efficiency for problem instances of fixed graph sizes.

2.1 Overview

The notebook `main.ipynb` contains top-level methods to reproduce our experiments or train models for TSP from scratch. Several modes are provided:

- **Notebook Mode**: For debugging as a Jupyter Notebook
- **Visualization Mode**: For visualization and evaluation of saved model checkpoints (in a Jupyter Notebook)
- **Script Mode**: For running full experiments as a python script

Configuration parameters for notebooks and scripts are passed as `.json` files and are documented in `config.py`.

2.2 Pre-requisite Downloads

2.2.0.1 TSP Datasets

Download TSP datasets from `this link`: Extract the `.tar.gz` file and place each `.txt` file in the `/data` directory. (We provide TSP10, TSP20, TSP30, TSP50 and TSP100.)

2.2.0.2 Pre-trained Models

Download pre-trained model checkpoints from this link: Extract the `.tar.gz` file and place each directory in the `/logs` directory. (We provide TSP20, TSP50 and TSP100 models.)

2.3 Usage

2.3.0.1 Installation

We ran our code on Ubuntu 16.04, using Python 3.6.7, PyTorch 0.4.1 and CUDA 9.0.

> **Note:** This codebase was developed for a rather outdated version of PyTorch. Attempting to run the code with PyTorch 1.x may need further modifications, e.g. see this issue.

Step-by-step guide for local installation using a Terminal (Mac/Linux) or Git Bash (Windows) via Anaconda:

```
# Install [Anaconda 3](https://www.anaconda.com/) for managing Python packages and environments.
curl -o ~/miniconda.sh -O https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
chmod +x ~/miniconda.sh
./miniconda.sh
source ~/.bashrc

# Clone the repository.
git clone https://github.com/chaitjo/graph-convnet-tsp.git
cd graph-convnet-tsp

# Set up a new conda environment and activate it.
conda create -n gcn-tsp-env python=3.6.7
source activate gcn-tsp-env

# Install all dependencies and Jupyter Lab (for using notebooks).
conda install pytorch=0.4.1 cuda90 -c pytorch
conda install numpy==1.15.4 scipy==1.1.0 matplotlib==3.0.2 seaborn==0.9.0 pandas==0.24.2 networkx==2.2
      scikit-learn==0.20.2 tensorflow-gpu==1.12.0 tensorboard==1.12.0 Cython
pip3 install tensorboardx==1.5 fastprogress==0.1.18
conda install -c conda-forge jupyterlab
```

2.3.0.2 Running in Notebook/Visualization Mode

Launch Jupyter Lab and execute/modify `main.ipynb` cell-by-cell in Notebook Mode.

```
jupyter lab
```

Set `viz_mode = True` in the first cell of `main.ipynb` to toggle Visualization Mode.

2.3.0.3 Running in Script Mode

Set `notebook_mode = False` and `viz_mode = False` in the first cell of `main.ipynb`. Then convert the notebook from `.ipynb` to `.py` and run the script (pass path of config file as argument):

```
jupyter nbconvert --to python main.ipynb
python main.py --config <path-to-config.json>
```

2.3.0.4 Splitting datasets into Training and Validation sets

For TSP10, TSP20 and TSP30 datasets, everything is good to go once you download and extract the files. For TSP50 and TSP100, the 1M training set needs to be split into 10K validation samples and 999K training samples. Use the `split_train_val.py` script to do so. For consistency, the script uses the first 10K samples in the 1M file as the validation set and the remaining 999K as the training set.

```
cd data
python split_train_val.py --num_nodes <num-nodes>
```

2.3.0.5 Generating new data

New TSP data can be generated using the Concorde solver.

```
# Install the pyConcorde library in the /data directory
cd data
git clone https://github.com/jvkersch/pyconcorde
cd pyconcorde
pip install -e .
cd ..

# Run the data generation script
python generate_tsp_concorde.py --num_samples <num-sample> --num_nodes <num-nodes>
```

2.4 Resources

- Optimal TSP Datasets generated with Concorde
- Paper on arXiv
- Follow-up workshop paper

Chapter 3

Main

This is the heart of the Graph Convolutional Branch and Bound Solver, with the main file being `HybridSolver.py` written in Python. The script first employs the `Convolutional Graph Network` to calculate the probability of each edge being included in the optimal tour, which is then saved in a `.csv` adjacency matrix file along with weights. Next, the script runs the 1-Tree Branch-and-Bound algorithm on the instance using the `main.c` script. The Branch-and-Bound code is divided into two primary subfolders: `algorithms` and `data_structure`, while the Graph Conv Net is located in the `graph-convnet-tsp` subfolder. Within the latter folder, a `main.py` file was created by combining the code from the original repository's Python notebook and adding some functions specific to the Hybrid Solver. Credit for the neural network code goes to the authors of the Graph Convolutional Network repository, and interested readers are referred to that repository for a more thorough explanation of the code.

Chapter 4

Namespace Index

4.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 5

Class Index

5.1 Class List

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

Chapter 6

File Index

6.1 File List

Here is a list of all files with brief descriptions:

Chapter 7

Namespace Documentation

7.1 HybridSolver Namespace Reference

Functions

- def adjacency_matrix (orig_graph)
- def create_temp_file (num_nodes, str_grap)
- def get_nodes (graph)
- def get_instance (instance, num_nodes)
- def build_c_program (build_directory, num_nodes, hyb_mode)
- def hybrid_solver (num_instances, num_nodes, hyb_mode, gen_matrix)

Variables

- argparse parser = argparse.ArgumentParser()
- type
- str
- default
- int
- action
- argparse opts = parser.parse_args()
- argparse gen_matrix = False

7.1.1 Detailed Description

```
@file: HybridSolver.py
@author Lorenzo Sciandra
@brief First it builds the program in C, specifying the number of nodes to use and whether it is in hybrid
Then it runs the graph conv net on the instance, and finally it runs the Branch and Bound.
It can be run on a single instance or a range of instances.
The input matrix is generated by the neural network and stored in the data folder. The output is stored in
@version 1.0.0
@data 2024-05-1
@copyright Copyright (c) 2024, license MIT

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound
```

7.1.2 Function Documentation

7.1.2.1 adjacency_matrix()

```
def HybridSolver.adjacency_matrix (
              orig_graph )
```

```
Calculates the adjacency matrix of the graph.
Args:
    orig_graph: The original graph.

Returns:
    The adjacency matrix of the graph.
```

Definition at line 22 of file HybridSolver.py.

7.1.2.2 build_c_program()

```
def HybridSolver.build_c_program (
              build_directory,
              num_nodes,
              hyb_mode )
```

```
Builds the C program with the specified number of nodes and whether it is in hybrid mode or not.
Args:
    build_directory: The directory where the CMakeLists.txt file is located and where the executable will be b
    num_nodes: The number of nodes to use in the C program.
    hyb_mode: 1 if the program is in hybrid mode, 0 otherwise.
```

Definition at line 115 of file HybridSolver.py.

7.1.2.3 create_temp_file()

```
def HybridSolver.create_temp_file (
              num_nodes,
              str_grap )
```

```
Creates a temporary file to store the current instance of the TSP for the neural network.
Args:
    num_nodes: The number of nodes in the TSP instance.
    str_grap: The string representation of the graph.
```

Definition at line 41 of file HybridSolver.py.

7.1.2.4 get_instance()

```
def HybridSolver.get_instance (
              instance,
              num_nodes )
```

```
Gets the instance of the TSP from the file.
Args:
    instance: The instance to get.
    num_nodes: The number of nodes in the TSP instance.

Returns:
    The graph in a list and string format.
```

Definition at line 81 of file HybridSolver.py.

7.1.2.5 get_nodes()

```
def HybridSolver.get_nodes (
              graph )
```

```
From a graph, it returns the nodes in a string format.
Args:
    graph: The graph to get the nodes from.

Returns:
    The nodes in a string format.
```

Definition at line 63 of file HybridSolver.py.

7.1.2.6 hybrid_solver()

```
def HybridSolver.hybrid_solver (
              num_instances,
              num_nodes,
              hyb_mode,
              gen_matrix )
```

```
The Graph Convolutional Branch-and-Bound Solver.
Args:
    num_instances: The range of instances to run on the Solver.
    num_nodes: The number of nodes in each TSP instance.
    hyb_mode: True if the program is in hybrid mode, False otherwise.
    gen_matrix: True if the adjacency matrix is already generated, False otherwise.
```

Definition at line 147 of file HybridSolver.py.

7.1.3 Variable Documentation

7.1.3.1 action

```
HybridSolver.action
```

Definition at line 263 of file HybridSolver.py.

7.1.3.2 default

```
HybridSolver.default
```

Definition at line 261 of file HybridSolver.py.

7.1.3.3 gen_matrix

```
argparse HybridSolver.gen_matrix = False
```

Definition at line 268 of file HybridSolver.py.

7.1.3.4 int

```
HybridSolver.int
```

Definition at line 262 of file HybridSolver.py.

7.1.3.5 opts

```
argparse HybridSolver.opts = parser.parse_args()
```

Definition at line 264 of file HybridSolver.py.

7.1.3.6 parser

```
argparse HybridSolver.parser = argparse.ArgumentParser()
```

Definition at line 260 of file HybridSolver.py.

7.1.3.7 str

```
HybridSolver.str
```

Definition at line 261 of file HybridSolver.py.

7.1.3.8 type

```
HybridSolver.type
```

Definition at line 261 of file HybridSolver.py.

7.2 main Namespace Reference

Functions

- def compute_prob (net, config, dtypeLong, dtypeFloat)
- def write_adjacency_matrix (graph, y_probs, x_edges_values, nodes_coord, filepath, num_nodes, kmedoids_labels=None)
- def add_dummy_cities (num_nodes, model_size)
- def create_temp_file (num_nodes, str_grap)
- def cluster_nodes (graph, k)
- def fix_instance_size (graph, num_nodes, model_size=100)
- def get_instance (num_nodes)
- def main (filepath, num_nodes, model_size)

Variables

- category
- sys filepath = sys.argv[1]
- int num_nodes = int(sys.argv[2])
- int model_size = int(sys.argv[3])

7.2.1 Detailed Description

```
@file main.py
@author Lorenzo Sciandra
@brief A recombination of code take from: https://github.com/chaitjo/graph-convnet-tsp.
Some functions were created for the purpose of the paper.
@version 1.0.0
@data 2024-05-1
@copyright Copyright (c) 2024, license MIT
Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound
```

7.2.2 Function Documentation

7.2.2.1 add_dummy_cities()

```
def main.add_dummy_cities (
              num_nodes,
              model_size )
```

```
This function adds dummy cities to the graph instance. The dummy cities are randomly generated are
added to the graph instance and the new instance is saved in a temporary file.
Args:
    num_nodes: The number of nodes of the graph instance.
    model_size: The size of the Graph Convolutional Network to use.
```

Definition at line 199 of file main.py.

7.2.2.2 cluster_nodes()

```
def main.cluster_nodes (
              graph,
              k )
```

```
Applies the k-medoids clustering to the graph.
Args:
    graph: The graph to cluster.
    k: The number of clusters to create.
Returns:
    medoids_str: The medoids of the clusters.
    kmedoids.labels_: The labels of the clusters.
```

Definition at line 269 of file main.py.

7.2.2.3 compute_prob()

```
def main.compute_prob (
              net,
              config,
              dtypeLong,
              dtypeFloat )
```

```
This function computes the probability of the edges being in the optimal tour, by running the GCN.
Args:
    net: The Graph Convolutional Network.
    config: The configuration file, from which the parameters are taken.
    dtypeLong: The data type for the long tensors.
    dtypeFloat: The data type for the float tensors.
Returns:
    y_probs: The probability of the edges being in the optimal tour.
    x_edges_values: The distance between the nodes.
```

Definition at line 37 of file main.py.

7.2.2.4 create_temp_file()

```
def main.create_temp_file (
              num_nodes,
              str_grap )
```

```
Creates a temporary file with the graph instance.
Args:
    num_nodes: The number of nodes of the graph instance.
    str_grap: The graph instance.
```

Definition at line 253 of file main.py.

7.2.2.5 fix_instance_size()

```
def main.fix_instance_size (
              graph,
              num_nodes,
              model_size = 100 )
```

```
The function that fixes the instance size with clustering.
It applies the k-medoids clustering to the graph and creates a new instance with the medoids as the new nodes.
Args:
    graph: The graph to fix.
    num_nodes: The number of nodes of the graph instance.
    model_size: The size of the Graph Convolutional Network to use.
```

Definition at line 287 of file main.py.

7.2.2.6 get_instance()

```
def main.get_instance (
              num_nodes )
```

```
The function that reads the current instance from the file.
Args:
    num_nodes: The number of nodes of the graph instance.
Returns:
    graph: The graph instance.
```

Definition at line 312 of file main.py.

7.2.2.7 main()

```
def main.main (
              filepath,
              num_nodes,
              model_size )
```

```
The function that runs the Graph Convolutional Network and writes the adjacency matrix to a file
for the given input instance.
Args:
    filepath: The path to the file where the adjacency matrix will be written.
    num_nodes: The number of nodes in the TSP instance.
    model_size: The size of the Graph Convolutional Network to use.
```

Definition at line 345 of file main.py.

7.2.2.8 write_adjacency_matrix()

```
def main.write_adjacency_matrix (
              graph,
              y_probs,
              x_edges_values,
              nodes_coord,
              filepath,
              num_nodes,
              kmedoids_labels = None )
```

```
This function writes the adjacency matrix to a file.
The file is in the format:
        cities: (x1, y1);(x2, y2);...;(xn, yn)
        adjacency matrix:
        (0.0, 0.0);(0.23, 0.9);...;(0.15, 0.56)
        ...
        (0.23, 0.9);(0.3, 0.59);...;(0.0, 0.0)
        where each entry is (distance, probability)
If needed adjusts the size of the graph when the model size is different from the number of nodes in the insta
Args:
    graph: The set of nodes in the graph.
    y_probs: The probability of the edges being in the optimal tour.
    x_edges_values: The weight of the edges.
    nodes_coord: The nodes coordinates used in the GCN.
    filepath: The path to the file where the adjacency matrix will be written.
    num_nodes: The number of nodes in the TSP instance.
    kmedoids_labels: The labels of the k-medoids clustering.
```

Definition at line 115 of file main.py.

7.2.3 Variable Documentation

7.2.3.1 category

```
main.category
```

Definition at line 24 of file main.py.

7.2.3.2 filepath

```
sys main.filepath = sys.argv[1]
```

Definition at line 421 of file main.py.

7.2.3.3 model_size

```
int main.model_size = int(sys.argv[3])
```

Definition at line 423 of file main.py.

7.2.3.4 num_nodes

```
int main.num_nodes = int(sys.argv[2])
```

Definition at line 422 of file main.py.

Chapter 8

Class Documentation

8.1 ConstrainedEdge Struct Reference

A reduced form of an Edge in the Graph, with only the source and destination Nodes.

```
#include <mst.h>
```

Public Attributes

- unsigned short src

  The source Node of the Edge.
- unsigned short dest

  The destination Node of the Edge.

8.1.1 Detailed Description

A reduced form of an Edge in the Graph, with only the source and destination Nodes.

Definition at line 21 of file mst.h.

8.1.2 Member Data Documentation

8.1.2.1 dest

```
unsigned short ConstrainedEdge::dest
```

The destination Node of the Edge.

Definition at line 23 of file mst.h.

8.1.2.2 src

```
unsigned short ConstrainedEdge::src
```

The source Node of the Edge.

Definition at line 22 of file mst.h.

8.2 DllElem Struct Reference

The double linked List element.

```
#include <linked_list.h>
```

Public Attributes

- void * value
  The value of the element, void pointer to be able to store any type of data.
- struct DllElem * next
  The next element in the List.
- struct DllElem * prev
  The previous element in the List.

8.2.1 Detailed Description

The double linked List element.

Definition at line 27 of file linked_list.h.

8.2.2 Member Data Documentation

8.2.2.1 next

```
struct DllElem* DllElem::next
```

The next element in the List.

Definition at line 29 of file linked_list.h.

8.2.2.2 prev

```
struct DllElem* DllElem::prev
```

The previous element in the List.

Definition at line 30 of file linked_list.h.

8.2.2.3 value

```
void* DllElem::value
```

The value of the element, void pointer to be able to store any type of data.

Definition at line 28 of file linked_list.h.

8.3 Edge Struct Reference

Structure of an Edge.

```
#include <graph.h>
```

Public Attributes

- unsigned short src

  ID of the source vertex.
- unsigned short dest

  ID of the destination vertex.
- unsigned short symbol

  Symbol of the Edge, i.e. its unique ID.
- double weight

  Weight of the Edge, 1 if the data_structures is not weighted.
- double prob

  Probability of the Edge to be in an optimal tour.
- unsigned short positionInGraph

  Position of the Edge in the list of Edges of the Graph.

8.3.1 Detailed Description

Structure of an Edge.

Definition at line 40 of file graph.h.

8.3.2 Member Data Documentation

8.3.2.1 dest

```
unsigned short Edge::dest
```

ID of the destination vertex.

Definition at line 42 of file graph.h.

8.3.2.2 positionInGraph

```
unsigned short Edge::positionInGraph
```

Position of the Edge in the list of Edges of the Graph.

Definition at line 46 of file graph.h.

8.3.2.3 prob

```
double Edge::prob
```

Probability of the Edge to be in an optimal tour.

Definition at line 45 of file graph.h.

8.3.2.4 src

```
unsigned short Edge::src
```

ID of the source vertex.

Definition at line 41 of file graph.h.

8.3.2.5 symbol

```
unsigned short Edge::symbol
```

Symbol of the Edge, i.e. its unique ID.

Definition at line 43 of file graph.h.

8.3.2.6 weight

```
double Edge::weight
```

Weight of the Edge, 1 if the data_structures is not weighted.

Definition at line 44 of file graph.h.

8.4 FibonacciHeap Struct Reference

The Fibonacci Heap datastructure as collection of Heap-ordered Trees.

```
#include <fibonacci_heap.h>
```

Public Attributes

- OrdTreeNode ∗ min_root

  The root of the Heap-ordered Tree with the minimum value.
- OrdTreeNode ∗ head_tree_list

  The root of the head Tree in the Fibonacci Heap.
- OrdTreeNode ∗ tail_tree_list

  The root of the tail Tree in the Fibonacci Heap.
- unsigned short num_nodes

  The number of Nodes in the Heap.
- unsigned short num_trees

  The number of Trees in the Heap.

8.4.1 Detailed Description

The Fibonacci Heap datastructure as collection of Heap-ordered Trees.

Definition at line 43 of file fibonacci_heap.h.

8.4.2 Member Data Documentation

8.4.2.1 head_tree_list

```
OrdTreeNode* FibonacciHeap::head_tree_list
```

The root of the head Tree in the Fibonacci Heap.

Definition at line 45 of file fibonacci_heap.h.

8.4.2.2 min_root

`OrdTreeNode* FibonacciHeap::min_root`

The root of the Heap-ordered Tree with the minimum value.

Definition at line 44 of file fibonacci_heap.h.

8.4.2.3 num_nodes

`unsigned short FibonacciHeap::num_nodes`

The number of Nodes in the Heap.

Definition at line 47 of file fibonacci_heap.h.

8.4.2.4 num_trees

`unsigned short FibonacciHeap::num_trees`

The number of Trees in the Heap.

Definition at line 48 of file fibonacci_heap.h.

8.4.2.5 tail_tree_list

`OrdTreeNode* FibonacciHeap::tail_tree_list`

The root of the tail Tree in the Fibonacci Heap.

Definition at line 46 of file fibonacci_heap.h.

8.5 Forest Struct Reference

A Forest is a list of Sets.

```
#include <mfset.h>
```

Public Attributes

- unsigned short num_sets

  Number of Sets in the Forest.
- Set sets [MAX_VERTEX_NUM]

  Array of Sets.

8.5.1 Detailed Description

A Forest is a list of Sets.

Definition at line 28 of file mfset.h.

8.5.2 Member Data Documentation

8.5.2.1 num_sets

```
unsigned short Forest::num_sets
```

Number of Sets in the Forest.

Definition at line 29 of file mfset.h.

8.5.2.2 sets

```
Set Forest::sets[MAX_VERTEX_NUM]
```

Array of Sets.

Definition at line 30 of file mfset.h.

8.6 Graph Struct Reference

Structure of a Graph.

```
#include <graph.h>
```

Public Attributes

- GraphKind kind

  Type of the Graph.
- double cost

  Sum of the weights of the Edges in the Graph.
- unsigned short num_nodes

  Number of Nodes in the Graph.
- unsigned short num_edges

  Number of Edges in the Graph.
- bool orderedEdges

  True if the Edges are ordered by weight, false otherwise.
- Node nodes [MAX_VERTEX_NUM]

  Array of Nodes.
- Edge edges [MAX_EDGES_NUM]

  Array of Edges.
- Edge edges_matrix [MAX_VERTEX_NUM][MAX_VERTEX_NUM]

  Adjacency matrix of the Graph.

8.6.1 Detailed Description

Structure of a Graph.

Definition at line 51 of file graph.h.

8.6.2 Member Data Documentation

8.6.2.1 cost

`double Graph::cost`

Sum of the weights of the Edges in the Graph.

Definition at line 53 of file graph.h.

8.6.2.2 edges

`Edge Graph::edges[MAX_EDGES_NUM]`

Array of Edges.

Definition at line 58 of file graph.h.

8.6.2.3 edges_matrix

`Edge Graph::edges_matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]`

Adjacency matrix of the Graph.

Definition at line 59 of file graph.h.

8.6.2.4 kind

`GraphKind Graph::kind`

Type of the Graph.

Definition at line 52 of file graph.h.

8.6.2.5 nodes

```
Node Graph::nodes[MAX_VERTEX_NUM]
```

Array of Nodes.

Definition at line 57 of file graph.h.

8.6.2.6 num_edges

```
unsigned short Graph::num_edges
```

Number of Edges in the Graph.

Definition at line 55 of file graph.h.

8.6.2.7 num_nodes

```
unsigned short Graph::num_nodes
```

Number of Nodes in the Graph.

Definition at line 54 of file graph.h.

8.6.2.8 orderedEdges

```
bool Graph::orderedEdges
```

True if the Edges are ordered by weight, false otherwise.

Definition at line 56 of file graph.h.

8.7 List Struct Reference

The double linked list.

```
#include <linked_list.h>
```

Public Attributes

- DllElem * head

  The head of the list as a DllElem.
- DllElem * tail

  The tail of the list as a DllElem.
- size_t size

  The current size of the List.

8.7.1 Detailed Description

The double linked list.

Definition at line 35 of file linked_list.h.

8.7.2 Member Data Documentation

8.7.2.1 head

`DllElem* List::head`

The head of the list as a DllElem.

Definition at line 36 of file linked_list.h.

8.7.2.2 size

`size_t List::size`

The current size of the List.

Definition at line 38 of file linked_list.h.

8.7.2.3 tail

`DllElem* List::tail`

The tail of the list as a DllElem.

Definition at line 37 of file linked_list.h.

8.8 ListIterator Struct Reference

The iterator for the List.

```
#include <linked_list.h>
```

Public Attributes

- List * list

  The List to iterate.
- DllElem * curr

  The current DllElem (element) of the List.
- size_t index

  The current index of the element in the List.

8.8.1 Detailed Description

The iterator for the List.

Definition at line 43 of file linked_list.h.

8.8.2 Member Data Documentation

8.8.2.1 curr

```
DllElem* ListIterator::curr
```

The current DllElem (element) of the List.

Definition at line 45 of file linked_list.h.

8.8.2.2 index

```
size_t ListIterator::index
```

The current index of the element in the List.

Definition at line 46 of file linked_list.h.

8.8.2.3 list

```
List* ListIterator::list
```

The List to iterate.

Definition at line 44 of file linked_list.h.

8.9 MST Struct Reference

Minimum Spanning Tree, or MST, and also a 1-Tree.

```
#include <mst.h>
```

Public Attributes

- bool isValid
  True if the MST has the correct number of Edges, false otherwise.
- double cost
  The total cost of the MST, i.e. the sum of the weights of the Edges.
- double prob
  The probability of the MST, i.e. the average of the probabilities of its Edges.
- unsigned short num_nodes
  The number of Nodes in the MST.
- unsigned short num_edges
  The number of Edges in the MST.
- Node nodes [MAX_VERTEX_NUM]
  The set of Nodes in the MST.
- Edge edges [MAX_VERTEX_NUM]
  The set of Edges in the MST, these are $|V|$ because the MST can be a 1-Tree.
- short edges_matrix [MAX_VERTEX_NUM][MAX_VERTEX_NUM]
  -1 if there is no Edge between the two Nodes, otherwise the index of the Edge in the MST.

8.9.1 Detailed Description

Minimum Spanning Tree, or MST, and also a 1-Tree.

Definition at line 28 of file mst.h.

8.9.2 Member Data Documentation

8.9.2.1 cost

```
double MST::cost
```

The total cost of the MST, i.e. the sum of the weights of the Edges.

Definition at line 30 of file mst.h.

8.9.2.2 edges

```
Edge MST::edges[MAX_VERTEX_NUM]
```

The set of Edges in the MST, these are $|V|$ because the MST can be a 1-Tree.

Definition at line 35 of file mst.h.

8.9.2.3 edges_matrix

```
short MST::edges_matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]
```

-1 if there is no Edge between the two Nodes, otherwise the index of the Edge in the MST.

Definition at line 36 of file mst.h.

8.9.2.4 isValid

```
bool MST::isValid
```

True if the MST has the correct number of Edges, false otherwise.

Definition at line 29 of file mst.h.

8.9.2.5 nodes

```
Node MST::nodes[MAX_VERTEX_NUM]
```

The set of Nodes in the MST.

Definition at line 34 of file mst.h.

8.9.2.6 num_edges

```
unsigned short MST::num_edges
```

The number of Edges in the MST.

Definition at line 33 of file mst.h.

8.9.2.7 num_nodes

```
unsigned short MST::num_nodes
```

The number of Nodes in the MST.

Definition at line 32 of file mst.h.

8.9.2.8 prob

```
double MST::prob
```

The probability of the MST, i.e. the average of the probabilities of its Edges.

Definition at line 31 of file mst.h.

8.10 Node Struct Reference

Structure of a Node.

```
#include <graph.h>
```

Public Attributes

- double x

  x coordinate of the Node.
- double y

  y coordinate of the Node.
- unsigned short positionInGraph

  Position of the Node in the list of Nodes of the Graph, i.e. its unique ID.
- unsigned short num_neighbours

  Number of neighbours of the Node.
- unsigned short neighbours [MAX_VERTEX_NUM - 1]

  Array of IDs of the Node's neighbors.

8.10.1 Detailed Description

Structure of a Node.

Definition at line 30 of file graph.h.

8.10.2 Member Data Documentation

8.10.2.1 neighbours

```
unsigned short Node::neighbours[MAX_VERTEX_NUM - 1]
```

Array of IDs of the Node's neighbors.

Definition at line 35 of file graph.h.

8.10.2.2 num_neighbours

```
unsigned short Node::num_neighbours
```

Number of neighbours of the Node.

Definition at line 34 of file graph.h.

8.10.2.3 positionInGraph

```
unsigned short Node::positionInGraph
```

Position of the Node in the list of Nodes of the Graph, i.e. its unique ID.

Definition at line 33 of file graph.h.

8.10.2.4 x

```
double Node::x
```

x coordinate of the Node.

Definition at line 31 of file graph.h.

8.10.2.5 y

```
double Node::y
```

y coordinate of the Node.

Definition at line 32 of file graph.h.

8.11 OrdTreeNode Struct Reference

A Heap-ordered Tree Node where the key of the parent is <= the key of its children.

```
#include <fibonacci_heap.h>
```

Public Attributes

- unsigned short key

  The key of the Node.
- double value

  The value of the Node.
- struct OrdTreeNode * parent

  The parent of the Node.
- struct OrdTreeNode * left_sibling

  The left sibling of the Node.
- struct OrdTreeNode * right_sibling

  The right sibling of the Node.
- struct OrdTreeNode * head_child_list

  The head of the list of children of the Node.
- struct OrdTreeNode * tail_child_list

  The tail of the list of children of the Node.
- unsigned short num_children

  The number of children of the Node.
- bool marked

  True if the Node has lost a child, false otherwise.
- bool is_root

  True if the Node is a root, false otherwise.

8.11.1 Detailed Description

A Heap-ordered Tree Node where the key of the parent is <= the key of its children.

Definition at line 26 of file fibonacci_heap.h.

8.11.2 Member Data Documentation

8.11.2.1 head_child_list

```
struct OrdTreeNode* OrdTreeNode::head_child_list
```

The head of the list of children of the Node.

Definition at line 34 of file fibonacci_heap.h.

8.11.2.2 is_root

```
bool OrdTreeNode::is_root
```

True if the Node is a root, false otherwise.

Definition at line 38 of file fibonacci_heap.h.

8.11.2.3 key

```
unsigned short OrdTreeNode::key
```

The key of the Node.

Definition at line 27 of file fibonacci_heap.h.

8.11.2.4 left_sibling

```
struct OrdTreeNode* OrdTreeNode::left_sibling
```

The left sibling of the Node.

Definition at line 31 of file fibonacci_heap.h.

8.11.2.5 marked

```
bool OrdTreeNode::marked
```

True if the Node has lost a child, false otherwise.

Definition at line 37 of file fibonacci_heap.h.

8.11.2.6 num_children

```
unsigned short OrdTreeNode::num_children
```

The number of children of the Node.

Definition at line 36 of file fibonacci_heap.h.

8.11.2.7 parent

```
struct OrdTreeNode* OrdTreeNode::parent
```

The parent of the Node.

Definition at line 29 of file fibonacci_heap.h.

8.11.2.8 right_sibling

```
struct OrdTreeNode* OrdTreeNode::right_sibling
```

The right sibling of the Node.

Definition at line 32 of file fibonacci_heap.h.

8.11.2.9 tail_child_list

```
struct OrdTreeNode* OrdTreeNode::tail_child_list
```

The tail of the list of children of the Node.

Definition at line 35 of file fibonacci_heap.h.

8.11.2.10 value

```
double OrdTreeNode::value
```

The value of the Node.

Definition at line 28 of file fibonacci_heap.h.

8.12 Problem Struct Reference

The struct used to represent the overall problem.

```
#include <b_and_b_data.h>
```

Public Attributes

- Graph graph

  The Graph of the problem.
- Graph reformulationGraph

  The Graph used to perform the dual reformulation of Edge weights.
- unsigned short candidateNodeId

  The id of the candidate node.
- unsigned short totTreeLevels

  The total number of levels in the Branch and Bound tree.
- SubProblem bestSolution

  The best solution found so far.
- double bestValue

  The cost of the best solution found so far.
- unsigned int generatedBBNodes

  The number of nodes generated in the Branch and Bound tree.
- unsigned int exploredBBNodes

  The number of nodes explored in the Branch and Bound tree.
- unsigned int num_fixed_edges

  The number of fixed edges in the Branch and Bound tree.
- bool interrupted

  True if the algorithm has been interrupted by timeout.
- clock_t start

  The time when the algorithm started.
- clock_t end

  The time when the algorithm ended.

8.12.1 Detailed Description

The struct used to represent the overall problem.

Definition at line 62 of file b_and_b_data.h.

8.12.2 Member Data Documentation

8.12.2.1 bestSolution

`SubProblem Problem::bestSolution`

The best solution found so far.

Definition at line 67 of file b_and_b_data.h.

8.12.2.2 bestValue

`double Problem::bestValue`

The cost of the best solution found so far.

Definition at line 68 of file b_and_b_data.h.

8.12.2.3 candidateNodeId

`unsigned short Problem::candidateNodeId`

The id of the candidate node.

Definition at line 65 of file b_and_b_data.h.

8.12.2.4 end

`clock_t Problem::end`

The time when the algorithm ended.

Definition at line 74 of file b_and_b_data.h.

8.12.2.5 exploredBBNodes

`unsigned int Problem::exploredBBNodes`

The number of nodes explored in the Branch and Bound tree.

Definition at line 70 of file b_and_b_data.h.

8.12.2.6 generatedBBNodes

`unsigned int Problem::generatedBBNodes`

The number of nodes generated in the Branch and Bound tree.

Definition at line 69 of file b_and_b_data.h.

8.12.2.7 graph

`Graph Problem::graph`

The Graph of the problem.

Definition at line 63 of file b_and_b_data.h.

8.12.2.8 interrupted

`bool Problem::interrupted`

True if the algorithm has been interrupted by timeout.

Definition at line 72 of file b_and_b_data.h.

8.12.2.9 num_fixed_edges

`unsigned int Problem::num_fixed_edges`

The number of fixed edges in the Branch and Bound tree.

Definition at line 71 of file b_and_b_data.h.

8.12.2.10 reformulationGraph

`Graph Problem::reformulationGraph`

The Graph used to perform the dual reformulation of Edge weights.

Definition at line 64 of file b_and_b_data.h.

8.12.2.11 start

```
clock_t Problem::start
```

The time when the algorithm started.

Definition at line 73 of file b_and_b_data.h.

8.12.2.12 totTreeLevels

```
unsigned short Problem::totTreeLevels
```

The total number of levels in the Branch and Bound tree.

Definition at line 66 of file b_and_b_data.h.

8.13 Set Struct Reference

A Set is a node in the Forest.

```
#include <mfset.h>
```

Public Attributes

- struct Set * parentSet

  Pointer to the parent Set in a tree representation of the Forest.
- unsigned short rango

  Rank of the Set, used to optimize the find operation.
- Node curr

  Current Node.
- unsigned short num_in_forest

  Number of the position of the Set in the Forest.

8.13.1 Detailed Description

A Set is a node in the Forest.

Definition at line 19 of file mfset.h.

8.13.2 Member Data Documentation

8.13.2.1 curr

`Node Set::curr`

Current Node.

Definition at line 22 of file mfset.h.

8.13.2.2 num_in_forest

`unsigned short Set::num_in_forest`

Number of the position of the Set in the Forest.

Definition at line 23 of file mfset.h.

8.13.2.3 parentSet

`struct Set* Set::parentSet`

Pointer to the parent Set in a tree representation of the Forest.

Definition at line 20 of file mfset.h.

8.13.2.4 rango

`unsigned short Set::rango`

Rank of the Set, used to optimize the find operation.

Definition at line 21 of file mfset.h.

8.14 SubProblem Struct Reference

The struct used to represent a SubProblem or node of the Branch and Bound tree.

```
#include <b_and_b_data.h>
```

Public Attributes

- BBNodeType type

    The label of the SubProblem.
- unsigned int id

    The id of the SubProblem, an incremental number.
- unsigned int fatherId

    The id of the father of the SubProblem.
- double value

    The cost of the SubProblem.
- unsigned short treeLevel

    The level of the SubProblem in the Branch and Bound tree.
- float timeToReach

    The time needed to reach the SubProblem, in seconds.
- MST oneTree

    The 1Tree of the SubProblem.
- unsigned short num_edges_in_cycle

    The number of edges in the cycle of the SubProblem.
- double prob

    The probability of the SubProblem to be the best tour.
- ConstrainedEdge cycleEdges [MAX_VERTEX_NUM]

    The edges in the cycle of the SubProblem.
- unsigned short num_forbidden_edges

    The number of forbidden edges in the SubProblem.
- unsigned short num_mandatory_edges

    The number of mandatory edges in the SubProblem.
- int edge_to_branch

    The id of the edge to branch in the SubProblem.
- ConstrainedEdge mandatoryEdges [MAX_VERTEX_NUM]

    The mandatory edges in the SubProblem.
- ConstraintType constraints [MAX_VERTEX_NUM][MAX_VERTEX_NUM]

    The constraints of the edges in the SubProblem.

8.14.1 Detailed Description

The struct used to represent a SubProblem or node of the Branch and Bound tree.

Definition at line 42 of file b_and_b_data.h.

8.14.2 Member Data Documentation

8.14.2.1 constraints

`ConstraintType SubProblem::constraints[MAX_VERTEX_NUM][MAX_VERTEX_NUM]`

The constraints of the edges in the SubProblem.

Definition at line 57 of file b_and_b_data.h.

8.14.2.2 cycleEdges

```
ConstrainedEdge SubProblem::cycleEdges[MAX_VERTEX_NUM]
```

The edges in the cycle of the SubProblem.

Definition at line 52 of file b_and_b_data.h.

8.14.2.3 edge_to_branch

```
int SubProblem::edge_to_branch
```

The id of the edge to branch in the SubProblem.

Definition at line 55 of file b_and_b_data.h.

8.14.2.4 fatherId

```
unsigned int SubProblem::fatherId
```

The id of the father of the SubProblem.

Definition at line 45 of file b_and_b_data.h.

8.14.2.5 id

```
unsigned int SubProblem::id
```

The id of the SubProblem, an incremental number.

Definition at line 44 of file b_and_b_data.h.

8.14.2.6 mandatoryEdges

```
ConstrainedEdge SubProblem::mandatoryEdges[MAX_VERTEX_NUM]
```

The mandatory edges in the SubProblem.

Definition at line 56 of file b_and_b_data.h.

8.14.2.7 num_edges_in_cycle

```
unsigned short SubProblem::num_edges_in_cycle
```

The number of edges in the cycle of the SubProblem.

Definition at line 50 of file b_and_b_data.h.

8.14.2.8 num_forbidden_edges

```
unsigned short SubProblem::num_forbidden_edges
```

The number of forbidden edges in the SubProblem.

Definition at line 53 of file b_and_b_data.h.

8.14.2.9 num_mandatory_edges

```
unsigned short SubProblem::num_mandatory_edges
```

The number of mandatory edges in the SubProblem.

Definition at line 54 of file b_and_b_data.h.

8.14.2.10 oneTree

```
MST SubProblem::oneTree
```

The 1Tree of the SubProblem.

Definition at line 49 of file b_and_b_data.h.

8.14.2.11 prob

```
double SubProblem::prob
```

The probability of the SubProblem to be the best tour.

Definition at line 51 of file b_and_b_data.h.

8.14.2.12 timeToReach

```
float SubProblem::timeToReach
```

The time needed to reach the SubProblem, in seconds.

Definition at line 48 of file b_and_b_data.h.

8.14.2.13 treeLevel

```
unsigned short SubProblem::treeLevel
```

The level of the SubProblem in the Branch and Bound tree.

Definition at line 47 of file b_and_b_data.h.

8.14.2.14 type

```
BBNodeType SubProblem::type
```

The label of the SubProblem.

Definition at line 43 of file b_and_b_data.h.

8.14.2.15 value

```
double SubProblem::value
```

The cost of the SubProblem.

Definition at line 46 of file b_and_b_data.h.

8.15 SubProblemElem Struct Reference

The element of the list of SubProblems.

```
#include <b_and_b_data.h>
```

Public Attributes

- SubProblem subProblem

    The SubProblem.
- struct SubProblemElem * next

    The next element of the list.
- struct SubProblemElem * prev

    The previous element of the list.

8.15.1 Detailed Description

The element of the list of SubProblems.

Definition at line 79 of file b_and_b_data.h.

8.15.2 Member Data Documentation

8.15.2.1 next

```
struct SubProblemElem* SubProblemElem::next
```

The next element of the list.

Definition at line 81 of file b_and_b_data.h.

8.15.2.2 prev

```
struct SubProblemElem* SubProblemElem::prev
```

The previous element of the list.

Definition at line 82 of file b_and_b_data.h.

8.15.2.3 subProblem

```
SubProblem SubProblemElem::subProblem
```

The SubProblem.

Definition at line 80 of file b_and_b_data.h.

8.16 SubProblemsList Struct Reference

The list of open SubProblems.

```
#include <b_and_b_data.h>
```

Public Attributes

- SubProblemElem * head

    The head of the list.
- SubProblemElem * tail

    The tail of the list.
- size_t size

    The size of the list.

8.16.1 Detailed Description

The list of open SubProblems.

Definition at line 87 of file b_and_b_data.h.

8.16.2 Member Data Documentation

8.16.2.1 head

```
SubProblemElem* SubProblemsList::head
```

The head of the list.

Definition at line 88 of file b_and_b_data.h.

8.16.2.2 size

```
size_t SubProblemsList::size
```

The size of the list.

Definition at line 90 of file b_and_b_data.h.

8.16.2.3 tail

`SubProblemElem* SubProblemsList::tail`

The tail of the list.

Definition at line 89 of file b_and_b_data.h.

8.17 SubProblemsListIterator Struct Reference

The iterator of the list of SubProblems.

`#include <b_and_b_data.h>`

Public Attributes

- SubProblemsList ∗ list

    The list to iterate.
- SubProblemElem ∗ curr

    The current element of the list.
- size_t index

    The index of the current element of the list.

8.17.1 Detailed Description

The iterator of the list of SubProblems.

Definition at line 95 of file b_and_b_data.h.

8.17.2 Member Data Documentation

8.17.2.1 curr

`SubProblemElem* SubProblemsListIterator::curr`

The current element of the list.

Definition at line 97 of file b_and_b_data.h.

8.17.2.2 index

`size_t SubProblemsListIterator::index`

The index of the current element of the list.

Definition at line 98 of file b_and_b_data.h.

8.17.2.3 list

`SubProblemsList* SubProblemsListIterator::list`

The list to iterate.

Definition at line 96 of file b_and_b_data.h.

Chapter 9

File Documentation

9.1 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/branch_and_bound.c File Reference

The implementation of all the methods used by the Branch and Bound algorithm.

```
#include "branch_and_bound.h"
```

Functions

- void dfs (SubProblem ∗subProblem)

    A Depth First Search algorithm on a Graph.
- bool check_hamiltonian (SubProblem ∗subProblem)

    Function that checks if the 1Tree of a SubProblem is a tour.
- BBNodeType mst_to_one_tree (SubProblem ∗currentSubproblem, Graph ∗graph)

    Transforms a MST into a 1Tree.
- void initialize_matrix (SubProblem ∗subProblem)
- bool infer_constraints (SubProblem ∗subProblem)

    Infer the values of some edge variables of a SubProblem.
- void copy_constraints (SubProblem ∗subProblem, const SubProblem ∗otherSubProblem)

    Copy the matrix of constraints of a SubProblem into another.
- void branch (SubProblemsList ∗openSubProblems, SubProblem ∗currentSubProblem)

    The Shutler's branching rule.
- double max_edge_path_1Tree (SubProblem ∗currentSubProb, double ∗replacement_costs, unsigned short start_node, unsigned short end_node)
- int variable_fixing (SubProblem ∗currentSubProb)

    The function used to fix the edge variables to be mandatory or forbidden.
- void constrained_kruskal (Graph ∗graph, SubProblem ∗subProblem, unsigned short candidateId)
- void constrained_prim (Graph ∗graph, SubProblem ∗subProblem, unsigned short candidateId)
- bool compare_subproblems (const SubProblem ∗a, const SubProblem ∗b)

    Compare two OPEN SubProblems.
- void bound (SubProblem ∗currentSubProb)

    The Held-Karp bound function with the subgradient algorithm.
- bool time_limit_reached (void)

Check if the time limit has been reached.
- void nearest_prob_neighbour (unsigned short start_node)

  This function is used to find the first feasible tour.
- bool compare_candidate_node (SubProblem ∗a, SubProblem ∗b)
- unsigned short find_candidate_node (void)

  Select the candidate Node, i.e. the starting vertex of the tour.
- bool check_feasibility (Graph ∗graph)

  Check if the Graph associated to the Problem is feasible.
- void branch_and_bound (Problem ∗current_problem)

  The Branch and Bound algorithm.
- void set_problem (Problem ∗current_problem)

  Define the problem to solve.
- void print_subProblem (const SubProblem ∗subProblem)

  Get all metrics of a certain SubProblem.
- void print_problem (void)

  Get all metrics of the problem.

9.1.1 Detailed Description

The implementation of all the methods used by the Branch and Bound algorithm.

Author

Lorenzo Sciandra

This file contains all the methods used by the Hybrid and Classic Branch and Bound solver.

Version

1.0.0 @data 2024-05-1

Copyright

Copyright (c) 2024, license MIT

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file branch_and_bound.c.

9.1.2 Function Documentation

9.1.2.1 bound()

```
void bound (
            SubProblem * currentSubProb )
```

The Held-Karp bound function with the subgradient algorithm.

This function has a primal and dual behaviour: after the minimal 1Tree is found, a subgradient algorithm is used to do a dual ascent of the Lagrangian relaxation. More details at https://www.sciencedirect.com/science/article/abs/pii/S0377221796002147?via%3Dihub.

Parameters

current_problem	The pointer to the SubProblem or branch-and-bound Node in the tree.

Definition at line 657 of file branch_and_bound.c.

9.1.2.2 branch()

```
void branch (
            SubProblemsList * openSubProblems,
            SubProblem * subProblem )
```

The Shutler's branching rule.

Every SubProblem is branched into 2 new SubProblems, one including the "edge_to_branch" and the other not. More details at `http://www.jstor.org/stable/254144`.

Parameters

openSubProblems	The list of open SubProblems, to which the new SubProblems will be added.
subProblem	The SubProblem to branch.

Definition at line 256 of file branch_and_bound.c.

9.1.2.3 branch_and_bound()

```
void branch_and_bound (
            Problem * current_problem )
```

The Branch and Bound algorithm.

This is the main function of the Branch and Bound algorithm. It stores all the open SubProblems in a SubProblemsList and analyzes them one by one with the branch() and held_karp_bound() functions.

Parameters

current_problem	The pointer to the problem to solve.

Definition at line 1012 of file branch_and_bound.c.

9.1.2.4 check_feasibility()

```
bool check_feasibility (
            Graph * graph )
```

Check if the Graph associated to the Problem is feasible.

A Graph is feasible if every Node has at least degree 2.

Parameters

graph	The Graph to check.

Returns

true if the Graph is feasible, false otherwise.

Definition at line 997 of file branch_and_bound.c.

9.1.2.5 check_hamiltonian()

```
bool check_hamiltonian (
            SubProblem * subProblem )
```

Function that checks if the 1Tree of a SubProblem is a tour.

This is done by simply check if all the edges are in the cycle passing through the candidate Node.

Parameters

subProblem	The SubProblem to check.

Returns

true if the SubProblem is a Hamiltonian cycle, false otherwise.

Definition at line 82 of file branch_and_bound.c.

9.1.2.6 compare_candidate_node()

```
bool compare_candidate_node (
            SubProblem * a,
            SubProblem * b )
```

Definition at line 951 of file branch_and_bound.c.

9.1.2.7 compare_subproblems()

```
bool compare_subproblems (
            const SubProblem * a,
            const SubProblem * b )
```

Compare two OPEN SubProblems.

This function is used to sort the SubProblems in the open list to define its order.

Parameters

a	The first SubProblem to compare.
b	The second SubProblem to compare.

Returns

true if the first SubProblem is better than the second, false otherwise.

Definition at line 647 of file branch_and_bound.c.

9.1.2.8 constrained_kruskal()

```
void constrained_kruskal (
            Graph * graph,
            SubProblem * subProblem,
            unsigned short candidateId )
```

Definition at line 503 of file branch_and_bound.c.

9.1.2.9 constrained_prim()

```
void constrained_prim (
            Graph * graph,
            SubProblem * subProblem,
            unsigned short candidateId )
```

Definition at line 565 of file branch_and_bound.c.

9.1.2.10 copy_constraints()

```
void copy_constraints (
            SubProblem * subProblem,
            const SubProblem * otherSubProblem )
```

Copy the matrix of constraints of a SubProblem into another.

This function is used when a SubProblem is branched into two new SubProblems, and the constraints of the father SubProblem are copied into the sons.

Parameters

subProblem	The SubProblem to which the ConstraintType will be copied.
otherSubProblem	The SubProblem from which the ConstraintType will be copied.

Definition at line 234 of file branch_and_bound.c.

9.1.2.11 dfs()

```
void dfs (
            SubProblem * subProblem )
```

A Depth First Search algorithm on a Graph.

This function is used to find the cycle in the 1Tree SubProblem, passing through the candidate Node.

Parameters

subProblem	The SubProblem to inspect.

Definition at line 18 of file branch_and_bound.c.

9.1.2.12 find_candidate_node()

```
unsigned short find_candidate_node (
            void )
```

Select the candidate Node, i.e. the starting vertex of the tour.

Every Node is tried and the one with the best lower bound is chosen. In the Hybrid mode, when two nodes have the same lower bound, the one with the best probability is chosen.

Returns

the candidate Node id.

Definition at line 961 of file branch_and_bound.c.

9.1.2.13 infer_constraints()

```
bool infer_constraints (
            SubProblem * subProblem )
```

Infer the values of some edge variables of a SubProblem.

According to the constraints of the father SubProblem and the one added to the son, we can infer new variables values in order to check if the SubProblem is still feasible or not.

Parameters

subProblem	The SubProblem to which we want to infer the variables values.

Returns

true if the subproblem remains feasible, false otherwise.

Definition at line 185 of file branch_and_bound.c.

9.1.2.14 initialize_matrix()

```
void initialize_matrix (
            SubProblem * subProblem )
```

Definition at line 167 of file branch_and_bound.c.

9.1.2.15 max_edge_path_1Tree()

```
double max_edge_path_1Tree (
            SubProblem * currentSubProb,
            double * replacement_costs,
            unsigned short start_node,
            unsigned short end_node )
```

Definition at line 342 of file branch_and_bound.c.

9.1.2.16 mst_to_one_tree()

```
BBNodeType mst_to_one_tree (
            SubProblem * currentSubproblem,
            Graph * graph )
```

Transforms a MST into a 1Tree.

This is done by adding the two least-cost edges incident to the candidate Node in the MST.

Parameters

currentSubproblem	The SubProblem to which the MST belongs.
graph	The Graph of the Problem.

Returns

an enum value that indicates if the SubProblem is feasible or not.

Definition at line 88 of file branch_and_bound.c.

9.1.2.17 nearest_prob_neighbour()

```
void nearest_prob_neighbour (
            unsigned short start_node )
```

This function is used to find the first feasible tour.

If the Hybrid mode is disabled, it is the simple nearest neighbour algorithm. Otherwise, it also implements the Probabilistic Nearest Neighbour algorithm where, starting from a Node, the Edge with the best probability is chosen. This method is repeated by choosing every Node as the starting Node. The best tour found is stored as the best tour found so far.

Parameters

start_node	The Node from which the tour will start.

Definition at line 844 of file branch_and_bound.c.

9.1.2.18 print_problem()

```
void print_problem (
            void )
```

Get all metrics of the problem.

It is used at the end of the algorithm to print the solution obtained. It calls the print_subProblem() function on the best SubProblem found.

Definition at line 1138 of file branch_and_bound.c.

9.1.2.19 print_subProblem()

```
void print_subProblem (
            const SubProblem * subProblem )
```

Get all metrics of a certain SubProblem.

It is used at the end of the algorithm to print the solution obtained.

Parameters

subProblem	The SubProblem to print.

Definition at line 1073 of file branch_and_bound.c.

9.1.2.20 set_problem()

```
void set_problem (
            Problem * current_problem )
```

Define the problem to solve.

This function is used to set the pointer to the problem to solve.

Parameters

current_problem	The pointer to the problem to solve.

Definition at line 1068 of file branch_and_bound.c.

9.1.2.21 time_limit_reached()

```
bool time_limit_reached (
            void )
```

Check if the time limit has been reached.

This function is used to check if the time limit has been reached.

Returns

true if the time limit has been reached, false otherwise.

Definition at line 839 of file branch_and_bound.c.

9.1.2.22 variable_fixing()

```
int variable_fixing (
            SubProblem * subProblem )
```

The function used to fix the edge variables to be mandatory or forbidden.

By calculating the calculating of marginal and replacement costs, the edge variables are fixed to be forbidden or mandatory. More details at https://link.springer.com/chapter/10.1007/978-3-642-13520-0↩_6.

Parameters

subProblem	The SubProblem that we want to add the constraints to.

Returns

the num of variables fixed.

Definition at line 421 of file branch_and_bound.c.

9.2 branch_and_bound.c

Go to the documentation of this file.

```
#include "branch_and_bound.h"


void dfs(SubProblem *subProblem) {
    List *stack = new_list();
    unsigned short num_nodes = subProblem->oneTree.num_nodes;
    Node start;
    int parentId[num_nodes];
    unsigned short pathLength[num_nodes];
    bool visited[num_nodes];

    for (unsigned short i = 0; i < num_nodes; i++) {
        Node current = subProblem->oneTree.nodes[i];
        if (current.positionInGraph == problem->graph.nodes[problem->candidateNodeId].positionInGraph) {
            start = current;
        }
        parentId[i] = -1;
        pathLength[i] = 0;
        visited[i] = false;
    }

    add_elem_list_bottom(stack, &start);

    while (stack->size > 0 && parentId[start.positionInGraph] == -1) {
        Node *currentNode = get_list_elem_index(stack, 0);
        delete_list_elem_index(stack, 0);
        if (!visited[currentNode->positionInGraph]) {
            visited[currentNode->positionInGraph] = true;
            for (unsigned short i = 0; i < currentNode->num_neighbours; i++) {
                unsigned short dest = currentNode->neighbours[i];
                if (!visited[dest]) {
                    pathLength[dest] = pathLength[currentNode->positionInGraph] + 1;
                    parentId[dest] = currentNode->positionInGraph;
                    Node *neighbour = &subProblem->oneTree.nodes[dest];
                    add_elem_list_index(stack, neighbour, 0);
                } else if (parentId[dest] == -1) {
                    // start node
                    unsigned short path = pathLength[currentNode->positionInGraph] + 1;
                    if (path > 2) {
                        parentId[dest] = currentNode->positionInGraph;
                        pathLength[dest] = path;
                    }
                }
            }
        }
    }
    del_list(stack);

    int fromNode = -1;
    int toNode = start.positionInGraph;

    //printf("Path Length: %d\n", pathLength[toNode]);

    if (pathLength[toNode] > 2) {
        while (fromNode != start.positionInGraph) {
            fromNode = parentId[toNode];
            Edge current_in_cycle = problem->graph.edges_matrix[fromNode][toNode];
            subProblem->cycleEdges[subProblem->num_edges_in_cycle].src = current_in_cycle.src;
            subProblem->cycleEdges[subProblem->num_edges_in_cycle].dest = current_in_cycle.dest;
```

```
            subProblem->num_edges_in_cycle++;
            toNode = fromNode;
        }
    }

}


bool check_hamiltonian(SubProblem *subProblem) {
    dfs(subProblem);
    return subProblem->num_edges_in_cycle == subProblem->oneTree.num_edges;
}


BBNodeType mst_to_one_tree(SubProblem *currentSubproblem, Graph *graph) {

    Node candidate = graph->nodes[problem->candidateNodeId];
    int bestEdgesPos[2];
    double bestEdgesWeight[2];
    unsigned short src = candidate.positionInGraph;
    unsigned short others [candidate.num_neighbours];
    unsigned short num_others = 0;
    unsigned short toAdd = 0;

    for (unsigned short i = 0; i < candidate.num_neighbours && toAdd<=2; i++) {
        unsigned short dest = candidate.neighbours[i];

        if (currentSubproblem->num_mandatory_edges > 0 &&
                currentSubproblem->constraints[src][dest] == MANDATORY) {
            bestEdgesWeight[toAdd] = graph->edges_matrix[src][dest].weight;
            bestEdgesPos[toAdd] = graph->edges_matrix[src][dest].positionInGraph;
            toAdd++;
        } else if (currentSubproblem->num_forbidden_edges == 0 ||
                currentSubproblem->constraints[src][dest] == NOTHING) {
            others[num_others] = dest;
            num_others++;
        }
    }
    if (toAdd > 2 || (toAdd + num_others) < 2) {
        return CLOSED_UNFEASIBLE;
    } else if (toAdd == 2){
        add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[0]]);
        add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[1]]);
        return OPEN;
    } else if(toAdd == 1){
        add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[0]]);

        double bestFoundWeight = INFINITE;
        int bestFoundPos = -1;

        for (unsigned short j = 0; j < num_others; j++) {
            unsigned short dest = others[j];
            Edge candidateEdge = graph->edges_matrix[src][dest];

            if (bestFoundWeight > candidateEdge.weight) {
                bestFoundPos = candidateEdge.positionInGraph;
                bestFoundWeight = candidateEdge.weight;
            }
        }

        add_edge(&currentSubproblem->oneTree, &graph->edges[bestFoundPos]);

        return OPEN;
    } else{

        bestEdgesPos[0] = -1;
        bestEdgesPos[1] = -1;
        bestEdgesWeight[0] = INFINITE;
        bestEdgesWeight[1] = INFINITE;

        for (unsigned short j = 0; j < num_others; j++) {
            unsigned short dest = others[j];
            Edge candidateEdge = graph->edges_matrix[src][dest];

            if (bestEdgesWeight[0] > candidateEdge.weight) {
                bestEdgesPos[1] = bestEdgesPos[0];
                bestEdgesWeight[1] = bestEdgesWeight[0];
                bestEdgesPos[0] = candidateEdge.positionInGraph;
                bestEdgesWeight[0] = candidateEdge.weight;
            } else if (bestEdgesWeight[1] > candidateEdge.weight) {
                bestEdgesPos[1] = candidateEdge.positionInGraph;
                bestEdgesWeight[1] = candidateEdge.weight;
            }
        }

        add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[0]]);
        add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[1]]);
```

```
        return OPEN;
    }
}


void initialize_matrix(SubProblem *subProblem) {

    subProblem->num_mandatory_edges = 0;
    subProblem->num_forbidden_edges = 0;
    for (short i = 0; i < MAX_VERTEX_NUM; i++) {
        for (short j = i; j < MAX_VERTEX_NUM; j++) {
            if(j == i){
                subProblem->constraints[i][j] = FORBIDDEN;

            } else{
                subProblem->constraints[i][j] = NOTHING;
                subProblem->constraints[j][i] = NOTHING;
            }
        }
    }
}


bool infer_constraints(SubProblem * subProblem){

    bool valid = true;

    for (short i = 0; i < MAX_VERTEX_NUM && valid; i++) {

        short num_nothing_node = 0;
        short num_mandatory_node = 0;
        short num_forbidden_node = 0;
        short nothing_nodes[MAX_VERTEX_NUM];

        for (short j = 0; j < MAX_VERTEX_NUM; j++) {
            if (subProblem->constraints[i][j] == NOTHING) {
                nothing_nodes[num_nothing_node] = j;
                num_nothing_node++;
            } else if (subProblem->constraints[i][j] == MANDATORY) {
                num_mandatory_node++;
            } else {
                num_forbidden_node++;
            }
        }

        if (num_mandatory_node == 2) {
            for (short j = 0; j < num_nothing_node; j++) {
                subProblem->constraints[i][nothing_nodes[j]] = FORBIDDEN;
                subProblem->constraints[nothing_nodes[j]][i] = FORBIDDEN;
                subProblem->num_forbidden_edges++;
                problem->num_fixed_edges++;
            }
        } else if (num_mandatory_node > 2 || (MAX_VERTEX_NUM - num_forbidden_node < 2)) {
            valid = false;
        } else if (MAX_VERTEX_NUM - num_forbidden_node == 2) {

            if (num_nothing_node + num_mandatory_node == 2) {
                for (short j = 0; j < num_nothing_node; j++) {
                    subProblem->constraints[i][nothing_nodes[j]] = MANDATORY;
                    subProblem->constraints[nothing_nodes[j]][i] = MANDATORY;
                    subProblem->mandatoryEdges[subProblem->num_mandatory_edges].src = i;
                    subProblem->mandatoryEdges[subProblem->num_mandatory_edges].dest =
nothing_nodes[j];
                    subProblem->num_mandatory_edges++;
                    problem->num_fixed_edges++;
                }
            }
        }
    }
    return valid;
}


void copy_constraints(SubProblem *subProblem, const SubProblem *otherSubProblem) {
    subProblem->num_mandatory_edges = 0;
    subProblem->num_forbidden_edges = 0;
    for (short i = 0; i < MAX_VERTEX_NUM; i++) {

        for (short j = i; j < MAX_VERTEX_NUM; j++) {
            subProblem->constraints[i][j] = otherSubProblem->constraints[i][j];
            subProblem->constraints[j][i] = otherSubProblem->constraints[j][i];

            if(subProblem->constraints[i][j] == MANDATORY){
                subProblem->mandatoryEdges[subProblem->num_mandatory_edges].src = i;
                subProblem->mandatoryEdges[subProblem->num_mandatory_edges].dest = j;
                subProblem->num_mandatory_edges++;
```

```
            }
            else if(subProblem->constraints[i][j] == FORBIDDEN){
                subProblem->num_forbidden_edges++;
            }
        }
    }
}


void branch(SubProblemsList *openSubProblems, SubProblem *currentSubProblem){

    if (currentSubProblem->treeLevel + 1 > problem->totTreeLevels) {
        problem->totTreeLevels++;
    }

    Edge to_branch = problem->graph.edges[currentSubProblem->edge_to_branch];
    double prob_edge = to_branch.prob;

    for (short i = 0; i < 2; i++) {
        problem->generatedBBNodes++;
        SubProblem child;
        child.num_edges_in_cycle = 0;
        child.type = OPEN;
        child.prob = currentSubProblem->prob;
        child.id = problem->generatedBBNodes;
        child.fatherId = currentSubProblem->id;
        child.value = currentSubProblem->value;
        child.treeLevel = currentSubProblem->treeLevel + 1;
        child.edge_to_branch = -1;
        copy_constraints(&child, currentSubProblem);

        if(HYBRID) {

            if (prob_edge >= PROB_BRANCH) {
                if (i == 0) {
                    child.constraints[to_branch.src][to_branch.dest] = MANDATORY;
                    child.constraints[to_branch.dest][to_branch.src] = MANDATORY;
                    child.mandatoryEdges[child.num_mandatory_edges].src = to_branch.src;
                    child.mandatoryEdges[child.num_mandatory_edges].dest = to_branch.dest;
                    child.num_mandatory_edges++;
                } else {
                    child.constraints[to_branch.src][to_branch.dest] = FORBIDDEN;
                    child.constraints[to_branch.dest][to_branch.src] = FORBIDDEN;
                    child.num_forbidden_edges++;
                }
            } else {
                if (i == 0) {
                    child.constraints[to_branch.src][to_branch.dest] = FORBIDDEN;
                    child.constraints[to_branch.dest][to_branch.src] = FORBIDDEN;
                    child.num_forbidden_edges++;
                } else {
                    child.constraints[to_branch.src][to_branch.dest] = MANDATORY;
                    child.constraints[to_branch.dest][to_branch.src] = MANDATORY;
                    child.mandatoryEdges[child.num_mandatory_edges].src = to_branch.src;
                    child.mandatoryEdges[child.num_mandatory_edges].dest = to_branch.dest;
                    child.num_mandatory_edges++;
                }
            }
        }
        else{
            if (i == 0) {
                child.constraints[to_branch.src][to_branch.dest] = MANDATORY;
                child.constraints[to_branch.dest][to_branch.src] = MANDATORY;
                child.mandatoryEdges[child.num_mandatory_edges].src = to_branch.src;
                child.mandatoryEdges[child.num_mandatory_edges].dest = to_branch.dest;
                child.num_mandatory_edges++;
            } else {
                child.constraints[to_branch.src][to_branch.dest] = FORBIDDEN;
                child.constraints[to_branch.dest][to_branch.src] = FORBIDDEN;
                child.num_forbidden_edges++;
            }
        }

        if(infer_constraints(&child)){
            long position = -1;

            SubProblemsListIterator *subProblem_iterators = create_SubProblemList_iterator(
                    openSubProblems);
            for (size_t j = 0; j < openSubProblems->size && position == -1; j++) {
                SubProblem *open_subProblem = SubProblemList_iterator_get_next(subProblem_iterators);
                if (compare_subproblems(&child, open_subProblem)) {
                    position = (long) j;
                }
            }
            delete_SubProblemList_iterator(subProblem_iterators);
            if (position == -1) {
                add_elem_SubProblemList_bottom(openSubProblems, &child);
```

```
            } else {
                add_elem_SubProblemList_index(openSubProblems, &child, position);
            }
        }
    }
}


double max_edge_path_1Tree(SubProblem *currentSubProb, double *replacement_costs,
                           unsigned short start_node, unsigned short end_node){
    List *stack = new_list();
    unsigned short num_nodes = currentSubProb->oneTree.num_nodes;
    Node start;

    bool visited[num_nodes];
    int parentId[num_nodes];

    for (unsigned short i = 0; i < num_nodes; i++) {
        Node current = currentSubProb->oneTree.nodes[i];
        if (current.positionInGraph == start_node) {
            start = current;
        }
        visited[i] = false;
        parentId[i] = -1;
    }

    add_elem_list_bottom(stack, &start);

    while (stack->size > 0 && parentId[end_node] == -1) {
        Node *currentNode = get_list_elem_index(stack, 0);
        delete_list_elem_index(stack, 0);
        if (!visited[currentNode->positionInGraph]) {
            visited[currentNode->positionInGraph] = true;
            for (unsigned short i = 0; i < currentNode->num_neighbours; i++) {
                unsigned short dest = currentNode->neighbours[i];
                if (!visited[dest] && problem->candidateNodeId != dest) {
                    parentId[dest] = currentNode->positionInGraph;
                    Node *neighbour = &currentSubProb->oneTree.nodes[dest];
                    add_elem_list_index(stack, neighbour, 0);
                }
            }
        }
    }
    del_list(stack);


    unsigned short fromNode = -1;
    unsigned short toNode = end_node;
    double max_edge_weight = -1;
    double edge_nin_1tree_weight = problem->graph.edges_matrix[start_node][end_node].weight;

    while (fromNode != start_node) {
        fromNode = parentId[toNode];

        short position_in_1_tree = -1;

        if(currentSubProb->oneTree.edges_matrix[fromNode][toNode] != -1){

            if(currentSubProb->constraints[fromNode][toNode] == NOTHING) {

                position_in_1_tree = currentSubProb->oneTree.edges_matrix[fromNode][toNode];

                Edge current_in_path = currentSubProb->oneTree.edges[position_in_1_tree];
                double current_edge_weight =
problem->graph.edges_matrix[current_in_path.src][current_in_path.dest].weight;

                if (current_edge_weight > max_edge_weight) {
                    max_edge_weight = current_edge_weight;
                }

                if (edge_nin_1tree_weight < current_edge_weight +
replacement_costs[current_in_path.positionInGraph]) {
                    replacement_costs[current_in_path.positionInGraph] = edge_nin_1tree_weight -
current_edge_weight;
                }
            }

            toNode = fromNode;

        } else{
            printf("ERROR: Edge not found in 1-tree\n");
            exit(1);
        }
    }


    return max_edge_weight == -1 ? -INFINITE: max_edge_weight;
```

```
}


int variable_fixing(SubProblem *currentSubProb){
    int num_fixed = 0;
    double replacement_costs [MAX_VERTEX_NUM - 2]; // for all the edges in the 1-tree except the best replacement to maintain the 1-tree
    double best_candidate_replacement = INFINITE;

    for (int i = 0; i < MAX_VERTEX_NUM - 2; i++){
        replacement_costs[i] = INFINITE;
    }

    for (unsigned int i = 0; i < problem->graph.num_edges; i++){
        Edge current_edge = problem->graph.edges[i];

        if(currentSubProb->constraints[current_edge.src][current_edge.dest] == NOTHING){
            if(currentSubProb->oneTree.edges_matrix[current_edge.src][current_edge.dest] == -1){

                double max_edge_path = 0;

                if(current_edge.dest != problem->candidateNodeId && current_edge.src != problem->candidateNodeId){
                    max_edge_path = max_edge_path_1Tree(currentSubProb, replacement_costs,current_edge.src, current_edge.dest);
                }

                else{

                    double first_candidate_edge_weight = problem->graph.edges_matrix[problem->candidateNodeId][currentSubProb->oneTree.nodes[problem->candidateNodeId].neighbours
                    double second_candidate_edge_weight = problem->graph.edges_matrix[problem->candidateNodeId][currentSubProb->oneTree.nodes[problem->candidateNodeId].neighbours

                    if (first_candidate_edge_weight > second_candidate_edge_weight){
                        max_edge_path = first_candidate_edge_weight;
                    } else{
                        max_edge_path = second_candidate_edge_weight;
                    }
                }

                if (currentSubProb->oneTree.cost + current_edge.weight - max_edge_path >= problem->bestValue){
                    currentSubProb->constraints[current_edge.src][current_edge.dest] = FORBIDDEN;
                    currentSubProb->constraints[current_edge.dest][current_edge.src] = FORBIDDEN;
                    currentSubProb->num_forbidden_edges++;
                    num_fixed++;
                }

            }

            if(current_edge.src == problem->candidateNodeId || current_edge.dest == problem->candidateNodeId){
                if(current_edge.weight < best_candidate_replacement){
                    best_candidate_replacement = current_edge.weight;
                }
            }
        }
    }


    for (int i = 0; i < MAX_VERTEX_NUM; i++){
        Edge edge_1tree = currentSubProb->oneTree.edges[i];
        double edge_1tree_weight = problem->graph.edges_matrix[edge_1tree.src][edge_1tree.dest].weight;

        if(currentSubProb->constraints[edge_1tree.src][edge_1tree.dest] == NOTHING) {
            if (edge_1tree.src != problem->candidateNodeId && edge_1tree.dest != problem->candidateNodeId) {
                if (currentSubProb->oneTree.cost + replacement_costs[i] - edge_1tree_weight >= problem->bestValue) {
                    currentSubProb->constraints[edge_1tree.src][edge_1tree.dest] = MANDATORY;
                    currentSubProb->constraints[edge_1tree.dest][edge_1tree.src] = MANDATORY;
                    currentSubProb->mandatoryEdges[currentSubProb->num_mandatory_edges].src = edge_1tree.src;
                    currentSubProb->mandatoryEdges[currentSubProb->num_mandatory_edges].dest = edge_1tree.dest;
                    currentSubProb->num_mandatory_edges++;
                    num_fixed++;
                }
            } else {
                if(currentSubProb->oneTree.cost + best_candidate_replacement - edge_1tree_weight >= problem->bestValue){
                    currentSubProb->constraints[edge_1tree.src][edge_1tree.dest] = MANDATORY;
                    currentSubProb->constraints[edge_1tree.dest][edge_1tree.src] = MANDATORY;
                    currentSubProb->mandatoryEdges[currentSubProb->num_mandatory_edges].src = edge_1tree.src;
```

```
                    currentSubProb->mandatoryEdges[currentSubProb->num_mandatory_edges].dest = edge_1tree.dest;
                    currentSubProb->num_mandatory_edges++;
                    num_fixed++;
                }
            }
        }
    }

    return num_fixed;
}


void constrained_kruskal(Graph * graph, SubProblem * subProblem, unsigned short candidateId) {
    create_mst(&subProblem->oneTree,graph->nodes, graph->num_nodes);
    Forest forest;
    create_forest_constrained(&forest, graph->nodes, graph->num_nodes, candidateId);
    wrap_quick_sort(graph);

    unsigned short num_edges_inMST = 0;
    for (unsigned short i = 0; i < subProblem->num_mandatory_edges; i++) {
        ConstrainedEdge current_mandatory = subProblem->mandatoryEdges[i];
        Edge mandatory_edge = graph->edges_matrix[current_mandatory.src][current_mandatory.dest];
        unsigned short src = mandatory_edge.src;
        unsigned short dest = mandatory_edge.dest;

        if (src != candidateId && dest != candidateId) {

            Set *set1_root = find(&forest.sets[src]);
            Set *set2_root = find(&forest.sets[dest]);
            if (set1_root->num_in_forest != set2_root->num_in_forest) {
                merge(set1_root, set2_root);
                // add the edge to the MST
                add_edge(&subProblem->oneTree, &mandatory_edge);
                num_edges_inMST++;
            }
        }
    }

    unsigned short num_edges_inG = 0;

    while (num_edges_inG < graph->num_edges && num_edges_inMST < graph->num_nodes - 2) {

        Edge current_edge = graph->edges[num_edges_inG];

        unsigned short src = current_edge.src;
        unsigned short dest = current_edge.dest;

        if (src != candidateId && dest != candidateId){

            if (subProblem->num_forbidden_edges == 0 || subProblem->constraints[src][dest] != FORBIDDEN) {

                Set *set1_root = find(&forest.sets[src]);
                Set *set2_root = find(&forest.sets[dest]);

                if (set1_root->num_in_forest != set2_root->num_in_forest) {
                    merge(set1_root, set2_root);
                    // add the edge to the MST
                    add_edge(&subProblem->oneTree, &current_edge);
                    num_edges_inMST++;
                }
            }
        }

        num_edges_inG++;
    }

    if (num_edges_inMST == graph->num_nodes - 2) {
        subProblem->oneTree.isValid = true;
    } else {
        subProblem->oneTree.isValid = false;
    }
}


void constrained_prim(Graph * graph, SubProblem * subProblem, unsigned short candidateId) {

    Graph graph_copy = *graph;
    create_mst(&subProblem->oneTree,graph->nodes, graph->num_nodes);
    FibonacciHeap heap;
    create_fibonacci_heap(&heap);

    OrdTreeNode tree_nodes[graph->num_nodes];
    bool in_heap [graph->num_nodes];
    double values [graph->num_nodes];
    int fathers [graph->num_nodes];
```

```

    bool start = true;

    for (unsigned short i = 0; i < graph->num_nodes; i++){
        in_heap[i] = false;
        fathers[i] = -1;
        values[i] = start ? FLT_MAX/2 : FLT_MAX;
        start = false;
    }

    for (unsigned short i = 0; i < subProblem->num_mandatory_edges; i++){
        ConstrainedEdge current_mandatory = subProblem->mandatoryEdges[i];
        unsigned short src = current_mandatory.src;
        unsigned short dest = current_mandatory.dest;

        if(src != candidateId && dest != candidateId) {

            graph_copy.edges_matrix[src][dest].weight = -((double) problem->graph.num_nodes);
            graph_copy.edges_matrix[dest][src].weight = -((double) problem->graph.num_nodes);
            graph_copy.edges[graph->edges_matrix[src][dest].positionInGraph].weight = -((double) problem->graph.num_nodes);
        }
    }


    for (unsigned short i = 0; i < graph_copy.num_nodes; i++) {
        if(i != candidateId){
            create_insert_node(&heap, &tree_nodes[i], i, values[i]);
            in_heap[i] = true;
        }
    }

    while(heap.num_nodes != 0){
        int min_pos = extract_min(&heap);
        if(min_pos == -1){
            fprintf(stderr, "Error: min_pos == -1\n");
            exit(1);
        }
        else{
            in_heap[min_pos] = false;
            for(unsigned short i = 0; i < graph_copy.nodes[min_pos].num_neighbours; i++){
                unsigned short neigh = graph_copy.nodes[min_pos].neighbours[i];
                if(neigh != min_pos && in_heap[neigh]) {
                    double weight = graph_copy.edges_matrix[min_pos][neigh].weight;
                    if (weight < tree_nodes[neigh].value){
                        if (subProblem->num_forbidden_edges == 0 || subProblem->constraints[min_pos][neigh] != FORBIDDEN){
                            fathers[neigh] = min_pos;
                            decrease_value(&heap, &tree_nodes[neigh], weight);
                        }
                    }
                }
            }
        }
    }

    for(unsigned short i = 0; i < graph->num_nodes; i++){
        if(fathers[i] != -1){
            add_edge(&subProblem->oneTree, &graph->edges_matrix[i][fathers[i]]);
        }
    }

    if(subProblem->oneTree.num_edges == graph->num_nodes - 2){
        subProblem->oneTree.isValid = true;
    }
    else{
        subProblem->oneTree.isValid = false;
    }

}


// a better than b?
bool compare_subproblems(const SubProblem *a, const SubProblem *b) {
    if (HYBRID) {
        return (b->value - a->value) > EPSILON ||
               ( (b->value > a->value) && (a->prob - b->prob) >= BETTER_PROB);
    } else {
        return (b->value - a->value) > EPSILON;
    }
}


void bound(SubProblem *currentSubProb) {

    if ((problem->bestSolution.value - currentSubProb->value) > EPSILON || currentSubProb->treeLevel == 1) {
```

```
        currentSubProb->timeToReach = ((float) (clock() - problem->start)) / CLOCKS_PER_SEC;
        problem->exploredBBNodes++;

        double pi[MAX_VERTEX_NUM] = {0};
        double v[MAX_VERTEX_NUM] = {0};
        double v_old[MAX_VERTEX_NUM] = {0};
        double total_pi = 0;
        int k = 10;
        int max_iter = currentSubProb->treeLevel == 1 ? (int) NUM_HK_INITIAL_ITERATIONS : (int) NUM_HK_ITERATIONS;
        max_iter += k;
        double best_lower_bound = 0;
        double t_0;
        SubProblemsList generatedSubProblems;
        new_SubProblemList(&generatedSubProblems);
        Graph *used_graph = &problem->graph;
        bool first_iter = true;

        double prob_branch [MAX_EDGES_NUM] = {0};

        for (int iter = 1; iter <= max_iter; iter++) {

            SubProblem analyzedSubProblem = *currentSubProb;
            BBNodeType type;
            if(!first_iter) {
                for (unsigned short j = 0; j < problem->graph.num_edges; j++) {
                    if ((pi[used_graph->edges[j].src] +
                         pi[used_graph->edges[j].dest]) != 0) {
                        used_graph->edges[j].weight += (pi[used_graph->edges[j].src] +
                                                        pi[used_graph->edges[j].dest]);
                        used_graph->edges_matrix[used_graph->edges[j].src][used_graph->edges[j].dest].weight = used_graph->edges[j].weight;
                        used_graph->edges_matrix[used_graph->edges[j].dest][used_graph->edges[j].src].weight = used_graph->edges[j].weight;
                        used_graph->orderedEdges = false;
                    }
                }
            }

            if(PRIM){
                constrained_prim(used_graph, &analyzedSubProblem, problem->candidateNodeId);
            }
            else{
                constrained_kruskal(used_graph, &analyzedSubProblem, problem->candidateNodeId);
            }

            if (analyzedSubProblem.oneTree.isValid) {
                type = mst_to_one_tree(&analyzedSubProblem, used_graph);

                if (type == OPEN) {
                    analyzedSubProblem.prob = analyzedSubProblem.oneTree.prob;
                    analyzedSubProblem.type = type;
                    analyzedSubProblem.value = 0;

                    for (int e = 0; e < problem->graph.num_nodes; e++) {
                        Edge *edge = &analyzedSubProblem.oneTree.edges[e];
                        analyzedSubProblem.value += problem->graph.edges_matrix[edge->src][edge->dest].weight;

                        if (iter > max_iter - k) {
                            prob_branch[problem->graph.edges_matrix[edge->src][edge->dest].positionInGraph] =
                                ((prob_branch[problem->graph.edges_matrix[edge->src][edge->dest].positionInGraph] *
                                                 (double) k) + 1) / ((double) k);
                        }
                    }

                    if (problem->bestValue - analyzedSubProblem.value <= EPSILON) {
                        analyzedSubProblem.type = CLOSED_BOUND;
                    } else {
                        analyzedSubProblem.num_edges_in_cycle = 0;
                        if (check_hamiltonian(&analyzedSubProblem)) {
                            problem->bestValue = analyzedSubProblem.value;
                            analyzedSubProblem.type = iter == 1 ? CLOSED_1TREE: CLOSED_SUBGRADIENT;
                            problem->bestSolution = analyzedSubProblem;
                            printf("Updated best value: %f, at time: %f\n", problem->bestValue, ((float) (clock() - problem->start)) / CLOCKS_PER_SEC);
                        } else {
                            analyzedSubProblem.type = OPEN;
                        }
                    }

                    if(analyzedSubProblem.type != OPEN){
                        if(iter == 1 || (analyzedSubProblem.type == CLOSED_SUBGRADIENT)) {
```

```
                            currentSubProb->type = analyzedSubProblem.type;
                            return;
                        }
                    }

                    double current_lb = analyzedSubProblem.oneTree.cost - (2 * total_pi);

                    if (current_lb > best_lower_bound || first_iter) {
                        best_lower_bound = current_lb;
                        if (first_iter) {
                            first_iter = false;
                            used_graph = &problem->reformulationGraph;
                        }

                        t_0 = best_lower_bound / (2 * MAX_VERTEX_NUM);
                    }

                    // change the graph to the original one, because the dual variables are calculated on the original graph
                    *used_graph = problem->graph;
                    add_elem_SubProblemList_bottom(&generatedSubProblems, &analyzedSubProblem);

                    for (unsigned short i = 0; i < problem->graph.num_nodes; i++) {
                        v[i] = (double) (analyzedSubProblem.oneTree.nodes[i].num_neighbours - 2);
                    }

                    double t =
                            (((double) (iter - 1)) * (((double) (2 * max_iter) - 5) / (2 * (double) (max_iter - 1))) * t_0)
                            - (((double) iter - 2) * t_0) +
                            ((t_0 * ((double) iter - 1) * ((double) iter - 2)) /
                             (2 * ((double) max_iter - 1) * ((double) max_iter - 2)));

                    total_pi = 0;

                    for (unsigned short j = 0; j < problem->graph.num_nodes; j++) {
                        if (v[j] != 0) {
                            pi[j] += (double) ((0.6 * t * v[j]) + (0.4 * t * v_old[j]));
                        }
                        v_old[j] = v[j];
                        total_pi += pi[j];
                    }
                }

                else{
                    currentSubProb->type = CLOSED_UNFEASIBLE;
                    return;
                }
            } else {
                currentSubProb->type = CLOSED_UNFEASIBLE;
                return;
            }

        }

        *currentSubProb = *get_SubProblemList_elem_index(&generatedSubProblems,0);
        currentSubProb->oneTree.cost = currentSubProb->value; // mi serve valore originale per variabile fixing
        SubProblem best_subproblem = *currentSubProb;
        SubProblemsListIterator *subProblem_iterators = create_SubProblemList_iterator(&generatedSubProblems);
        for (size_t j = 0; j < generatedSubProblems.size; j++) {
            SubProblem *generatedSubProblem = SubProblemList_iterator_get_next(subProblem_iterators);
            if (generatedSubProblem->value > best_subproblem.value &&
                generatedSubProblem->value <= best_lower_bound) {
                best_subproblem = *generatedSubProblem;
            }
        }
        currentSubProb->type = best_subproblem.type;
        currentSubProb->value = best_subproblem.value; // metto valore massimo trovato per migliorare ordinamento
        delete_SubProblemList_iterator(subProblem_iterators);
        delete_SubProblemList(&generatedSubProblems);
        //printf("\nBest lower bound: %f, Best value: %f\n", best_lower_bound, best_subproblem.value);

        if(currentSubProb->type == OPEN) {
            double best_prob_branch = -1;
            double best_prob = -1;

            for (int i = 0; i < problem->graph.num_edges; i++) {

                Edge current_edge = problem->graph.edges[i];

                if (currentSubProb->constraints[current_edge.src][current_edge.dest] == NOTHING) {

                    if(best_subproblem.oneTree.edges_matrix[current_edge.src][current_edge.dest] != -1){
```

```c
                        if ((fabs(prob_branch[i] - 0.5) < fabs(best_prob_branch - 0.5)) ||
                            (HYBRID && (fabs(prob_branch[i] - 0.5) == fabs(best_prob_branch - 0.5))
                             && (current_edge.prob > best_prob))) {

                            best_prob_branch = prob_branch[i];
                            best_prob = current_edge.prob;
                            currentSubProb->edge_to_branch = i;
                        }

                    }
                }
            }
            //printf("\nBest edge to branch: %d, with score %f and prob %f; BEST VALUE: %f\n", currentSubProb->edge_to_branch, best_prob_branch, best_prob, problem->bestValue);
        }
    } else {
        currentSubProb->type = CLOSED_BOUND;
    }
}


bool time_limit_reached(void) {
    return ((clock() - problem->start) / CLOCKS_PER_SEC) > TIME_LIMIT_SECONDS;
}


void nearest_prob_neighbour(unsigned short start_node) {
    SubProblem nn_subProblem;
    nn_subProblem.num_forbidden_edges = 0;
    nn_subProblem.num_mandatory_edges = 0;
    nn_subProblem.num_edges_in_cycle = 0;
    nn_subProblem.timeToReach = ((float) (clock() - problem->start)) / CLOCKS_PER_SEC;
    create_mst(&nn_subProblem.oneTree, problem->graph.nodes, problem->graph.num_nodes);
    unsigned short current_node = start_node;
    bool visited[MAX_VERTEX_NUM] = {false};
    ConstrainedEdge cycleEdge;

    for (unsigned short visited_count = 0; visited_count < problem->graph.num_nodes; visited_count++) {

        if (visited_count == problem->graph.num_nodes - 1) {
            add_edge(&nn_subProblem.oneTree, &problem->graph.edges_matrix[current_node][start_node]);
            cycleEdge.src = current_node;
            cycleEdge.dest = start_node;
            nn_subProblem.cycleEdges[nn_subProblem.num_edges_in_cycle] = cycleEdge;
            nn_subProblem.num_edges_in_cycle++;
        } else {
            double best_edge_value = INFINITE;
            unsigned short best_neighbour = current_node;
            for (unsigned short i = 0; i < problem->graph.nodes[current_node].num_neighbours; i++) {

                if (problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].weight <
                    best_edge_value
                    && !visited[problem->graph.nodes[current_node].neighbours[i]]) {
                    best_edge_value = problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].weight;
                    best_neighbour = problem->graph.nodes[current_node].neighbours[i];
                }
            }
            add_edge(&nn_subProblem.oneTree, &problem->graph.edges_matrix[current_node][best_neighbour]);
            cycleEdge.src = current_node;
            cycleEdge.dest = best_neighbour;
            nn_subProblem.cycleEdges[nn_subProblem.num_edges_in_cycle] = cycleEdge;
            nn_subProblem.num_edges_in_cycle++;
            visited[current_node] = true;
            current_node = best_neighbour;
        }
    }
    nn_subProblem.value = nn_subProblem.oneTree.cost;
    nn_subProblem.oneTree.isValid = true;
    nn_subProblem.type = CLOSED_NN;
    nn_subProblem.prob = nn_subProblem.oneTree.prob;

    if (HYBRID) {
        SubProblem prob_nn_subProblem;
        prob_nn_subProblem.num_forbidden_edges = 0;
        prob_nn_subProblem.num_mandatory_edges = 0;
        prob_nn_subProblem.num_edges_in_cycle = 0;
        create_mst(&prob_nn_subProblem.oneTree, problem->graph.nodes, problem->graph.num_nodes);
        bool prob_visited[MAX_VERTEX_NUM] = {false};
        current_node = start_node;

        for (unsigned short visited_count = 0; visited_count < problem->graph.num_nodes; visited_count++) {

```

```
            if (visited_count == problem->graph.num_nodes - 1) {
                add_edge(&prob_nn_subProblem.oneTree, &problem->graph.edges_matrix[current_node][start_node]);
                cycleEdge.src = current_node;
                cycleEdge.dest = start_node;
                prob_nn_subProblem.cycleEdges[prob_nn_subProblem.num_edges_in_cycle] = cycleEdge;
                prob_nn_subProblem.num_edges_in_cycle++;
            } else {
                double best_edge_prob = -1;
                unsigned short best_neighbour = current_node;
                for (unsigned short i = 0; i < problem->graph.nodes[current_node].num_neighbours; i++) {

                    if (problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].prob >
                        best_edge_prob
                        && !prob_visited[problem->graph.nodes[current_node].neighbours[i]]) {
                        best_edge_prob = problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].prob;
                        best_neighbour = problem->graph.nodes[current_node].neighbours[i];
                    }
                }
                add_edge(&prob_nn_subProblem.oneTree, &problem->graph.edges_matrix[current_node][best_neighbour]);
                cycleEdge.src = current_node;
                cycleEdge.dest = best_neighbour;
                prob_nn_subProblem.cycleEdges[prob_nn_subProblem.num_edges_in_cycle] = cycleEdge;
                prob_nn_subProblem.num_edges_in_cycle++;
                prob_visited[current_node] = true;
                current_node = best_neighbour;
            }
        }
        prob_nn_subProblem.value = prob_nn_subProblem.oneTree.cost;
        prob_nn_subProblem.oneTree.isValid = true;
        prob_nn_subProblem.type = CLOSED_NN_HYBRID;
        prob_nn_subProblem.prob = prob_nn_subProblem.oneTree.prob;

        bool better_prob = prob_nn_subProblem.value < nn_subProblem.value;
        SubProblem *best = better_prob ? &prob_nn_subProblem : &nn_subProblem;

        if (best->value < problem->bestValue) {
            problem->bestValue = best->value;
            problem->bestSolution = *best;
        }

    } else {
        if (nn_subProblem.value < problem->bestValue) {
            problem->bestValue = nn_subProblem.value;
            problem->bestSolution = nn_subProblem;
        }
    }

}


// "a" is better than "b" if its LB is higher
bool compare_candidate_node(SubProblem * a, SubProblem * b){
    if (HYBRID) {
        return (a->value - b->value) > EPSILON ||
            ((a->value > b->value) && (a->prob - b->prob) >= BETTER_PROB);
    } else {
        return (a->value - b->value) >= EPSILON;
    }
}


unsigned short find_candidate_node(void) {

    unsigned short best_candidate = 0;
    SubProblem best_subProblem;
    best_subProblem.value = 0;
    best_subProblem.prob = 0;

    for (unsigned short i = 0; i < problem->graph.num_nodes; i++) {
        SubProblem currentCandidate;
        problem->candidateNodeId = i;
        currentCandidate.num_forbidden_edges = 0;
        currentCandidate.num_mandatory_edges = 0;

        nearest_prob_neighbour(i);

        if(PRIM){
            constrained_prim(&problem->graph, &currentCandidate, i);
        }
        else{
            constrained_kruskal(&problem->graph, &currentCandidate, i);
        }
```

```
        mst_to_one_tree(&currentCandidate, &problem->graph);
        currentCandidate.value = currentCandidate.oneTree.cost;
        currentCandidate.prob = currentCandidate.oneTree.prob;

        if (compare_candidate_node(&currentCandidate, &best_subProblem)) {
            best_candidate = i;
            best_subProblem.value = currentCandidate.value;
            best_subProblem.prob = currentCandidate.prob;
        }

    }
    return best_candidate;
}


bool check_feasibility(Graph *graph) {

    bool feasible = true;
    for (short i = 0; feasible && i < graph->num_nodes; i++) {
        Node current_node = graph->nodes[i];
        if (current_node.num_neighbours < 2) {
            feasible = false;
            printf("\nThe graph is not feasible for the BB algorithm. Node %i has less than 2
neighbors.",
                   current_node.positionInGraph);
        }
    }
    return feasible;
}


void branch_and_bound(Problem *current_problem) {

    problem = current_problem;

    if (check_feasibility(&problem->graph)) {

        problem->start = clock();
        problem->bestValue = INFINITE;
        problem->candidateNodeId = find_candidate_node();
        problem->bestSolution.id = 0;
        problem->bestSolution.treeLevel = 0;
        problem->bestSolution.fatherId = -1;
        problem->exploredBBNodes = 0;
        problem->generatedBBNodes = 0;
        problem->totTreeLevels = 1;
        problem->interrupted = false;
        problem->reformulationGraph = problem->graph;
        problem->generatedBBNodes++;
        problem->num_fixed_edges = 0;

        SubProblem subProblem;
        subProblem.treeLevel = 1;
        subProblem.id = problem->generatedBBNodes;
        subProblem.fatherId = 0;
        subProblem.type = OPEN;
        subProblem.prob = 0;
        subProblem.value = INFINITE;
        subProblem.num_edges_in_cycle = 0;
        subProblem.edge_to_branch = -1;
        initialize_matrix(&subProblem);


        SubProblemsList subProblems;
        new_SubProblemList(&subProblems);
        add_elem_SubProblemList_bottom(&subProblems, &subProblem);

        while (subProblems.size != 0 && !time_limit_reached()) {
            SubProblem current_sub_problem = *get_SubProblemList_elem_index(&subProblems, 0);
            delete_SubProblemList_elem_index(&subProblems, 0);
            bound(&current_sub_problem);
            if (current_sub_problem.type == OPEN && current_sub_problem.edge_to_branch != -1) {
                problem->num_fixed_edges += variable_fixing(&current_sub_problem);
                branch(&subProblems, &current_sub_problem);
            }
        }

        if (time_limit_reached()) {
            problem->interrupted = true;
        }

        problem->end = clock();
        delete_SubProblemList(&subProblems);
    }
}
```

```c
void set_problem(Problem *current_problem) {
    problem = current_problem;
}


void print_subProblem(const SubProblem *subProblem) {

    char *type;
    if (subProblem->type == OPEN) {
        type = "OPEN";
    } else if (subProblem->type == CLOSED_UNFEASIBLE) {
        type = "CLOSED_UNFEASIBLE";
    } else if (subProblem->type == CLOSED_BOUND) {
        type = "CLOSED_BOUND";
    } else if (subProblem->type == CLOSED_NN) {
        type = "CLOSED_NEAREST_NEIGHBOR";
    } else if (subProblem->type == CLOSED_NN_HYBRID) {
        type = "CLOSED_NEAREST_NEIGHBOR_HYBRID";
    } else {
        type = "CLOSED_SUBGRADIENT";
    }
    printf("\nSUBPROBLEM with cost = %lf, type = %s, level of the BB tree = %i, prob_tour = %lf, BBNode number = %u and time to obtain = %lfs, edge to branch = %i\n",
           subProblem->value, type, subProblem->treeLevel, subProblem->prob, subProblem->id,
           subProblem->timeToReach, subProblem->edge_to_branch);

    //print_mst_original_weight(&subProblem->oneTree, &problem->graph);

    printf("\nCycle with %i edges:", subProblem->num_edges_in_cycle);
    unsigned short last_dest = 0;
    for (unsigned short i = 0; i < subProblem->num_edges_in_cycle; i++) {
        ConstrainedEdge edge_cycle = subProblem->cycleEdges[i];
        unsigned short src = edge_cycle.src;
        unsigned short dest = edge_cycle.dest;

        if (i == 0){
            if (src == subProblem->cycleEdges[subProblem->num_edges_in_cycle - 1].dest ||
                src == subProblem->cycleEdges[subProblem->num_edges_in_cycle - 1].src){
                printf(" %i <-> %i,", src, dest);
                last_dest = dest;
            }
            else{
                printf(" %i <-> %i,", dest, src);
                last_dest = src;
            }
        } else{
            if(src == last_dest){
                if (i == subProblem->num_edges_in_cycle - 1) {
                    printf("  %i <-> %i", src, dest);
                } else {
                    printf("  %i <-> %i,", src, dest);
                }

                last_dest = dest;
            }
            else{
                if (i == subProblem->num_edges_in_cycle - 1) {
                    printf("  %i <-> %i", dest, src);
                } else {
                    printf("  %i <-> %i,", dest, src);
                }
                last_dest = src;
            }
        }
    }

    printf("\n%i Mandatory edges:", subProblem->num_mandatory_edges);
    printf("\n%i Forbidden edges:", subProblem->num_forbidden_edges);
    printf("\n");
}

void print_problem(void) {
    printf("Optimal tour found with candidate node = %i, elapsed time = %lfs and interrupted = %s\n",
           problem->candidateNodeId,  ((double) (problem->end - problem->start)) / CLOCKS_PER_SEC,
           problem->interrupted ? "TRUE" : "FALSE");

    printf("\nMST solved with: %s algorithm\n", PRIM ? "Prim" : "Kruskal");

    printf("\nGHOSH_UB = %lf, SUBPROBLEM_UB = %lf\n", GHOSH_UB, EPSILON);

    printf("\nB-&-B tree with generated BBNodes = %u,  explored BBNodes = %u and max tree level = %u\n",
           problem->generatedBBNodes, problem->exploredBBNodes, problem->totTreeLevels);

    printf("\nNumber of fixed edges = %i\n", problem->num_fixed_edges);

    print_subProblem(&problem->bestSolution);
```

```
01153 }
```

9.3 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/branch_and_bound.h File Reference

The declaration of all the methods used by the Branch and Bound algorithm.

```
#include "kruskal.h"
#include "prim.h"
#include "../data_structures/b_and_b_data.h"
```

Functions

- void dfs (SubProblem ∗subProblem)
  A Depth First Search algorithm on a Graph.
- bool check_hamiltonian (SubProblem ∗subProblem)
  Function that checks if the 1Tree of a SubProblem is a tour.
- BBNodeType mst_to_one_tree (SubProblem ∗currentSubproblem, Graph ∗graph)
  Transforms a MST into a 1Tree.
- void clean_matrix (SubProblem ∗subProblem)
  Clean the matrix of constraints of a SubProblem.
- void copy_constraints (SubProblem ∗subProblem, const SubProblem ∗otherSubProblem)
  Copy the matrix of constraints of a SubProblem into another.
- bool compare_subproblems (const SubProblem ∗a, const SubProblem ∗b)
  Compare two OPEN SubProblems.
- void branch (SubProblemsList ∗openSubProblems, SubProblem ∗subProblem)
  The Shutler's branching rule.
- int variable_fixing (SubProblem ∗subProblem)
  The function used to fix the edge variables to be mandatory or forbidden.
- bool infer_constraints (SubProblem ∗subProblem)
  Infer the values of some edge variables of a SubProblem.
- void bound (SubProblem ∗currentSubProb)
  The Held-Karp bound function with the subgradient algorithm.
- bool time_limit_reached (void)
  Check if the time limit has been reached.
- void nearest_prob_neighbour (unsigned short start_node)
  This function is used to find the first feasible tour.
- unsigned short find_candidate_node (void)
  Select the candidate Node, i.e. the starting vertex of the tour.
- void branch_and_bound (Problem ∗current_problem)
  The Branch and Bound algorithm.
- bool check_feasibility (Graph ∗graph)
  Check if the Graph associated to the Problem is feasible.
- void set_problem (Problem ∗current_problem)
  Define the problem to solve.
- void print_subProblem (const SubProblem ∗subProblem)
  Get all metrics of a certain SubProblem.
- void print_problem (void)
  Get all metrics of the problem.

Variables

- static Problem * problem

    The pointer to the problem to solve.

9.3.1 Detailed Description

The declaration of all the methods used by the Branch and Bound algorithm.

Author

Lorenzo Sciandra

This file contains all the methods used by the Hybrid and Classic Branch and Bound solver.

Version

1.0.0 @data 2024-05-1

Copyright

Repo: `https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound`

Definition in file branch_and_bound.h.

9.3.2 Function Documentation

9.3.2.1 bound()

```
void bound (
            SubProblem * currentSubProb )
```

The Held-Karp bound function with the subgradient algorithm.

This function has a primal and dual behaviour: after the minimal 1Tree is found, a subgradient algorithm is used to do a dual ascent of the Lagrangian relaxation. More details at `https://www.sciencedirect.↩com/science/article/abs/pii/S0377221796002147?via%3Dihub`.

Parameters

current_problem	The pointer to the SubProblem or branch-and-bound Node in the tree.

Definition at line 657 of file branch_and_bound.c.

9.3.2.2 branch()

```
void branch (
            SubProblemsList * openSubProblems,
            SubProblem * subProblem )
```

The Shutler's branching rule.

Every SubProblem is branched into 2 new SubProblems, one including the "edge_to_branch" and the other not. More details at `http://www.jstor.org/stable/254144`.

Parameters

openSubProblems	The list of open SubProblems, to which the new SubProblems will be added.
subProblem	The SubProblem to branch.

Definition at line 256 of file branch_and_bound.c.

9.3.2.3 branch_and_bound()

```
void branch_and_bound (
            Problem * current_problem )
```

The Branch and Bound algorithm.

This is the main function of the Branch and Bound algorithm. It stores all the open SubProblems in a SubProblemsList and analyzes them one by one with the branch() and held_karp_bound() functions.

Parameters

current_problem	The pointer to the problem to solve.

Definition at line 1012 of file branch_and_bound.c.

9.3.2.4 check_feasibility()

```
bool check_feasibility (
            Graph * graph )
```

Check if the Graph associated to the Problem is feasible.

A Graph is feasible if every Node has at least degree 2.

Parameters

graph	The Graph to check.

Returns

true if the Graph is feasible, false otherwise.

Definition at line 997 of file branch_and_bound.c.

9.3.2.5 check_hamiltonian()

```
bool check_hamiltonian (
            SubProblem * subProblem )
```

Function that checks if the 1Tree of a SubProblem is a tour.

This is done by simply check if all the edges are in the cycle passing through the candidate Node.

Parameters

subProblem	The SubProblem to check.

Returns

true if the SubProblem is a Hamiltonian cycle, false otherwise.

Definition at line 82 of file branch_and_bound.c.

9.3.2.6 clean_matrix()

```
void clean_matrix (
            SubProblem * subProblem )
```

Clean the matrix of constraints of a SubProblem.

This function is used to initialize the matrix of ConstraintType for a SubProblem.

Parameters

subProblem	The SubProblem with no ConstraintType.

9.3.2.7 compare_subproblems()

```
bool compare_subproblems (
            const SubProblem * a,
            const SubProblem * b )
```

Compare two OPEN SubProblems.

This function is used to sort the SubProblems in the open list to define its order.

Parameters

a	The first SubProblem to compare.
b	The second SubProblem to compare.

Returns

true if the first SubProblem is better than the second, false otherwise.

Definition at line 647 of file branch_and_bound.c.

9.3.2.8 copy_constraints()

```
void copy_constraints (
            SubProblem * subProblem,
            const SubProblem * otherSubProblem )
```

Copy the matrix of constraints of a SubProblem into another.

This function is used when a SubProblem is branched into two new SubProblems, and the constraints of the father SubProblem are copied into the sons.

Parameters

subProblem	The SubProblem to which the ConstraintType will be copied.
otherSubProblem	The SubProblem from which the ConstraintType will be copied.

Definition at line 234 of file branch_and_bound.c.

9.3.2.9 dfs()

```
void dfs (
            SubProblem * subProblem )
```

A Depth First Search algorithm on a Graph.

This function is used to find the cycle in the 1Tree SubProblem, passing through the candidate Node.

Parameters

subProblem	The SubProblem to inspect.

Definition at line 18 of file branch_and_bound.c.

9.3.2.10 find_candidate_node()

```
unsigned short find_candidate_node (
            void )
```

Select the candidate Node, i.e. the starting vertex of the tour.

Every Node is tried and the one with the best lower bound is chosen. In the Hybrid mode, when two nodes have the same lower bound, the one with the best probability is chosen.

Returns

the candidate Node id.

Definition at line 961 of file branch_and_bound.c.

9.3.2.11 infer_constraints()

```
bool infer_constraints (
            SubProblem * subProblem )
```

Infer the values of some edge variables of a SubProblem.

According to the constraints of the father SubProblem and the one added to the son, we can infer new variables values in order to check if the SubProblem is still feasible or not.

Parameters

subProblem	The SubProblem to which we want to infer the variables values.

Returns

true if the subproblem remains feasible, false otherwise.

Definition at line 185 of file branch_and_bound.c.

9.3.2.12 mst_to_one_tree()

```
BBNodeType mst_to_one_tree (
            SubProblem * currentSubproblem,
            Graph * graph )
```

Transforms a MST into a 1Tree.

This is done by adding the two least-cost edges incident to the candidate Node in the MST.

Parameters

currentSubproblem	The SubProblem to which the MST belongs.
graph	The Graph of the Problem.

Returns

an enum value that indicates if the SubProblem is feasible or not.

Definition at line 88 of file branch_and_bound.c.

9.3.2.13 nearest_prob_neighbour()

```
void nearest_prob_neighbour (
            unsigned short start_node )
```

This function is used to find the first feasible tour.

If the Hybrid mode is disabled, it is the simple nearest neighbour algorithm. Otherwise, it also implements the Probabilistic Nearest Neighbour algorithm where, starting from a Node, the Edge with the best probability is chosen. This method is repeated by choosing every Node as the starting Node. The best tour found is stored as the best tour found so far.

Parameters

start_node	The Node from which the tour will start.

Definition at line 844 of file branch_and_bound.c.

9.3.2.14 print_problem()

```
void print_problem (
            void )
```

Get all metrics of the problem.

It is used at the end of the algorithm to print the solution obtained. It calls the print_subProblem() function on the best SubProblem found.

Definition at line 1138 of file branch_and_bound.c.

9.3.2.15 print_subProblem()

```
void print_subProblem (
            const SubProblem * subProblem )
```

Get all metrics of a certain SubProblem.

It is used at the end of the algorithm to print the solution obtained.

Parameters

subProblem	The SubProblem to print.

Definition at line 1073 of file branch_and_bound.c.

9.3.2.16 set_problem()

```
void set_problem (
            Problem * current_problem )
```

Define the problem to solve.

This function is used to set the pointer to the problem to solve.

Parameters

current_problem	The pointer to the problem to solve.

Definition at line 1068 of file branch_and_bound.c.

9.3.2.17 time_limit_reached()

```
bool time_limit_reached (
            void )
```

Check if the time limit has been reached.

This function is used to check if the time limit has been reached.

Returns

true if the time limit has been reached, false otherwise.

Definition at line 839 of file branch_and_bound.c.

9.3.2.18 variable_fixing()

```
int variable_fixing (
            SubProblem * subProblem )
```

The function used to fix the edge variables to be mandatory or forbidden.

By calculating the calculating of marginal and replacement costs, the edge variables are fixed to be forbidden or mandatory. More details at https://link.springer.com/chapter/10.1007/978-3-642-13520-0↩_6.

Parameters

subProblem	The SubProblem that we want to add the constraints to.

Returns

the num of variables fixed.

Definition at line 421 of file branch_and_bound.c.

9.3.3 Variable Documentation

9.3.3.1 problem

```
Problem* problem [static]
```

The pointer to the problem to solve.

Definition at line 22 of file branch_and_bound.h.

9.4 branch_and_bound.h

Go to the documentation of this file.

```
#ifndef BRANCHANDBOUND1TREE_BRANCH_AND_BOUND_H
#define BRANCHANDBOUND1TREE_BRANCH_AND_BOUND_H
#include "kruskal.h"
#include "prim.h"
#include "../data_structures/b_and_b_data.h"


static Problem * problem;


void dfs(SubProblem *subProblem);


bool check_hamiltonian(SubProblem *subProblem);


BBNodeType mst_to_one_tree(SubProblem *currentSubproblem, Graph *graph);


void clean_matrix(SubProblem *subProblem);


void copy_constraints(SubProblem *subProblem, const SubProblem *otherSubProblem);


bool compare_subproblems(const SubProblem *a, const SubProblem *b);


void branch(SubProblemsList *openSubProblems, SubProblem *subProblem);


int variable_fixing (SubProblem * subProblem);


bool infer_constraints(SubProblem * subProblem);


void bound(SubProblem *currentSubProb);


bool time_limit_reached(void);


void nearest_prob_neighbour(unsigned short start_node);


unsigned short find_candidate_node(void);


void branch_and_bound(Problem * current_problem);


bool check_feasibility(Graph * graph);


void set_problem(Problem * current_problem);


void print_subProblem(const SubProblem *subProblem);


void print_problem(void);

```

```
00187
00188 #endif //BRANCHANDBOUND1TREE_BRANCH_AND_BOUND_H
```

9.5 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/kruskal.c File Reference

The implementaion of the functions needed to compute the MST with Kruskal's algorithm.

```
#include "kruskal.h"
```

Functions

- static void swap (Graph ∗graph, unsigned short swap_1, unsigned short swap_2)
- static int pivot_quicksort (Graph ∗graph, unsigned short first, unsigned short last)
- static void quick_sort (Graph ∗graph, unsigned short first, unsigned short last)
- void wrap_quick_sort (Graph ∗graph)

  The wrapper of the quick sort algorithm.
- void kruskal (Graph ∗graph, MST ∗mst)

  The Kruskal algorithm to find the Minimum Spanning Tree O(|E| log |V|)

9.5.1 Detailed Description

The implementaion of the functions needed to compute the MST with Kruskal's algorithm.

Author

Lorenzo Sciandra

This file contains the implementation of the Kruskal algorithm to find the Minimum Spanning Tree.

Version

1.0.0 @data 2024-05-1

Copyright

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file kruskal.c.

9.5.2 Function Documentation

9.5.2.1 kruskal()

```
void kruskal (
            Graph * graph,
            MST * mst )
```

The Kruskal algorithm to find the Minimum Spanning Tree O(|E| log |V|)

This is the classic Kruskal algorithm that uses Merge-Find Sets.

Parameters

graph	The Graph from which we want to find the MST.
mst	The Minimum Spanning Tree.

Definition at line 131 of file kruskal.c.

9.5.2.2 pivot_quicksort()

```
static int pivot_quicksort (
            Graph * graph,
            unsigned short first,
            unsigned short last )  [static]
```

Definition at line 39 of file kruskal.c.

9.5.2.3 quick_sort()

```
static void quick_sort (
            Graph * graph,
            unsigned short first,
            unsigned short last )  [static]
```

Definition at line 110 of file kruskal.c.

9.5.2.4 swap()

```
static void swap (
            Graph * graph,
            unsigned short swap_1,
            unsigned short swap_2 )  [static]
```

Definition at line 18 of file kruskal.c.

9.5.2.5 wrap_quick_sort()

```
void wrap_quick_sort (
            Graph * graph )
```

The wrapper of the quick sort algorithm.

If the Graph is not sorted, this function calls the quick sort algorithm to sort the edges of the Graph.

Parameters

graph	The Graph to which we want to sort the edges.

Definition at line 123 of file kruskal.c.

9.6 kruskal.c

Go to the documentation of this file.

```
#include "kruskal.h"


static void swap(Graph *graph, unsigned short swap_1, unsigned short swap_2) {

    Edge * edges = graph->edges;

    //printf("\nswap values %lf - %lf, at %d - %d\n", edges[swap_1].weight, edges[swap_2].weight, swap_1, swap_2);

    graph->edges_matrix[edges[swap_1].src][edges[swap_1].dest].positionInGraph = swap_2;
    graph->edges_matrix[edges[swap_2].src][edges[swap_2].dest].positionInGraph = swap_1;


    graph->edges_matrix[edges[swap_1].dest][edges[swap_1].src].positionInGraph = swap_2;
    graph->edges_matrix[edges[swap_2].dest][edges[swap_2].src].positionInGraph = swap_1;

    edges[swap_1].positionInGraph = swap_2;
    edges[swap_2].positionInGraph = swap_1;
    Edge temp = edges[swap_1];
    edges[swap_1] = edges[swap_2];
    edges[swap_2] = temp;
}


static int pivot_quicksort(Graph * graph, unsigned short first, unsigned short last) {
    Edge * edges = graph->edges;
    Edge last_edge = edges[last];
    Edge first_edge = edges[first];
    unsigned short middle = (first + last) / 2;
    Edge middle_edge = edges[middle];
    double pivot_weight = first_edge.weight;
    double pivot_prob = first_edge.prob;

    if (last_edge.weight > first_edge.weight ||
            (HYBRID && last_edge.weight == first_edge.weight && last_edge.prob < first_edge.prob)){
        if ((last_edge.weight > middle_edge.weight) ||
                (HYBRID && last_edge.weight == middle_edge.weight && last_edge.prob < middle_edge.prob)) {
            if ((middle_edge.weight > first_edge.weight) ||
                    (HYBRID && middle_edge.weight == first_edge.weight && middle_edge.prob < first_edge.prob)){
                pivot_weight = middle_edge.weight;
                pivot_prob = middle_edge.prob;
                swap(graph, first, middle);
            }
        } else {
            pivot_weight = last_edge.weight;
            pivot_prob = last_edge.prob;
            swap(graph, first, last);
        }
    } else {
        if (last_edge.weight > middle_edge.weight ||
                (HYBRID && last_edge.weight == middle_edge.weight && last_edge.prob < middle_edge.prob)) {
            pivot_weight = last_edge.weight;
            pivot_prob = last_edge.prob;
            swap(graph, first, last);

        } else if ((first_edge.weight > middle_edge.weight) ||
                (HYBRID && first_edge.weight == middle_edge.weight && first_edge.prob < middle_edge.prob)) {
            pivot_weight = middle_edge.weight;
            pivot_prob = middle_edge.prob;
            swap(graph, first, middle);
        }
    }
```

```
    unsigned short j = last;
    unsigned short i = first + 1;
    bool condition = true;
    while (condition) {
        Edge i_edge = edges[i];
        while (i <= j && ((!HYBRID && pivot_weight >= i_edge.weight) ||
                          (HYBRID && (pivot_weight > i_edge.weight || (pivot_weight == i_edge.weight
&& pivot_prob <= i_edge.prob))))){
            i += 1;
            i_edge = edges[i];
        }
        Edge j_edge = edges[j];
        while (i <= j && (j_edge.weight > pivot_weight
                          || (HYBRID && (j_edge.weight == pivot_weight && j_edge.prob < pivot_prob))))
{
            j -= 1;
            j_edge = edges[j];
        }
        if (i <= j) {
            swap(graph, i,  j);
        } else {
            condition = false;
        }

    }

    if(j != first){
        swap( graph,  first,  j);
    }

    return j;

}


static void quick_sort(Graph * graph, unsigned short first, unsigned short last) {
    if (first < last) {
        unsigned short pivot = pivot_quicksort(graph, first, last);
        if(pivot -1 > first) {
            quick_sort(graph, first, pivot - 1);
        }
        if(pivot + 1 < last) {
            quick_sort(graph, pivot + 1, last);
        }
    }
}


void wrap_quick_sort(Graph * graph) {
    if (!graph->orderedEdges) {
        graph->orderedEdges = true;
        quick_sort(graph, 0, graph->num_edges - 1);
    }
}


void kruskal(Graph * graph, MST * mst) {
    create_mst(mst, graph->nodes, graph->num_nodes);
    Forest forest;
    create_forest(&forest, graph->nodes, graph->num_nodes);
    wrap_quick_sort(graph);
    unsigned short num_edges_inG = 0;
    unsigned short num_edges_inMST = 0;

    while (num_edges_inG < graph->num_edges && num_edges_inMST < graph->num_nodes - 1) {
        // get the edge with the minimum weight
        Edge current_edge = graph->edges[num_edges_inG];
        unsigned short src = current_edge.src;
        unsigned short dest = current_edge.dest;

        Set *set1_root = find(&forest.sets[src]);
        Set *set2_root = find(&forest.sets[dest]);

        if (set1_root->num_in_forest != set2_root->num_in_forest) {
            merge(set1_root, set2_root);
            // add the edge to the MST
            add_edge(mst, &current_edge);
            num_edges_inMST++;
        }
        num_edges_inG++;
    }
    if (num_edges_inMST == graph->num_nodes - 1) {
        mst->isValid = true;
    }
}
```

9.7 GraphConvolutionalBranchandBound/src/Hybrid↩ Solver/main/algorithms/kruskal.h File Reference

The declaration of the functions needed to compute the MST with Kruskal's algorithm.

```
#include "../data_structures/mst.h"
```

Functions

- static void swap (Graph ∗graph, unsigned short swap_1, unsigned short swap_2)
  Swaps two edges in the list of edges in the Graph.
- static int pivot_quicksort (Graph ∗graph, unsigned short first, unsigned short last)
  The core of the quick sort algorithm.
- static void quick_sort (Graph ∗graph, unsigned short first, unsigned short last)
  The quick sort algorithm O(n log n).
- void wrap_quick_sort (Graph ∗graph)
  The wrapper of the quick sort algorithm.
- void kruskal (Graph ∗graph, MST ∗mst)
  The Kruskal algorithm to find the Minimum Spanning Tree O(|E| log |V|)

9.7.1 Detailed Description

The declaration of the functions needed to compute the MST with Kruskal's algorithm.

Author

Lorenzo Sciandra

This file contains the declaration of the Kruskal algorithm to find the Minimum Spanning Tree.

Version

1.0.0 @data 2024-05-1

Copyright

Copyright (c) 2024, license MIT

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file kruskal.h.

9.7.2 Function Documentation

9.7.2.1 kruskal()

```
void kruskal (
            Graph * graph,
            MST * mst )
```

The Kruskal algorithm to find the Minimum Spanning Tree O(|E| log |V|)

This is the classic Kruskal algorithm that uses Merge-Find Sets.

Parameters

graph	The Graph from which we want to find the MST.
mst	The Minimum Spanning Tree.

Definition at line 131 of file kruskal.c.

9.7.2.2 pivot_quicksort()

```
static int pivot_quicksort (
            Graph * graph,
            unsigned short first,
            unsigned short last ) [static]
```

The core of the quick sort algorithm.

This function find the pivot position to recursively call the quick sort algorithm. While doing this all the edges with weight less than the pivot are moved to the left of the pivot and to the right otherwise.

Parameters

graph	The Graph to which we want to sort the edges.
first	The index of the first Edge to consider in the list of edges.
last	The index of the last Edge to consider in the list of edges.

Returns

the index of the pivot.

9.7.2.3 quick_sort()

```
static void quick_sort (
            Graph * graph,
            unsigned short first,
            unsigned short last ) [static]
```

The quick sort algorithm O(n log n).

It is used to sort the edges of the Graph in ascending order in O(n log n). It is recursive.

Parameters

graph	The Graph to which we want to sort the edges.
first	The index of the first Edge to consider in the list of edges.
last	The index of the last Edge to consider in the list of edges.

9.7.2.4 swap()

```
static void swap (
            Graph * graph,
            unsigned short swap_1,
            unsigned short swap_2 )  [static]
```

Swaps two edges in the list of edges in the Graph.

This function is used to swap two edges in the list of edges in the Graph.

Parameters

graph	The Graph to which the edges belong.
swap_1	The index of the first Edge to swap.
swap_2	The index of the second Edge to swap.

9.7.2.5 wrap_quick_sort()

```
void wrap_quick_sort (
            Graph * graph )
```

The wrapper of the quick sort algorithm.

If the Graph is not sorted, this function calls the quick sort algorithm to sort the edges of the Graph.

Parameters

graph	The Graph to which we want to sort the edges.

Definition at line 123 of file kruskal.c.

9.8 kruskal.h

Go to the documentation of this file.

```
#ifndef BRANCHANDBOUND1TREE_KRUSKAL_H
#define BRANCHANDBOUND1TREE_KRUSKAL_H
#include "../data_structures/mst.h"



static void swap(Graph * graph, unsigned short swap_1, unsigned short swap_2);



```

```
static int pivot_quicksort(Graph * graph, unsigned short first, unsigned short last);
static void quick_sort(Graph * graph, unsigned short first, unsigned short last);
void wrap_quick_sort(Graph * graph);
void kruskal(Graph * graph, MST * mst);
#endif //BRANCHANDBOUND1TREE_KRUSKAL_H
```

9.9 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/prim.c File Reference

The implementaion of the functions needed to compute the MST with Prim's algorithm.

```
#include "prim.h"
```

Functions

- void prim (const Graph ∗graph, MST ∗mst)

    The Prim algorithm to find the Minimum Spanning Tree O(|E| + |V| log |V|)

9.9.1 Detailed Description

The implementaion of the functions needed to compute the MST with Prim's algorithm.

Author

Lorenzo Sciandra

This file contains the implementation of the Prim algorithm to find the Minimum Spanning Tree.

Version

1.0.0 @data 2024-05-1

Copyright

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file prim.c.

9.9.2 Function Documentation

9.9.2.1 prim()

```
void prim (
            const Graph * graph,
            MST * mst )
```

The Prim algorithm to find the Minimum Spanning Tree O(|E| + |V| log |V|)

This is the implementation of the Prim algorithm with Fibonacci Heap to find the Minimum Spanning Tree. When the graph is large and complete, it is way faster than Kruskal's algorithm.

Parameters

graph	The Graph from which we want to find the MST.
mst	The Minimum Spanning Tree.

Definition at line 16 of file prim.c.

9.10 prim.c

Go to the documentation of this file.

```
00001
00014 #include "prim.h"
00015
00016 void prim(const Graph * graph, MST * mst){
00017     create_mst(mst, graph->nodes, graph->num_nodes);
00018     FibonacciHeap heap;
00019     create_fibonacci_heap(&heap);
00020
00021     OrdTreeNode tree_nodes[graph->num_nodes];
00022     bool in_heap [graph->num_nodes];
00023     int fathers [graph->num_nodes];
00024     bool start = true;
00025
00026     for (unsigned short i = 0; i < graph->num_nodes; i++) {
00027         if(start){
00028             create_insert_node(&heap, &tree_nodes[i], i, 0);
00029             start = false;
00030         } else{
00031             create_insert_node(&heap, &tree_nodes[i], i, DBL_MAX);
00032         }
00033         in_heap[i] = true;
00034         fathers[i] = -1;
00035     }
00036
00037     while(heap.num_nodes != 0){
00038         int min_pos = extract_min(&heap);
00039         if(min_pos == -1){
00040             fprintf(stderr, "Error: min_pos == -1\n");
00041             exit(1);
00042         }
00043         else{
00044             in_heap[min_pos] = false;
00045             for(unsigned short i = 0; i < graph->nodes[min_pos].num_neighbours; i++){
00046                 unsigned short neigh = graph->nodes[min_pos].neighbours[i];
00047                 if(neigh != min_pos && in_heap[neigh]) {
00048                     double weight = graph->edges_matrix[min_pos][neigh].weight;
00049                     if (weight < tree_nodes[neigh].value) {
00050                         fathers[neigh] = min_pos;
00051                         decrease_value(&heap, &tree_nodes[neigh], weight);
00052                     }
```

```
                }
            }
        }
    }

    for(unsigned short i = 0; i < graph->num_nodes; i++){
        if(fathers[i] != -1){
            add_edge(mst, &graph->edges_matrix[i][fathers[i]]);
        }
    }

    if(mst->num_edges == graph->num_nodes - 1){
        mst->isValid = true;
    }
    else{
        mst->isValid = false;
    }
}
```

9.11 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/prim.h File Reference

The declaration of the functions needed to compute the MST with Prim's algorithm.

```
#include "../data_structures/mst.h"
```

Functions

- void prim (const Graph *graph, MST *mst)

  The Prim algorithm to find the Minimum Spanning Tree O(|E| + |V| log |V|)

9.11.1 Detailed Description

The declaration of the functions needed to compute the MST with Prim's algorithm.

Author

Lorenzo Sciandra

This file contains the declaration of the Prim algorithm to find the Minimum Spanning Tree.

Version

1.0.0 @data 2024-05-1

Copyright

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file prim.h.

9.11.2 Function Documentation

9.11.2.1 prim()

```
void prim (
            const Graph * graph,
            MST * mst )
```

The Prim algorithm to find the Minimum Spanning Tree O(|E| + |V| log |V|)

This is the implementation of the Prim algorithm with Fibonacci Heap to find the Minimum Spanning Tree. When the graph is large and complete, it is way faster than Kruskal's algorithm.

Parameters

graph	The Graph from which we want to find the MST.
mst	The Minimum Spanning Tree.

Definition at line 16 of file prim.c.

9.12 prim.h

Go to the documentation of this file.

```
00001
00014 #ifndef BRANCHANDBOUND1TREE_PRIM_H
00015 #define BRANCHANDBOUND1TREE_PRIM_H
00016 #include "../data_structures/mst.h"
00017
00018
00020
00026 void prim(const Graph * graph, MST * mst);
00027
00028 #endif //BRANCHANDBOUND1TREE_PRIM_H
```

9.13 GraphConvolutionalBranchandBound/src/HybridSolver/main/data↩_structures/b_and_b_data.c File Reference

All the functions needed to manage the list of open subproblems.

```
#include "b_and_b_data.h"
```

Functions

- void new_SubProblemList (SubProblemsList *list)

  Create a new SubProblem List.
- void delete_SubProblemList (SubProblemsList *list)

  Delete a SubProblem List.

- bool is_SubProblemList_empty (SubProblemsList *list)

    Check if a SubProblem List is empty.
- SubProblemElem * build_list_elem (SubProblem *value, SubProblemElem *next, SubProblemElem *prev)
- size_t get_SubProblemList_size (SubProblemsList *list)

    Get the size of a SubProblem List.
- void add_elem_SubProblemList_bottom (SubProblemsList *list, SubProblem *element)

    Add a SubProblem to the bottom of a SubProblem List.
- void add_elem_SubProblemList_index (SubProblemsList *list, SubProblem *element, size_t index)

    Add a SubProblem at a specific index of a SubProblem List.
- void delete_SubProblemList_elem_index (SubProblemsList *list, size_t index)

    Remove a SubProblem from a specific index of a SubProblem List.
- SubProblem * get_SubProblemList_elem_index (SubProblemsList *list, size_t index)

    Get a SubProblem from a specific index of a SubProblem List.
- SubProblemsListIterator * create_SubProblemList_iterator (SubProblemsList *list)

    Create a new SubProblem List iterator on a SubProblem List.
- bool is_SubProblemList_iterator_valid (SubProblemsListIterator *iterator)

    Check if a SubProblem List iterator is valid.
- SubProblem * get_current_openSubProblemList_iterator_element (SubProblemsListIterator *iterator)
- void list_openSubProblemList_next (SubProblemsListIterator *iterator)
- SubProblem * SubProblemList_iterator_get_next (SubProblemsListIterator *iterator)

    Get the next element of a SubProblem List iterator.
- void delete_SubProblemList_iterator (SubProblemsListIterator *iterator)

    Delete a SubProblem List iterator.

9.13.1 Detailed Description

All the functions needed to manage the list of open subproblems.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Repo: `https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound`

Definition in file b_and_b_data.c.

9.13.2 Function Documentation

9.13.2.1 add_elem_SubProblemList_bottom()

```
void add_elem_SubProblemList_bottom (
            SubProblemsList * list,
            SubProblem * element )
```

Add a SubProblem to the bottom of a SubProblem List.

Parameters

list	The SubProblem List to modify.
element	The SubProblem to add.

Definition at line 59 of file b_and_b_data.c.

9.13.2.2 add_elem_SubProblemList_index()

```
void add_elem_SubProblemList_index (
              SubProblemsList * list,
              SubProblem * element,
              size_t index )
```

Add a SubProblem at a specific index of a SubProblem List.

Parameters

list	The SubProblem List to modify.
element	The SubProblem to add.
index	The index where to add the SubProblem.

list is clearer way but it is already checked inside get_list_size

Definition at line 75 of file b_and_b_data.c.

9.13.2.3 build_list_elem()

```
SubProblemElem * build_list_elem (
              SubProblem * value,
              SubProblemElem * next,
              SubProblemElem * prev )
```

Definition at line 44 of file b_and_b_data.c.

9.13.2.4 create_SubProblemList_iterator()

```
SubProblemsListIterator * create_SubProblemList_iterator (
              SubProblemsList * list )
```

Create a new SubProblem List iterator on a SubProblem List.

Parameters

list	The SubProblem List to iterate.

Returns

the SubProblem List iterator.

Definition at line 159 of file b_and_b_data.c.

9.13.2.5 delete_SubProblemList()

```
void delete_SubProblemList (
            SubProblemsList * list )
```

Delete a SubProblem List.

Parameters

list	The SubProblem List to delete.

Definition at line 23 of file b_and_b_data.c.

9.13.2.6 delete_SubProblemList_elem_index()

```
void delete_SubProblemList_elem_index (
            SubProblemsList * list,
            size_t index )
```

Remove a SubProblem from a specific index of a SubProblem List.

Parameters

list	The SubProblem List to modify.
index	The index of the SubProblem to remove.

Definition at line 113 of file b_and_b_data.c.

9.13.2.7 delete_SubProblemList_iterator()

```
void delete_SubProblemList_iterator (
            SubProblemsListIterator * iterator )
```

Delete a SubProblem List iterator.

Parameters

iterator	The SubProblem List iterator.

Definition at line 198 of file b_and_b_data.c.

9.13.2.8 get_current_openSubProblemList_iterator_element()

```
SubProblem * get_current_openSubProblemList_iterator_element (
             SubProblemsListIterator * iterator )
```

Definition at line 174 of file b_and_b_data.c.

9.13.2.9 get_SubProblemList_elem_index()

```
SubProblem * get_SubProblemList_elem_index (
             SubProblemsList * list,
             size_t index )
```

Get a SubProblem from a specific index of a SubProblem List.

Parameters

list	The SubProblem List to inspect.
index	The index of the SubProblem to get.

Returns

The SubProblem at the specified index.

Definition at line 146 of file b_and_b_data.c.

9.13.2.10 get_SubProblemList_size()

```
size_t get_SubProblemList_size (
             SubProblemsList * list )
```

Get the size of a SubProblem List.

Parameters

list	The SubProblem List to inspect.

Returns

The size of the SubProblem List.

Definition at line 54 of file b_and_b_data.c.

9.13.2.11 is_SubProblemList_empty()

```
bool is_SubProblemList_empty (
            SubProblemsList * list )
```

Check if a SubProblem List is empty.

Parameters

list	The SubProblem List to check.

Returns

True if the SubProblem List is empty, false otherwise.

Definition at line 39 of file b_and_b_data.c.

9.13.2.12 is_SubProblemList_iterator_valid()

```
bool is_SubProblemList_iterator_valid (
            SubProblemsListIterator * iterator )
```

Check if a SubProblem List iterator is valid.

An iterator is valid if it is not NULL and if the current element is not NULL.

Parameters

iterator	The SubProblem List iterator to check.

Returns

True if the SubProblem List iterator is valid, false otherwise.

Definition at line 170 of file b_and_b_data.c.

9.13.2.13 list_openSubProblemList_next()

```
void list_openSubProblemList_next (
            SubProblemsListIterator * iterator )
```

Definition at line 178 of file b_and_b_data.c.

9.13.2.14 new_SubProblemList()

```
void new_SubProblemList (
            SubProblemsList * list )
```

Create a new SubProblem List.

Parameters

list	The SubProblem List to create.

Definition at line 17 of file b_and_b_data.c.

9.13.2.15 SubProblemList_iterator_get_next()

```
SubProblem * SubProblemList_iterator_get_next (
            SubProblemsListIterator * iterator )
```

Get the next element of a SubProblem List iterator.

Parameters

iterator	The SubProblem List iterator.

Returns

The next element of the List pointed by the iterator.

Definition at line 188 of file b_and_b_data.c.

9.14 b_and_b_data.c

Go to the documentation of this file.

```
00001
00014 #include "b_and_b_data.h"
00015
00016
00017 void new_SubProblemList(SubProblemsList * list){
00018     list->size = 0;
```

```
    list->head = list->tail = NULL;
}


void delete_SubProblemList(SubProblemsList * list){
    if (!list) {
        return;
    }

    SubProblemElem *current = list->head;
    SubProblemElem *next;

    while (current) {
        next = current->next;
        free(current);
        current = next;
    }
}


bool is_SubProblemList_empty(SubProblemsList *list){
    return (list == NULL || list->size == 0);
}


SubProblemElem *build_list_elem(SubProblem *value, SubProblemElem *next, SubProblemElem *prev) {
    SubProblemElem *e = malloc(sizeof(SubProblemElem));
    e->subProblem = *value;
    e->next = next;
    e->prev = prev;

    return e;
}


size_t get_SubProblemList_size(SubProblemsList *list){
    return (list != NULL) ? list->size : 0;
}


void add_elem_SubProblemList_bottom(SubProblemsList *list, SubProblem *element){
    if (list == NULL) {
        return;
    }

    SubProblemElem *e = build_list_elem(element, NULL, list->tail);

    if (is_SubProblemList_empty(list))
        list->head = e;
    else
        list->tail->next = e;
    list->tail = e;
    list->size++;
}


void add_elem_SubProblemList_index(SubProblemsList *list, SubProblem *element, size_t index){
    if (!list || index > get_SubProblemList_size(list)) {
        return;
    }

// support element is a temporary pointer which avoids losing data
    SubProblemElem *e;
    SubProblemElem *supp = list->head;

    for (size_t i = 0; i < index; ++i)
        supp = supp->next;

    if (supp == list->head) {
        e = build_list_elem(element, supp, NULL);

        if (supp == NULL) {
            list->head = list->tail = e;
        } else {
//          e->next->prev = e;
            list->head->prev = e;
            list->head = e;
        }
    } else {
        if (supp == NULL) {
            e = build_list_elem(element, NULL, list->tail);
            list->tail->next = e;
        } else {
            e = build_list_elem(element, supp, supp->prev);
            e->next->prev = e;
            e->prev->next = e;
        }
```

```
    }

    list->size++;
}


void delete_SubProblemList_elem_index(SubProblemsList *list, size_t index){
    if (list == NULL || is_SubProblemList_empty(list) || index >= get_SubProblemList_size(list)) {
        return;
    }

    SubProblemElem *oldElem;
    oldElem = list->head;

    for (size_t i = 0; i < index; ++i)
        oldElem = oldElem->next;

    // Found index  to remove!!
    if (oldElem != list->head) {
        oldElem->prev->next = oldElem->next;
        if (oldElem->next != NULL) {
            oldElem->next->prev = oldElem->prev;
        } else {
            list->tail = oldElem->prev;
        }
    } else {
        if (list->head == list->tail) {
            list->head = list->tail = NULL;
        } else {
            list->head = list->head->next;
            list->head->prev = NULL;
        }
    }

    free(oldElem);
    list->size--;
}


SubProblem *get_SubProblemList_elem_index(SubProblemsList *list, size_t index){
    if (list == NULL  || index >= get_SubProblemList_size(list)) {
        return NULL;
    }

    SubProblemElem *supp; // iteration support element
    supp = list->head;

    for (size_t i = 0; i < index; ++i)
        supp = supp->next;
    return &supp->subProblem;
}

SubProblemsListIterator *create_SubProblemList_iterator(SubProblemsList *list) {
    if (!list)
        return NULL;

    SubProblemsListIterator *new_iterator = malloc(sizeof(SubProblemsListIterator));
    new_iterator->list = list;
    new_iterator->curr = new_iterator->list->head;
    new_iterator->index = 0;
    return new_iterator;
}

bool is_SubProblemList_iterator_valid(SubProblemsListIterator *iterator){
    return (iterator) ? iterator->index < get_SubProblemList_size(iterator->list) : 0;
}

SubProblem *get_current_openSubProblemList_iterator_element(SubProblemsListIterator *iterator) {
    return (iterator && iterator->curr) ? &iterator->curr->subProblem : NULL;
}

void list_openSubProblemList_next(SubProblemsListIterator *iterator) {
    if (is_SubProblemList_iterator_valid(iterator)) {
        iterator->index++;

        if (is_SubProblemList_iterator_valid(iterator)) {
            iterator->curr = iterator->curr->next;
        }
    }
}

SubProblem *SubProblemList_iterator_get_next(SubProblemsListIterator *iterator){
    if (!is_SubProblemList_iterator_valid(iterator)) {
        return NULL;
    }

    SubProblem *element = get_current_openSubProblemList_iterator_element(iterator);
```

```
    list_openSubProblemList_next(iterator);
    return element;
}

void delete_SubProblemList_iterator(SubProblemsListIterator *iterator){
    free(iterator);
}
```

9.15 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/b_and_b_data.h File Reference

The data structures used in the Branch and Bound algorithm.

`#include "mst.h"`

Classes

- struct SubProblem

  The struct used to represent a SubProblem or node of the Branch and Bound tree.
- struct Problem

  The struct used to represent the overall problem.
- struct SubProblemElem

  The element of the list of SubProblems.
- struct SubProblemsList

  The list of open SubProblems.
- struct SubProblemsListIterator

  The iterator of the list of SubProblems.

Typedefs

- typedef enum BBNodeType BBNodeType

  The labels used to identify the type of a SubProblem.
- typedef enum ConstraintType ConstraintType

  The enum used to identify the type of Edge constraints.
- typedef struct SubProblem SubProblem

  The struct used to represent a SubProblem or node of the Branch and Bound tree.
- typedef struct Problem Problem

  The struct used to represent the overall problem.
- typedef struct SubProblemElem SubProblemElem

  The element of the list of SubProblems.
- typedef struct SubProblemsList SubProblemsList

  The list of open SubProblems.

Enumerations

- enum BBNodeType {
  OPEN , CLOSED_NN , CLOSED_NN_HYBRID , CLOSED_BOUND ,
  CLOSED_UNFEASIBLE , CLOSED_1TREE , CLOSED_SUBGRADIENT }

  The labels used to identify the type of a SubProblem.
- enum ConstraintType { NOTHING , MANDATORY , FORBIDDEN }

  The enum used to identify the type of Edge constraints.

Functions

- void new_SubProblemList (SubProblemsList ∗list)

  Create a new SubProblem List.
- void delete_SubProblemList (SubProblemsList ∗list)

  Delete a SubProblem List.
- bool is_SubProblemList_empty (SubProblemsList ∗list)

  Check if a SubProblem List is empty.
- size_t get_SubProblemList_size (SubProblemsList ∗list)

  Get the size of a SubProblem List.
- void add_elem_SubProblemList_bottom (SubProblemsList ∗list, SubProblem ∗element)

  Add a SubProblem to the bottom of a SubProblem List.
- void add_elem_SubProblemList_index (SubProblemsList ∗list, SubProblem ∗element, size_t index)

  Add a SubProblem at a specific index of a SubProblem List.
- void delete_SubProblemList_elem_index (SubProblemsList ∗list, size_t index)

  Remove a SubProblem from a specific index of a SubProblem List.
- SubProblem ∗ get_SubProblemList_elem_index (SubProblemsList ∗list, size_t index)

  Get a SubProblem from a specific index of a SubProblem List.
- SubProblemsListIterator ∗ create_SubProblemList_iterator (SubProblemsList ∗list)

  Create a new SubProblem List iterator on a SubProblem List.
- bool is_SubProblemList_iterator_valid (SubProblemsListIterator ∗iterator)

  Check if a SubProblem List iterator is valid.
- SubProblem ∗ SubProblemList_iterator_get_next (SubProblemsListIterator ∗iterator)

  Get the next element of a SubProblem List iterator.
- void delete_SubProblemList_iterator (SubProblemsListIterator ∗iterator)

  Delete a SubProblem List iterator.

9.15.1 Detailed Description

The data structures used in the Branch and Bound algorithm.

Author

Lorenzo Sciandra

Header file that contains the core data structures used in the Branch and Bound algorithm. There are the data structures used to represent the problem, the sub-problems and the list of sub-problems.

Version

1.0.0 @data 2024-05-1

Copyright

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file b_and_b_data.h.

9.15.2 Typedef Documentation

9.15.2.1 BBNodeType

`typedef enum BBNodeType BBNodeType`

The labels used to identify the type of a SubProblem.

9.15.2.2 ConstraintType

`typedef enum ConstraintType ConstraintType`

The enum used to identify the type of Edge constraints.

9.15.2.3 Problem

`typedef struct Problem Problem`

The struct used to represent the overall problem.

9.15.2.4 SubProblem

`typedef struct SubProblem SubProblem`

The struct used to represent a SubProblem or node of the Branch and Bound tree.

9.15.2.5 SubProblemElem

`typedef struct SubProblemElem SubProblemElem`

The element of the list of SubProblems.

9.15.2.6 SubProblemsList

`typedef struct SubProblemsList SubProblemsList`

The list of open SubProblems.

9.15.3 Enumeration Type Documentation

9.15.3.1 BBNodeType

`enum BBNodeType`

The labels used to identify the type of a SubProblem.

Enumerator

OPEN	The SubProblem is a feasible 1Tree, with a value lower than the best solution found so far.
CLOSED_NN	The SubProblem is a feasible 1Tree founded with the Nearest Neighbor algorithm, it is the first feasible solution found.
CLOSED_NN_HYBRID	The SubProblem is a feasible 1Tree founded with the Hybrid version of Nearest Neighbor algorithm, it is the first feasible solution found.
CLOSED_BOUND	The SubProblem is a feasible 1Tree, with a value greater than the best solution found so far.
CLOSED_UNFEASIBLE	The SubProblem is not a feasible 1Tree, and so discarded.
CLOSED_1TREE	The SubProblem is closed by calculating the 1Tree.
CLOSED_SUBGRADIENT	The SubProblem is closed in the subgradient algorithm, the ascending dual.

Definition at line 22 of file b_and_b_data.h.

9.15.3.2 ConstraintType

```
enum ConstraintType
```

The enum used to identify the type of Edge constraints.

Enumerator

NOTHING	The Edge has no constraints.
MANDATORY	The Edge is mandatory.
FORBIDDEN	The Edge is forbidden.

Definition at line 34 of file b_and_b_data.h.

9.15.4 Function Documentation

9.15.4.1 add_elem_SubProblemList_bottom()

```
void add_elem_SubProblemList_bottom (
            SubProblemsList * list,
            SubProblem * element )
```

Add a SubProblem to the bottom of a SubProblem List.

Parameters

list	The SubProblem List to modify.
element	The SubProblem to add.

Definition at line 59 of file b_and_b_data.c.

9.15.4.2 add_elem_SubProblemList_index()

```
void add_elem_SubProblemList_index (
            SubProblemsList * list,
            SubProblem * element,
            size_t index )
```

Add a SubProblem at a specific index of a SubProblem List.

Parameters

list	The SubProblem List to modify.
element	The SubProblem to add.
index	The index where to add the SubProblem.

list is clearer way but it is already checked inside get_list_size

Definition at line 75 of file b_and_b_data.c.

9.15.4.3 create_SubProblemList_iterator()

```
SubProblemsListIterator * create_SubProblemList_iterator (
            SubProblemsList * list )
```

Create a new SubProblem List iterator on a SubProblem List.

Parameters

list	The SubProblem List to iterate.

Returns

the SubProblem List iterator.

Definition at line 159 of file b_and_b_data.c.

9.15.4.4 delete_SubProblemList()

```
void delete_SubProblemList (
            SubProblemsList * list )
```

Delete a SubProblem List.

Parameters

list	The SubProblem List to delete.

Definition at line 23 of file b_and_b_data.c.

9.15.4.5 delete_SubProblemList_elem_index()

```
void delete_SubProblemList_elem_index (
            SubProblemsList * list,
            size_t index )
```

Remove a SubProblem from a specific index of a SubProblem List.

Parameters

list	The SubProblem List to modify.
index	The index of the SubProblem to remove.

Definition at line 113 of file b_and_b_data.c.

9.15.4.6 delete_SubProblemList_iterator()

```
void delete_SubProblemList_iterator (
            SubProblemsListIterator * iterator )
```

Delete a SubProblem List iterator.

Parameters

iterator	The SubProblem List iterator.

Definition at line 198 of file b_and_b_data.c.

9.15.4.7 get_SubProblemList_elem_index()

```
SubProblem * get_SubProblemList_elem_index (
            SubProblemsList * list,
            size_t index )
```

Get a SubProblem from a specific index of a SubProblem List.

Parameters

list	The SubProblem List to inspect.
index	The index of the SubProblem to get.

Returns

The SubProblem at the specified index.

Definition at line 146 of file b_and_b_data.c.

9.15.4.8 get_SubProblemList_size()

```
size_t get_SubProblemList_size (
            SubProblemsList * list )
```

Get the size of a SubProblem List.

Parameters

list	The SubProblem List to inspect.

Returns

The size of the SubProblem List.

Definition at line 54 of file b_and_b_data.c.

9.15.4.9 is_SubProblemList_empty()

```
bool is_SubProblemList_empty (
            SubProblemsList * list )
```

Check if a SubProblem List is empty.

Parameters

list	The SubProblem List to check.

Returns

True if the SubProblem List is empty, false otherwise.

Definition at line 39 of file b_and_b_data.c.

9.15.4.10 is_SubProblemList_iterator_valid()

```
bool is_SubProblemList_iterator_valid (
            SubProblemsListIterator * iterator )
```

Check if a SubProblem List iterator is valid.

An iterator is valid if it is not NULL and if the current element is not NULL.

Parameters

iterator	The SubProblem List iterator to check.

Returns

True if the SubProblem List iterator is valid, false otherwise.

Definition at line 170 of file b_and_b_data.c.

9.15.4.11 new_SubProblemList()

```
void new_SubProblemList (
            SubProblemsList * list )
```

Create a new SubProblem List.

Parameters

list	The SubProblem List to create.

Definition at line 17 of file b_and_b_data.c.

9.15.4.12 SubProblemList_iterator_get_next()

```
SubProblem * SubProblemList_iterator_get_next (
            SubProblemsListIterator * iterator )
```

Get the next element of a SubProblem List iterator.

Parameters

iterator	The SubProblem List iterator.

Returns

The next element of the List pointed by the iterator.

Definition at line 188 of file b_and_b_data.c.

9.16 b_and_b_data.h

Go to the documentation of this file.

```

#ifndef BRANCHANDBOUND1TREE_B_AND_B_DATA_H
#define BRANCHANDBOUND1TREE_B_AND_B_DATA_H

#include "mst.h"

typedef enum BBNodeType{
    OPEN,
    CLOSED_NN,
    CLOSED_NN_HYBRID,
    CLOSED_BOUND,
    CLOSED_UNFEASIBLE,
    CLOSED_1TREE,
    CLOSED_SUBGRADIENT
}BBNodeType;


typedef enum ConstraintType{
    NOTHING,
    MANDATORY,
    FORBIDDEN
}ConstraintType;


typedef struct SubProblem{
    BBNodeType type;
    unsigned int id;
    unsigned int fatherId;
    double value;
    unsigned short treeLevel;
    float timeToReach;
    MST oneTree;
    unsigned short num_edges_in_cycle;
    double prob;
    ConstrainedEdge cycleEdges [MAX_VERTEX_NUM];
    unsigned short num_forbidden_edges;
    unsigned short num_mandatory_edges;
    int edge_to_branch;
    ConstrainedEdge mandatoryEdges [MAX_VERTEX_NUM];
    ConstraintType constraints [MAX_VERTEX_NUM][MAX_VERTEX_NUM];
}SubProblem;


typedef struct Problem{
    Graph graph;
    Graph reformulationGraph;
    unsigned short candidateNodeId;
    unsigned short totTreeLevels;
    SubProblem bestSolution;
    double bestValue;
    unsigned int generatedBBNodes;
    unsigned int exploredBBNodes;
    unsigned int num_fixed_edges;
    bool interrupted;
    clock_t start;
    clock_t end;
}Problem;


typedef struct SubProblemElem{
    SubProblem subProblem;
    struct SubProblemElem * next;
    struct SubProblemElem * prev;
}SubProblemElem;


typedef struct SubProblemsList{
    SubProblemElem * head;
    SubProblemElem * tail;
    size_t size;
```

```
}SubProblemsList;


typedef struct {
    SubProblemsList * list;
    SubProblemElem * curr;
    size_t index;
} SubProblemsListIterator;


void new_SubProblemList(SubProblemsList * list);


void delete_SubProblemList(SubProblemsList * list);


bool is_SubProblemList_empty(SubProblemsList *list);


size_t get_SubProblemList_size(SubProblemsList *list);


void add_elem_SubProblemList_bottom(SubProblemsList *list, SubProblem *element);


void add_elem_SubProblemList_index(SubProblemsList *list, SubProblem *element, size_t index);


void delete_SubProblemList_elem_index(SubProblemsList *list, size_t index);


SubProblem *get_SubProblemList_elem_index(SubProblemsList *list, size_t index);


SubProblemsListIterator  *create_SubProblemList_iterator(SubProblemsList *list);



bool is_SubProblemList_iterator_valid(SubProblemsListIterator *iterator);


SubProblem *SubProblemList_iterator_get_next(SubProblemsListIterator *iterator);


void delete_SubProblemList_iterator(SubProblemsListIterator *iterator);

#endif //BRANCHANDBOUND1TREE_B_AND_B_DATA_H
```

9.17 GraphConvolutionalBranchandBound/src/HybridSolver/main/data↩ _structures/doubly_linked_list/linked_list.h File Reference

```
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <assert.h>
#include <stdbool.h>
```

Classes

- struct DllElem

  The double linked List element.
- struct List

  The double linked list.
- struct ListIterator

  The iterator for the List.

Macros

- #define BRANCHANDBOUND1TREE_LINKED_LIST_H

Typedefs

- typedef struct DllElem DllElem

  The double linked List element.

9.17.1 Macro Definition Documentation

9.17.1.1 BRANCHANDBOUND1TREE_LINKED_LIST_H

```
#define BRANCHANDBOUND1TREE_LINKED_LIST_H
```

Definition at line 23 of file linked_list.h.

9.17.2 Typedef Documentation

9.17.2.1 DllElem

```
typedef struct DllElem DllElem
```

The double linked List element.

9.18 linked_list.h

Go to the documentation of this file.

```
00001
00015 #pragma once
00016 #include <stdlib.h>
00017 #include <stdio.h>
00018 #include <stddef.h>
00019 #include <string.h>
00020 #include <assert.h>
00021 #include <stdbool.h>
00022 #ifndef BRANCHANDBOUND1TREE_LINKED_LIST_H
00023 #define BRANCHANDBOUND1TREE_LINKED_LIST_H
00024
00025
00027 typedef struct DllElem {
00028     void *value;
00029     struct DllElem *next;
00030     struct DllElem *prev;
00031 } DllElem;
00032
00033
00035 typedef struct {
00036     DllElem *head;
00037     DllElem *tail;
00038     size_t size;
00039 } List;
00040
00041
00043 typedef struct {
00044     List * list;
00045     DllElem* curr;
00046     size_t index;
00047 } ListIterator;
00048
00049
00050 #endif //BRANCHANDBOUND1TREE_LINKED_LIST_H
```

9.19 GraphConvolutionalBranchandBound/src/HybridSolver/main/data↩_structures/doubly_linked_list/list_functions.c File Reference

The definition of the functions to manipulate the List.

```
#include "list_functions.h"
```

Functions

- List ∗ new_list (void)

  Create a new instance of a List.
- void del_list (List ∗list)

  Delete an instance of a List.
- DllElem ∗ build_dll_elem (void ∗value, DllElem ∗next, DllElem ∗prev)
- bool is_list_empty (List ∗list)

  Check if the List is empty.
- size_t get_list_size (List ∗list)

  Gets the size of the List.
- void add_elem_list_bottom (List ∗list, void ∗element)

  Adds an DllElem to the bottom of the List.
- void add_elem_list_index (List ∗list, void ∗element, size_t index)

  Adds an DllElem at the index indicated of the List.
- void delete_list_elem_bottom (List ∗list)

  Deletes the DllElem at the bottom of the List.
- void delete_list_elem_index (List ∗list, size_t index)

  Deletes the DllElem at the indicated index of the List.
- void ∗ get_list_elem_index (List ∗list, size_t index)

  Retrieves a pointer to an DllElem from the List.

9.19.1 Detailed Description

The definition of the functions to manipulate the List.

Authors

Lorenzo Sciandra, Stefano Vittorio Porta and Ivan Spada

This is a double linked List implementation that we have realized for an university project.

Version

1.0.0

Date

2019-07-9

Copyright

Repo: https://gitlab.com/Stefa168/laboratorio-algoritmi-2018-19/

Definition in file list_functions.c.

9.19.2 Function Documentation

9.19.2.1 add_elem_list_bottom()

```
void add_elem_list_bottom (
            List * list,
            void * element )
```

Adds an DllElem to the bottom of the List.

Parameters

list	The List to add the DllElem to.
element	The DllElem to add.

Definition at line 66 of file list_functions.c.

9.19.2.2 add_elem_list_index()

```
void add_elem_list_index (
            List * array,
            void * element,
            size_t index )
```

Adds an DllElem at the index indicated of the List.

Parameters

array	The List to add the DllElem to.
element	The DllElem to add.
index	At what index to add the DllElem to the List.

list is clearer way but it is already checked inside get_list_size

Definition at line 86 of file list_functions.c.

9.19.2.3 build_dll_elem()

```
DllElem * build_dll_elem (
            void * value,
            DllElem * next,
            DllElem * prev )
```

Definition at line 46 of file list_functions.c.

9.19.2.4 del_list()

```
void del_list (
            List * list )
```

Delete an instance of a List.

This method deallocates only the data structure, NOT the data contained.

Parameters

list	The List to delete.

Definition at line 28 of file list_functions.c.

9.19.2.5 delete_list_elem_bottom()

```
void delete_list_elem_bottom (
            List * list )
```

Deletes the DllElem at the bottom of the List.

Parameters

list	The List to remove the DllElem from.

Definition at line 127 of file list_functions.c.

9.19.2.6 delete_list_elem_index()

```
void delete_list_elem_index (
            List * list,
            size_t index )
```

Deletes the DllElem at the indicated index of the List.

Parameters

list	The List to remove the DllElem from.
index	The index of the DllElem to remove from the List.

list is clearer but it is already checked inside get_list_size

Definition at line 147 of file list_functions.c.

9.19.2.7 get_list_elem_index()

```
void * get_list_elem_index (
            List * list,
            size_t index )
```

Retrieves a pointer to an DllElem from the List.

Parameters

list	The List to retrieve the DllElem from.
index	The index of the DllElem to retrieve.

Returns

A pointer to the retrieved DllElem.

Definition at line 185 of file list_functions.c.

9.19.2.8 get_list_size()

```
size_t get_list_size (
            List * list )
```

Gets the size of the List.

Parameters

list	Pointer to the List to check.

Returns

Size of the List l.

Definition at line 61 of file list_functions.c.

9.19.2.9 is_list_empty()

```
bool is_list_empty (
            List * list )
```

Check if the List is empty.

Parameters

list	Pointer to the List to check.

Returns

true if empty, false otherwise.

Definition at line 56 of file list_functions.c.

9.19.2.10 new_list()

```
List * new_list (
            void )
```

Create a new instance of a List.

Returns

The newly created List.

Definition at line 18 of file list_functions.c.

9.20 list_functions.c

Go to the documentation of this file.

```
00001
00015 #include "list_functions.h"
00016
00017
00018 List *new_list(void) {
00019     List *l = calloc(1, sizeof(List));
00020
00021     l->size = 0;
00022     l->head = l->tail = NULL;
00023
00024     return l;
00025 }
00026
00027
00028 void del_list(List *list) {
00029     if (!list) {
00030         return;
00031     }
00032
00033     DllElem *current = list->head;
00034     DllElem *next;
00035
00036     while (current) {
00037         next = current->next;
00038         free(current);
00039         current = next;
00040     }
00041
00042     free(list);
00043 }
00044
00045
00046 DllElem *build_dll_elem(void *value, DllElem *next, DllElem *prev) {
00047     DllElem *e = malloc(sizeof(DllElem));
00048     e->value = value;
00049     e->next = next;
00050     e->prev = prev;
00051
00052     return e;
00053 }
00054
00055
00056 bool is_list_empty(List *list) {
00057     return (list == NULL || !(list->head));
```

```
}


size_t get_list_size(List *list) {
    return (list != NULL) ? list->size : 0;
}


void add_elem_list_bottom(List *list, void *element) {
    if (list == NULL) {
        return;
    }

    DllElem *e = build_dll_elem(element, NULL, list->tail);

    if (is_list_empty(list))
        list->head = e;
    else
        list->tail->next = e;
    list->tail = e;
    list->size++;
}


/*
 * This method deletes the element at the indicated index.
 * If the index is greater than the size of the List, no element is removed.
 */
void add_elem_list_index(List *list, void *element, size_t index) {
    if (!list || index > get_list_size(list)) {
        return;
    }

// support element is a temporary pointer which avoids losing data
    DllElem *e;
    DllElem *supp = list->head;

    for (size_t i = 0; i < index; ++i)
        supp = supp->next;

    if (supp == list->head) {
        e = build_dll_elem(element, supp, NULL);

        if (supp == NULL) {
            list->head = list->tail = e;
        } else {
//          e->next->prev = e;
            list->head->prev = e;
            list->head = e;
        }
    } else {
        if (supp == NULL) {
            e = build_dll_elem(element, NULL, list->tail);
            list->tail->next = e;
        } else {
            e = build_dll_elem(element, supp, supp->prev);
            e->next->prev = e;
            e->prev->next = e;
        }
    }

    list->size++;
}


/*
 * This method deletes the element at the bottom of the List.
 */
void delete_list_elem_bottom(List *list) {

    if (list == NULL || is_list_empty(list)) {
        return;
    }

    DllElem *oldTail = list->tail;

    list->tail = oldTail->prev;
    list->tail->next = NULL;

    free(oldTail);
    list->size--;
}


/*
 * This method iteratively finds and deletes the element at the specified index, but only if it doesn't exceed
```

```c
 * the size of the List. In this case, instead, no reference gets deleted.
 */
void delete_list_elem_index(List *list, size_t index) {
    if (list == NULL || is_list_empty(list) || index >= get_list_size(list)) {
        return;
    }

    DllElem *oldElem;
    oldElem = list->head;

    for (size_t i = 0; i < index; ++i)
        oldElem = oldElem->next;

    // Found index  to remove!!
    if (oldElem != list->head) {
        oldElem->prev->next = oldElem->next;
        if (oldElem->next != NULL) {
            oldElem->next->prev = oldElem->prev;
        } else {
            list->tail = oldElem->prev;
        }
    } else {
        if (list->head == list->tail) {
            list->head = list->tail = NULL;
        } else {
            list->head = list->head->next;
            list->head->prev = NULL;
        }
    }

    free(oldElem);
    list->size--;
}


/*
 * This method iteratively runs through the dllist elements and returns the one at the requested index.
 * If the index exceeds the size of the List, we instead return no element.
 */
void *get_list_elem_index(List *list, size_t index) {
    if (list == NULL  || index >= get_list_size(list)) {
        return NULL;
    }

    DllElem *supp; // iteration support element
    supp = list->head;

    for (size_t i = 0; i < index; ++i)
        supp = supp->next;
    return supp->value;
}
```

9.21 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_list/list_functions.h File Reference

The declaration of the functions to manipulate the List.

```c
#include "linked_list.h"
```

Functions

- List ∗ new_list (void)

  Create a new instance of a List.
- void del_list (List ∗list)

  Delete an instance of a List.
- bool is_list_empty (List ∗list)

  Check if the List is empty.
- size_t get_list_size (List ∗list)

Gets the size of the List.

- void add_elem_list_bottom (List ∗list, void ∗element)

  Adds an DllElem to the bottom of the List.

- void add_elem_list_index (List ∗array, void ∗element, size_t index)

  Adds an DllElem at the index indicated of the List.

- void delete_list_elem_bottom (List ∗list)

  Deletes the DllElem at the bottom of the List.

- void delete_list_elem_index (List ∗list, size_t index)

  Deletes the DllElem at the indicated index of the List.

- void ∗ get_list_elem_index (List ∗list, size_t index)

  Retrieves a pointer to an DllElem from the List.

9.21.1 Detailed Description

The declaration of the functions to manipulate the List.

Authors

Lorenzo Sciandra, Stefano Vittorio Porta and Ivan Spada

This is a double linked List implementation that we have realized for an university project.

Version

1.0.0

Date

2019-07-9

Copyright

Repo: `https://gitlab.com/Stefa168/laboratorio-algoritmi-2018-19/`

Definition in file list_functions.h.

9.21.2 Function Documentation

9.21.2.1 add_elem_list_bottom()

```
void add_elem_list_bottom (
            List * list,
            void * element )
```

Adds an DllElem to the bottom of the List.

Parameters

list	The List to add the DllElem to.
element	The DllElem to add.

Definition at line 66 of file list_functions.c.

9.21.2.2 add_elem_list_index()

```
void add_elem_list_index (
            List * array,
            void * element,
            size_t index )
```

Adds an DllElem at the index indicated of the List.

Parameters

array	The List to add the DllElem to.
element	The DllElem to add.
index	At what index to add the DllElem to the List.

list is clearer way but it is already checked inside get_list_size

Definition at line 86 of file list_functions.c.

9.21.2.3 del_list()

```
void del_list (
            List * list )
```

Delete an instance of a List.

This method deallocates only the data structure, NOT the data contained.

Parameters

list	The List to delete.

Definition at line 28 of file list_functions.c.

9.21.2.4 delete_list_elem_bottom()

```
void delete_list_elem_bottom (
            List * list )
```

Deletes the DllElem at the bottom of the List.

Parameters

list	The List to remove the DllElem from.

Definition at line 127 of file list_functions.c.

9.21.2.5 delete_list_elem_index()

```
void delete_list_elem_index (
            List * list,
            size_t index )
```

Deletes the DllElem at the indicated index of the List.

Parameters

list	The List to remove the DllElem from.
index	The index of the DllElem to remove from the List.

list is clearer but it is already checked inside get_list_size

Definition at line 147 of file list_functions.c.

9.21.2.6 get_list_elem_index()

```
void * get_list_elem_index (
            List * list,
            size_t index )
```

Retrieves a pointer to an DllElem from the List.

Parameters

list	The List to retrieve the DllElem from.
index	The index of the DllElem to retrieve.

Returns

A pointer to the retrieved DllElem.

Definition at line 185 of file list_functions.c.

9.21.2.7 get_list_size()

```
size_t get_list_size (
            List * list )
```

Gets the size of the List.

Parameters

list	Pointer to the List to check.

Returns

Size of the List l.

Definition at line 61 of file list_functions.c.

9.21.2.8 is_list_empty()

```
bool is_list_empty (
            List * list )
```

Check if the List is empty.

Parameters

list	Pointer to the List to check.

Returns

true if empty, false otherwise.

Definition at line 56 of file list_functions.c.

9.21.2.9 new_list()

```
List * new_list (
            void )
```

Create a new instance of a List.

Returns

The newly created List.

Definition at line 18 of file list_functions.c.

9.22 list_functions.h

Go to the documentation of this file.

```
#ifndef BRANCHANDBOUND1TREE_LIST_FUNCTIONS_H
#define BRANCHANDBOUND1TREE_LIST_FUNCTIONS_H

#include "linked_list.h"


List *new_list(void);


void del_list(List *list);


bool is_list_empty(List *list);


size_t get_list_size(List *list);


void add_elem_list_bottom(List *list, void *element);


void add_elem_list_index(List *array, void *element, size_t index);


void delete_list_elem_bottom(List *list);


void delete_list_elem_index(List *list, size_t index);


void *get_list_elem_index(List *list, size_t index);


#endif //BRANCHANDBOUND1TREE_LIST_FUNCTIONS_H
```

9.23 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_list/list_iterator.c File Reference

The definition of the functions to manipulate the ListIterator.

```
#include "list_functions.h"
```

Functions

- ListIterator ∗ create_list_iterator (List ∗list)

  Used for the creation of a new ListIterator.
- void ∗ get_current_list_iterator_element (ListIterator ∗iterator)

  Method used to get the current DllElem of an ListIterator.
- bool is_list_iterator_valid (ListIterator ∗iterator)

  Used to check if the ListIterator is valid.
- void list_iterator_next (ListIterator ∗iterator)

  Used to move the ListIterator to the next value of the object.
- void delete_list_iterator (ListIterator ∗iterator)

  Delete the ListIterator given.
- void ∗ list_iterator_get_next (ListIterator ∗iterator)

  Method that retrieves the current DllElem of an ListIterator and moves the pointer to the next object.

9.23.1 Detailed Description

The definition of the functions to manipulate the ListIterator.

Authors

Lorenzo Sciandra, Stefano Vittorio Porta and Ivan Spada

This is a double linked List implementation that we have realized for an university project.

Version

1.0.0

Date

2019-07-9

Copyright

Repo: `https://gitlab.com/Stefa168/laboratorio-algoritmi-2018-19/`

Definition in file list_iterator.c.

9.23.2 Function Documentation

9.23.2.1 create_list_iterator()

```
ListIterator * create_list_iterator (
            List * list )
```

Used for the creation of a new ListIterator.

Parameters

The	List that the new ListIterator will point.

Returns

A new ListIterator.

Definition at line 18 of file list_iterator.c.

9.23.2.2 delete_list_iterator()

```
void delete_list_iterator (
            ListIterator * iterator )
```

Delete the ListIterator given.

Parameters

An	ListIterator.

Definition at line 51 of file list_iterator.c.

9.23.2.3 get_current_list_iterator_element()

```
void * get_current_list_iterator_element (
            ListIterator * iterator )
```

Method used to get the current DllElem of an ListIterator.

Parameters

An	ListIterator.

Returns

A pointer to the current DllElem.

Definition at line 30 of file list_iterator.c.

9.23.2.4 is_list_iterator_valid()

```
bool is_list_iterator_valid (
            ListIterator * iterator )
```

Used to check if the ListIterator is valid.

Parameters

The	Iterator we want to analyze.

Returns

true if it's valid, false otherwise.

Definition at line 35 of file list_iterator.c.

9.23.2.5 list_iterator_get_next()

```
void * list_iterator_get_next (
            ListIterator * iterator )
```

Method that retrieves the current DllElem of an ListIterator and moves the pointer to the next object.

Parameters

iterator	The ListIterator to use.

Returns

The currently pointed object.

Definition at line 56 of file list_iterator.c.

9.23.2.6 list_iterator_next()

```
void list_iterator_next (
            ListIterator * iterator )
```

Used to move the ListIterator to the next value of the object.

Parameters

l	The ListIterator considered.

Definition at line 40 of file list_iterator.c.

9.24 list_iterator.c

Go to the documentation of this file.

```
00001
00015 #include "list_functions.h"
00016
00017
00018 ListIterator *create_list_iterator(List *list) {
00019     if (!list)
00020         return NULL;
00021
00022     ListIterator *new_iterator = malloc(sizeof(ListIterator));
00023     new_iterator->list = list;
00024     new_iterator->curr = new_iterator->list->head;
00025     new_iterator->index = 0;
00026     return new_iterator;
00027 }
```

```


void *get_current_list_iterator_element(ListIterator *iterator) {
    return (iterator && iterator->curr && iterator->curr->value) ? iterator->curr->value : NULL;
}


bool is_list_iterator_valid(ListIterator *iterator) {
    return (iterator) ? iterator->index < get_list_size(iterator->list) : 0;
}


void list_iterator_next(ListIterator *iterator) {
    if (is_list_iterator_valid(iterator)) {
        iterator->index++;

        if (is_list_iterator_valid(iterator)) {
            iterator->curr = iterator->curr->next;
        }
    }
}


void delete_list_iterator(ListIterator *iterator) {
    free(iterator);
}


void *list_iterator_get_next(ListIterator *iterator) {
    if (!is_list_iterator_valid(iterator)) {
        return NULL;
    }

    void *element = get_current_list_iterator_element(iterator);
    list_iterator_next(iterator);
    return element;
}
```

9.25 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_list/list_iterator.h File Reference

The declaration of the functions to manipulate the ListIterator.

```
#include "linked_list.h"
```

Functions

- ListIterator ∗ create_list_iterator (List ∗list)

    Used for the creation of a new ListIterator.
- bool is_list_iterator_valid (ListIterator ∗iterator)

    Used to check if the ListIterator is valid.
- void ∗ get_current_list_iterator_element (ListIterator ∗iterator)

    Method used to get the current DllElem of an ListIterator.
- void list_iterator_next (ListIterator ∗iterator)

    Used to move the ListIterator to the next value of the object.
- void ∗ list_iterator_get_next (ListIterator ∗iterator)

    Method that retrieves the current DllElem of an ListIterator and moves the pointer to the next object.
- void delete_list_iterator (ListIterator ∗iterator)

    Delete the ListIterator given.

9.25.1 Detailed Description

The declaration of the functions to manipulate the ListIterator.

Authors

Lorenzo Sciandra, Stefano Vittorio Porta and Ivan Spada

This is a double linked List implementation that we have realized for an university project.

Version

1.0.0

Date

2019-07-9

Copyright

Copyright (c) 2024, license MIT

Repo: `https://gitlab.com/Stefa168/laboratorio-algoritmi-2018-19/`

Definition in file list_iterator.h.

9.25.2 Function Documentation

9.25.2.1 create_list_iterator()

```
ListIterator * create_list_iterator (
            List * list )
```

Used for the creation of a new ListIterator.

Parameters

The	List that the new ListIterator will point.

Returns

A new ListIterator.

Definition at line 18 of file list_iterator.c.

9.25.2.2 delete_list_iterator()

```
void delete_list_iterator (
            ListIterator * iterator )
```

Delete the ListIterator given.

Parameters

An	ListIterator.

Definition at line 51 of file list_iterator.c.

9.25.2.3 get_current_list_iterator_element()

```
void * get_current_list_iterator_element (
            ListIterator * iterator )
```

Method used to get the current DllElem of an ListIterator.

Parameters

An	ListIterator.

Returns

A pointer to the current DllElem.

Definition at line 30 of file list_iterator.c.

9.25.2.4 is_list_iterator_valid()

```
bool is_list_iterator_valid (
            ListIterator * iterator )
```

Used to check if the ListIterator is valid.

Parameters

The	Iterator we want to analyze.

Returns

true if it's valid, false otherwise.

Definition at line 35 of file list_iterator.c.

9.25.2.5 list_iterator_get_next()

```
void * list_iterator_get_next (
            ListIterator * iterator )
```

Method that retrieves the current DllElem of an ListIterator and moves the pointer to the next object.

Parameters

iterator	The ListIterator to use.

Returns

The currently pointed object.

Definition at line 56 of file list_iterator.c.

9.25.2.6 list_iterator_next()

```
void list_iterator_next (
            ListIterator * iterator )
```

Used to move the ListIterator to the next value of the object.

Parameters

I	The ListIterator considered.

Definition at line 40 of file list_iterator.c.

9.26 list_iterator.h

Go to the documentation of this file.

```
#ifndef BRANCHANDBOUND1TREE_LIST_ITERATOR_H
#define BRANCHANDBOUND1TREE_LIST_ITERATOR_H
#include "linked_list.h"

ListIterator *create_list_iterator(List *list);


bool is_list_iterator_valid(ListIterator *iterator);


void *get_current_list_iterator_element(ListIterator *iterator);


```

```
void list_iterator_next(ListIterator *iterator);

void *list_iterator_get_next(ListIterator *iterator);

void delete_list_iterator(ListIterator *iterator);

#endif //BRANCHANDBOUND1TREE_LIST_ITERATOR_H
```

9.27 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/fibonacci_heap.c File Reference

This file contains the implementation of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem.

`#include "fibonacci_heap.h"`

Functions

- void create_fibonacci_heap (FibonacciHeap *heap)
  Create an empty Fibonacci Heap.
- void create_node (OrdTreeNode *node, unsigned short key, double value)
  Create a Node with a given key and value.
- void insert_node (FibonacciHeap *heap, OrdTreeNode *node)
  Insert a Node in the Fibonacci Heap.
- void create_insert_node (FibonacciHeap *heap, OrdTreeNode *node, unsigned short key, double value)
  A wrapper function to create a Node and insert it in the Fibonacci Heap.
- void link_trees (FibonacciHeap *heap, OrdTreeNode *child, OrdTreeNode *father)
- void swap_roots (FibonacciHeap *heap, OrdTreeNode *node1, OrdTreeNode *node2)
- void consolidate (FibonacciHeap *heap)
- int extract_min (FibonacciHeap *heap)
  Extract the minimum Node from the Fibonacci Heap.
- static void cut (FibonacciHeap *heap, OrdTreeNode *node, OrdTreeNode *parent)
- static void cascading_cut (FibonacciHeap *heap, OrdTreeNode *node)
- void decrease_value (FibonacciHeap *heap, OrdTreeNode *node, double new_value)
  Decrease the value of a Node in the Fibonacci Heap.
- void delete_node (FibonacciHeap *heap, OrdTreeNode *node)

9.27.1 Detailed Description

This file contains the implementation of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file fibonacci_heap.c.

9.27.2 Function Documentation

9.27.2.1 cascading_cut()

```
static void cascading_cut (
            FibonacciHeap * heap,
            OrdTreeNode * node ) [static]
```

Definition at line 280 of file fibonacci_heap.c.

9.27.2.2 consolidate()

```
void consolidate (
            FibonacciHeap * heap )
```

Definition at line 114 of file fibonacci_heap.c.

9.27.2.3 create_fibonacci_heap()

```
void create_fibonacci_heap (
            FibonacciHeap * heap )
```

Create an empty Fibonacci Heap.

Parameters

heap	The Fibonacci Heap to be created.

Definition at line 16 of file fibonacci_heap.c.

9.27.2.4 create_insert_node()

```
void create_insert_node (
            FibonacciHeap * heap,
            OrdTreeNode * node,
            unsigned short key,
            double value )
```

A wrapper function to create a Node and insert it in the Fibonacci Heap.

Parameters

heap	The Fibonacci Heap where the Node will be inserted.
node	The Node to be created and inserted.
key	The key of the Node.
value	The value of the Node.

Definition at line 63 of file fibonacci_heap.c.

9.27.2.5 create_node()

```
void create_node (
            OrdTreeNode * node,
            unsigned short key,
            double value )
```

Create a Node with a given key and value.

Parameters

node	The Node to be created.
key	The key of the Node.
value	The value of the Node.

Definition at line 25 of file fibonacci_heap.c.

9.27.2.6 cut()

```
static void cut (
            FibonacciHeap * heap,
            OrdTreeNode * node,
            OrdTreeNode * parent ) [static]
```

Definition at line 251 of file fibonacci_heap.c.

9.27.2.7 decrease_value()

```
void decrease_value (
            FibonacciHeap * heap,
            OrdTreeNode * node,
            double new_value )
```

Decrease the value of a Node in the Fibonacci Heap.

If the new value is still greater than the parent's value, nothing happens. Otherwise, the Node becomes a root. If the parent is marked, and is not a root, it becomes a root. This process is repeated until a parent is not marked or is a root.

Parameters

heap	The Fibonacci Heap where the Node is.
node	The Node whose value has to be decreased.
new_value	The new value of the Node.

Definition at line 294 of file fibonacci_heap.c.

9.27.2.8 delete_node()

```
void delete_node (
            FibonacciHeap * heap,
            OrdTreeNode * node )
```

Definition at line 312 of file fibonacci_heap.c.

9.27.2.9 extract_min()

```
int extract_min (
            FibonacciHeap * heap )
```

Extract the minimum Node from the Fibonacci Heap.

All the children of the minimum Node become new roots. The new minimum has to be found and, by doing so, the Heap is re-ordered to maintain the Heap property and minimize the height of the Heap-ordered Trees.

Parameters

heap	The Fibonacci Heap where the Node will be extracted.

Returns

The key of the minimum Node if the Heap is not empty, -1 otherwise.

Definition at line 175 of file fibonacci_heap.c.

9.27.2.10 insert_node()

```
void insert_node (
            FibonacciHeap * heap,
            OrdTreeNode * node )
```

Insert a Node in the Fibonacci Heap.

Parameters

heap	The Fibonacci Heap where the Node will be inserted.
node	The Node to be inserted.

Definition at line 38 of file fibonacci_heap.c.

9.27.2.11 link_trees()

```
void link_trees (
            FibonacciHeap * heap,
            OrdTreeNode * child,
            OrdTreeNode * father )
```

Definition at line 69 of file fibonacci_heap.c.

9.27.2.12 swap_roots()

```
void swap_roots (
            FibonacciHeap * heap,
            OrdTreeNode * node1,
            OrdTreeNode * node2 )
```

Definition at line 95 of file fibonacci_heap.c.

9.28 fibonacci_heap.c

Go to the documentation of this file.

```
00001
00013 #include "fibonacci_heap.h"
00014
00015
00016 void create_fibonacci_heap(FibonacciHeap * heap){
00017     heap->min_root = NULL;
00018     heap->head_tree_list = NULL;
00019     heap->tail_tree_list = NULL;
00020     heap->num_nodes = 0;
00021     heap->num_trees = 0;
00022 }
00023
00024
00025 void create_node(OrdTreeNode * node, unsigned short key, double value){
00026     node->value = value;
00027     node->key = key;
00028     node->parent = NULL;
00029     node->left_sibling = node;
00030     node->right_sibling = node;
00031
00032     node->head_child_list = NULL;
00033     node->tail_child_list = NULL;
00034     node->num_children = 0;
00035 }
00036
00037
00038 void insert_node(FibonacciHeap * heap, OrdTreeNode * node){
00039
00040     node->marked = false;
```

```
    node->is_root = true;

    if(heap->num_trees == 0){
        heap->min_root = node;
        heap->head_tree_list = node;
        heap->tail_tree_list = node;
    }
    else{
        if(node->value < heap->min_root->value){
            heap->min_root = node;
        }
        heap->tail_tree_list->right_sibling = node;
        heap->head_tree_list->left_sibling = node;
        node->left_sibling = heap->tail_tree_list;
        node->right_sibling = heap->head_tree_list;
        heap->tail_tree_list = node;
    }
    heap->num_nodes++;
    heap->num_trees++;
}


void create_insert_node(FibonacciHeap * heap, OrdTreeNode * node, unsigned short key, double value){
    create_node(node, key, value);
    insert_node(heap, node);
}


void link_trees(FibonacciHeap * heap, OrdTreeNode * child, OrdTreeNode * father){
    child->marked = false;
    child->right_sibling->left_sibling = child->left_sibling;
    child->left_sibling->right_sibling = child->right_sibling;
    child->is_root = false;
    heap->num_trees--;

    child->parent = father;
    father->num_children++;
    if(father->num_children == 1){
        father->head_child_list = child;
        father->tail_child_list = child;
        child->left_sibling = child;
        child->right_sibling = child;
    }
    else{
        father->tail_child_list->right_sibling = child;
        father->head_child_list->left_sibling = child;
        child->left_sibling = father->tail_child_list;
        child->right_sibling = father->head_child_list;
        father->tail_child_list = child;
    }

}


void swap_roots(FibonacciHeap * heap, OrdTreeNode * node1, OrdTreeNode * node2){
    if(node1->key == node2->key){
        return;
    }
    else{

        OrdTreeNode * node1_left = node1->left_sibling;
        OrdTreeNode * node1_right = node1->right_sibling;
        OrdTreeNode * node2_left = node2->left_sibling;
        OrdTreeNode * node2_right = node2->right_sibling;

        node1->left_sibling = node2_left;
        node1->right_sibling = node2_right;
        node2->left_sibling = node1_left;
        node2->right_sibling = node1_right;
    }
}


void consolidate(FibonacciHeap * heap){

    int dimension = ((int) ceil(log(heap->num_nodes) / log(2))) + 2;
    OrdTreeNode * degree_list [dimension] ;

    for (int i = 0; i < dimension; i++) {
        degree_list[i] = NULL;
    }

    OrdTreeNode * iter = heap->head_tree_list;
    int num_it = heap->num_trees;

    for (int i = 0; i < num_it; i++) {
        OrdTreeNode * deg_i_node = iter;
```

```
        unsigned short deg = deg_i_node->num_children;
        iter = iter->right_sibling;

        while(degree_list[deg]!=NULL){

            OrdTreeNode * same_deg_node = degree_list[deg];

            if(deg_i_node->value > same_deg_node->value){
                OrdTreeNode * temp = deg_i_node;
                deg_i_node = same_deg_node;
                same_deg_node = temp;
                //iter = deg_i_node;
            }

            link_trees(heap, same_deg_node, deg_i_node);
            // since same_deg_node is now a child of deg_i_node, it is no longer a root
            degree_list[deg] = NULL;
            deg++;
        }
        degree_list[deg] = deg_i_node;
    }

    heap->min_root = NULL;

    for (int i = 0; i < dimension; i++) {
        if(degree_list[i]!=NULL){
            if(heap->min_root == NULL){
                heap->min_root = degree_list[i];
                heap->head_tree_list = degree_list[i];
                heap->tail_tree_list = degree_list[i];
            }
            else{
                if(heap->min_root->value > degree_list[i]->value){
                    heap->min_root = degree_list[i];
                }
                heap->tail_tree_list->right_sibling = degree_list[i];
                heap->head_tree_list->left_sibling = degree_list[i];
                degree_list[i]->left_sibling = heap->tail_tree_list;
                degree_list[i]->right_sibling = heap->head_tree_list;
                heap->tail_tree_list = degree_list[i];
            }
        }
    }

}


int extract_min(FibonacciHeap * heap) {

    int min_pos;

    if (heap->min_root != NULL) {

        min_pos = heap->min_root->key;

        OrdTreeNode *child = heap->min_root->head_child_list;
        unsigned short num_children = heap->min_root->num_children;

        for (unsigned short i = 0; i < num_children; i++) {
            child->parent = NULL;
            child->is_root = true;
            child->marked = false;

            if(num_children - i == 1){
                heap->min_root->head_child_list = NULL;
                heap->min_root->tail_child_list = NULL;
            }
            else{
                if(heap->min_root->head_child_list->key == child->key){
                    heap->min_root->head_child_list = child->right_sibling;
                }
                if(heap->min_root->tail_child_list->key == child->key){
                    heap->min_root->tail_child_list = child->left_sibling;
                }
                child->left_sibling->right_sibling = child->right_sibling;
                child->right_sibling->left_sibling = child->left_sibling;
            }

            heap->tail_tree_list->right_sibling = child;
            heap->head_tree_list->left_sibling = child;
            child->left_sibling = heap->tail_tree_list;
            child->right_sibling = heap->head_tree_list;
            heap->tail_tree_list = child;

            heap->num_trees++;

            child = heap->min_root->head_child_list;
```

```
            heap->min_root->num_children--;
        }

        heap->num_trees--;

        if(heap->head_tree_list->key == heap->min_root->key){
            heap->head_tree_list = heap->head_tree_list->right_sibling;
        }
        if(heap->tail_tree_list->key == heap->min_root->key){
            heap->tail_tree_list = heap->tail_tree_list->left_sibling;
        }

        heap->min_root->left_sibling->right_sibling = heap->min_root->right_sibling;
        heap->min_root->right_sibling->left_sibling = heap->min_root->left_sibling;

        if (heap->min_root->key == heap->min_root->right_sibling->key) {
            // the min root is the only tree in the heap, and it has no children, so the heap is now empty
            heap->min_root = NULL;
            heap->tail_tree_list = NULL;
            heap->head_tree_list = NULL;
            heap->num_trees = 0;
        }
        else{
            // choose a new min root, arbitrarily
            heap->min_root = heap->min_root->right_sibling;
            // re-consolidate the heap
            consolidate(heap);
        }
        heap->num_nodes--;
    } else{
        min_pos = -1;
    }
    return min_pos;
}


static void cut(FibonacciHeap * heap, OrdTreeNode * node, OrdTreeNode * parent){
    node->parent = NULL;
    node->is_root = true;
    node->marked = false;
    parent->num_children--;
    if(parent->num_children == 0){
        parent->head_child_list = NULL;
        parent->tail_child_list = NULL;
    }
    else{
        if(parent->head_child_list->key == node->key){
            parent->head_child_list = node->right_sibling;
        }
        if(parent->tail_child_list->key == node->key){
            parent->tail_child_list = node->left_sibling;
        }
        node->left_sibling->right_sibling = node->right_sibling;
        node->right_sibling->left_sibling = node->left_sibling;
    }

    heap->tail_tree_list->right_sibling = node;
    heap->head_tree_list->left_sibling = node;
    node->left_sibling = heap->tail_tree_list;
    node->right_sibling = heap->head_tree_list;
    heap->tail_tree_list = node;
    heap->num_trees++;
}


static void cascading_cut(FibonacciHeap * heap, OrdTreeNode * node){
    if(node->parent != NULL){
        if(!node->marked){
            node->marked = true;
        }
        else{
            OrdTreeNode * parent = node->parent;
            cut(heap, node, parent);
            cascading_cut(heap, parent);
        }
    }
}


void decrease_value(FibonacciHeap * heap, OrdTreeNode * node, double new_value){
    if(new_value > node->value){
        fprintf(stderr, "Error: new value is greater than current value\n");
        exit(EXIT_FAILURE);
    }
    else{
        node->value = new_value;
```

```
        if(node->parent != NULL && node->value < node->parent->value){
            OrdTreeNode * parent = node->parent;
            cut(heap, node, node->parent);
            cascading_cut(heap, parent);
        }
        if(node->value < heap->min_root->value){
            heap->min_root = node;
        }
    }
}

void delete_node(FibonacciHeap * heap, OrdTreeNode * node){
    decrease_value(heap, node, -FLT_MAX);
    extract_min(heap);
}
```

9.29 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/fibonacci_heap.h File Reference

This file contains the declaration of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem.

```
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <assert.h>
#include <stdbool.h>
#include <float.h>
#include <math.h>
```

Classes

- struct OrdTreeNode

  A Heap-ordered Tree Node where the key of the parent is <= the key of its children.
- struct FibonacciHeap

  The Fibonacci Heap datastructure as collection of Heap-ordered Trees.

Macros

- #define BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H

Typedefs

- typedef struct OrdTreeNode OrdTreeNode

  A Heap-ordered Tree Node where the key of the parent is <= the key of its children.
- typedef struct FibonacciHeap FibonacciHeap

  The Fibonacci Heap datastructure as collection of Heap-ordered Trees.

Functions

- void create_fibonacci_heap (FibonacciHeap ∗heap)

  Create an empty Fibonacci Heap.
- void create_node (OrdTreeNode ∗node, unsigned short key, double value)

  Create a Node with a given key and value.
- void insert_node (FibonacciHeap ∗heap, OrdTreeNode ∗node)

  Insert a Node in the Fibonacci Heap.
- void create_insert_node (FibonacciHeap ∗heap, OrdTreeNode ∗node, unsigned short key, double value)

  A wrapper function to create a Node and insert it in the Fibonacci Heap.
- int extract_min (FibonacciHeap ∗heap)

  Extract the minimum Node from the Fibonacci Heap.
- void decrease_value (FibonacciHeap ∗heap, OrdTreeNode ∗node, double new_value)

  Decrease the value of a Node in the Fibonacci Heap.

9.29.1 Detailed Description

This file contains the declaration of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file fibonacci_heap.h.

9.29.2 Macro Definition Documentation

9.29.2.1 BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H

```
#define BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H
```

Definition at line 22 of file fibonacci_heap.h.

9.29.3 Typedef Documentation

9.29.3.1 FibonacciHeap

```
typedef struct FibonacciHeap FibonacciHeap
```

The Fibonacci Heap datastructure as collection of Heap-ordered Trees.

9.29.3.2 OrdTreeNode

```
typedef struct OrdTreeNode OrdTreeNode
```

A Heap-ordered Tree Node where the key of the parent is <= the key of its children.

9.29.4 Function Documentation

9.29.4.1 create_fibonacci_heap()

```
void create_fibonacci_heap (
            FibonacciHeap * heap )
```

Create an empty Fibonacci Heap.

Parameters

heap	The Fibonacci Heap to be created.

Definition at line 16 of file fibonacci_heap.c.

9.29.4.2 create_insert_node()

```
void create_insert_node (
            FibonacciHeap * heap,
            OrdTreeNode * node,
            unsigned short key,
            double value )
```

A wrapper function to create a Node and insert it in the Fibonacci Heap.

Parameters

heap	The Fibonacci Heap where the Node will be inserted.
node	The Node to be created and inserted.
key	The key of the Node.
value	The value of the Node.

Definition at line 63 of file fibonacci_heap.c.

9.29.4.3 create_node()

```
void create_node (
            OrdTreeNode * node,
            unsigned short key,
            double value )
```

Create a Node with a given key and value.

Parameters

node	The Node to be created.
key	The key of the Node.
value	The value of the Node.

Definition at line 25 of file fibonacci_heap.c.

9.29.4.4 decrease_value()

```
void decrease_value (
            FibonacciHeap * heap,
            OrdTreeNode * node,
            double new_value )
```

Decrease the value of a Node in the Fibonacci Heap.

If the new value is still greater than the parent's value, nothing happens. Otherwise, the Node becomes a root. If the parent is marked, and is not a root, it becomes a root. This process is repeated until a parent is not marked or is a root.

Parameters

heap	The Fibonacci Heap where the Node is.
node	The Node whose value has to be decreased.
new_value	The new value of the Node.

Definition at line 294 of file fibonacci_heap.c.

9.29.4.5 extract_min()

```
int extract_min (
            FibonacciHeap * heap )
```

Extract the minimum Node from the Fibonacci Heap.

All the children of the minimum Node become new roots. The new minimum has to be found and, by doing so, the Heap is re-ordered to maintain the Heap property and minimize the height of the Heap-ordered Trees.

Parameters

heap	The Fibonacci Heap where the Node will be extracted.

Returns

The key of the minimum Node if the Heap is not empty, -1 otherwise.

Definition at line 175 of file fibonacci_heap.c.

9.29.4.6 insert_node()

```
void insert_node (
            FibonacciHeap * heap,
            OrdTreeNode * node )
```

Insert a Node in the Fibonacci Heap.

Parameters

heap	The Fibonacci Heap where the Node will be inserted.
node	The Node to be inserted.

Definition at line 38 of file fibonacci_heap.c.

9.30 fibonacci_heap.h

Go to the documentation of this file.

```

#pragma once
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <assert.h>
#include <stdbool.h>
#include <float.h>
#include <math.h>
#ifndef BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H
#define BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H


typedef struct OrdTreeNode {
    unsigned short key;
    double value;
    struct OrdTreeNode *parent;

    struct OrdTreeNode *left_sibling;
    struct OrdTreeNode *right_sibling;

```

```
    struct OrdTreeNode *head_child_list;
    struct OrdTreeNode *tail_child_list;
    unsigned short num_children;
    bool marked;
    bool is_root;
}OrdTreeNode;

typedef struct FibonacciHeap{
    OrdTreeNode * min_root;
    OrdTreeNode * head_tree_list;
    OrdTreeNode * tail_tree_list;
    unsigned short num_nodes;
    unsigned short num_trees;
}FibonacciHeap;

void create_fibonacci_heap(FibonacciHeap * heap);

void create_node(OrdTreeNode * node, unsigned short key, double value);

void insert_node(FibonacciHeap * heap, OrdTreeNode * node);

void create_insert_node(FibonacciHeap * heap, OrdTreeNode * node, unsigned short key, double value);

int extract_min(FibonacciHeap * heap);

void decrease_value(FibonacciHeap * heap, OrdTreeNode * node, double new_value);

#endif //BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H
```

9.31 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/graph.c File Reference

The implementation of the graph data structure.

```
#include "graph.h"
#include "doubly_linked_list/list_iterator.h"
```

Functions

- void create_graph (Graph ∗graph, List ∗nodes_list, List ∗edges_list, GraphKind kind)

  Create a new instance of a Graph with all the needed parameters.
- void create_euclidean_graph (Graph ∗graph, List ∗nodes)

  Create a new instance of an euclidean graphs only the Nodes are necessary.
- void print_graph (const Graph ∗G)

  Print Nodes, Edges and other information of the Graph.

9.31.1 Detailed Description

The implementation of the graph data structure.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Copyright (c) 2024, license MIT

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file graph.c.

9.31.2 Function Documentation

9.31.2.1 create_euclidean_graph()

```
void create_euclidean_graph (
            Graph * graph,
            List * nodes )
```

Create a new instance of an euclidean graphs only the Nodes are necessary.

Parameters

nodes	Pointer to the List of Nodes.
graph	Pointer to the Graph to be initialized.

Definition at line 71 of file graph.c.

9.31.2.2 create_graph()

```
void create_graph (
            Graph * graph,
            List * nodes,
            List * edges,
            GraphKind kind )
```

Create a new instance of a Graph with all the needed parameters.

Parameters

nodes	Pointer to the List of Nodes.
edges	Pointer to the List of Edges.
kind	Type of the Graph.
graph	Pointer to the Graph to be initialized.

Definition at line 18 of file graph.c.

9.31.2.3 print_graph()

```
void print_graph (
            const Graph * graph )
```

Print Nodes, Edges and other information of the Graph.

Parameters

graph	Pointer to the Graph to be printed.

Definition at line 101 of file graph.c.

9.32 graph.c

Go to the documentation of this file.

```
00001
00014 #include "graph.h"
00015 #include "doubly_linked_list/list_iterator.h"
00016
00017
00018 void create_graph(Graph * graph, List *nodes_list, List *edges_list, GraphKind kind) {
00019     graph->kind = kind;
00020     graph->num_edges = 0;
00021     graph->num_nodes = 0;
00022     graph->orderedEdges = false;
00023     graph->cost = 0;
00024
00025     ListIterator *nodes_iterator = create_list_iterator(nodes_list);
00026     unsigned short numNodes = 0;
00027     for (size_t j = 0; j < nodes_list->size; j++) {
00028         Node *curr = list_iterator_get_next(nodes_iterator);
00029         graph->nodes[numNodes].positionInGraph = numNodes;
00030         graph->nodes[numNodes].num_neighbours = 0;
00031         graph->nodes[numNodes].y = curr->y;
00032         graph->nodes[numNodes].x = curr->x;
00033         graph->num_nodes++;
00034         numNodes++;
00035     }
00036     delete_list_iterator(nodes_iterator);
00037
00038     unsigned short numEdges = 0;
00039     ListIterator *edges_iterator = create_list_iterator(edges_list);
00040     for (size_t i = 0; i < edges_list->size ; i++) {
00041         //add the source vertex to the data_structures
00042         Edge * current_edge = list_iterator_get_next(edges_iterator);
00043         unsigned short src = current_edge->src;
00044         unsigned short dest = current_edge->dest;
00045
00046         graph->edges[numEdges].dest = dest;
00047         graph->edges[numEdges].src = src;
```

```
        graph->edges[numEdges].prob = current_edge->prob;
        graph->edges[numEdges].weight = current_edge->weight;
        graph->edges[numEdges].symbol = current_edge->symbol;
        graph->edges[numEdges].positionInGraph =  numEdges;

        graph->nodes[src].neighbours[graph->nodes[src].num_neighbours] = dest;
        graph->nodes[src].num_neighbours++;
        graph->edges_matrix[src][dest] = graph->edges[numEdges];
        graph->cost += current_edge->weight;

        graph->edges_matrix[dest][src] = graph->edges_matrix[src][dest];
        graph->nodes[dest].neighbours[graph->nodes[dest].num_neighbours] = src;
        graph->nodes[dest].num_neighbours++;

        numEdges++;
        graph->num_edges++;
    }
    delete_list_iterator(edges_iterator);
    del_list(edges_list);
    del_list(nodes_list);
}


void create_euclidean_graph(Graph * graph, List *nodes) {
    List *edges_list = new_list();

    unsigned short z = 0;
    Edge edges [MAX_EDGES_NUM];
    ListIterator *i_nodes_iterator = create_list_iterator(nodes);
    for (size_t i = 0; i < nodes->size; i++) {
        Node *node_src = list_iterator_get_next(i_nodes_iterator);
        for (size_t j = i + 1; j < nodes->size; j++) {
            Node *node_dest = get_list_elem_index(nodes,j);

            edges[z].src = node_src->positionInGraph;
            edges[z].dest = node_dest->positionInGraph;
            edges[z].symbol = z + 1;
            edges[z].positionInGraph = z;
            edges[z].prob = 0;
            edges[z].weight = (float) sqrt(pow(fabs(node_src->x - node_dest->x), 2) +
                                           pow(fabs(node_src->y - node_dest->y), 2));
            add_elem_list_bottom(edges_list, &edges[z]);
            z++;

        }

    }
    delete_list_iterator(i_nodes_iterator);

    create_graph(graph,nodes, edges_list, WEIGHTED_GRAPH);
}


void print_graph(const Graph *G) {
    printf("Nodes: %i\n", G->num_nodes);
    for (int i = 0; i < G->num_nodes; i++) {
        Node curr = G->nodes[i];
        printf("Node%i:\t(%.*f, %.*f)\t%i neighbours: ", curr.positionInGraph, DBL_DIG-1, curr.x, DBL_DIG-1, curr.y, curr.num_neighbours);

        for (int z = 0; z < curr.num_neighbours; z++) {
            printf("%i ", G->nodes[curr.neighbours[z]].positionInGraph);
        }
        printf("\n");
    }

    printf("\nCost: %lf\n", G->cost);
    printf("\nEdges: %i\n", G->num_edges);

    double dim = (log(G->num_nodes) / log(10) + 1) * 2 + 7;
    for (unsigned short j = 0; j < G->num_edges; j++) {
        char edge_print [(int) dim] ;
        char edge_print_dest [(int) (dim-7)/2] ;
        Edge curr = G->edges[j];
        sprintf(edge_print, "%i", curr.src);
        strcat(edge_print, " <--> ");
        sprintf(edge_print_dest, "%i", curr.dest);
        strcat(edge_print, edge_print_dest);

        printf("Edge%i:\t%s\tweight = %.*f\tprob = %.*f\n",
               curr.symbol,
               edge_print,
               DBL_DIG-1,
               curr.weight,
               DBL_DIG-1,
               curr.prob);
    }
```

```
00134
00135 }
```

9.33 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/graph.h File Reference

The data structures to model the Graph.

```
#include "./doubly_linked_list//linked_list.h"
#include "./doubly_linked_list/list_iterator.h"
#include "./doubly_linked_list/list_functions.h"
#include "../problem_settings.h"
```

Classes

- struct Node

  Structure of a Node.
- struct Edge

  Structure of an Edge.
- struct Graph

  Structure of a Graph.

Typedefs

- typedef enum GraphKind GraphKind

  Enum to specify the kind of the Graph.
- typedef struct Node Node

  Structure of a Node.
- typedef struct Edge Edge

  Structure of an Edge.
- typedef struct Graph Graph

  Structure of a Graph.

Enumerations

- enum GraphKind { WEIGHTED_GRAPH , UNWEIGHTED_GRAPH }

  Enum to specify the kind of the Graph.

Functions

- void create_graph (Graph *graph, List *nodes, List *edges, GraphKind kind)

  Create a new instance of a Graph with all the needed parameters.
- void create_euclidean_graph (Graph *graph, List *nodes)

  Create a new instance of an euclidean graphs only the Nodes are necessary.
- void print_graph (const Graph *graph)

  Print Nodes, Edges and other information of the Graph.

9.33.1 Detailed Description

The data structures to model the Graph.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Copyright (c) 2024, license MIT

Repo: `https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound`

Definition in file graph.h.

9.33.2 Typedef Documentation

9.33.2.1 Edge

```
typedef struct Edge Edge
```

Structure of an Edge.

9.33.2.2 Graph

```
typedef struct Graph Graph
```

Structure of a Graph.

9.33.2.3 GraphKind

```
typedef enum GraphKind GraphKind
```

Enum to specify the kind of the Graph.

9.33.2.4 Node

```
typedef struct Node Node
```

Structure of a Node.

9.33.3 Enumeration Type Documentation

9.33.3.1 GraphKind

```
enum GraphKind
```

Enum to specify the kind of the Graph.

Enumerator

WEIGHTED_GRAPH	The Graph is weighted.
UNWEIGHTED_GRAPH	The Graph is unweighted.

Definition at line 23 of file graph.h.

9.33.4 Function Documentation

9.33.4.1 create_euclidean_graph()

```
void create_euclidean_graph (
            Graph * graph,
            List * nodes )
```

Create a new instance of an euclidean graphs only the Nodes are necessary.

Parameters

nodes	Pointer to the List of Nodes.
graph	Pointer to the Graph to be initialized.

Definition at line 71 of file graph.c.

9.33.4.2 create_graph()

```
void create_graph (
            Graph * graph,
            List * nodes,
            List * edges,
            GraphKind kind )
```

Create a new instance of a Graph with all the needed parameters.

Parameters

nodes	Pointer to the List of Nodes.
edges	Pointer to the List of Edges.
kind	Type of the Graph.
graph	Pointer to the Graph to be initialized.

Definition at line 18 of file graph.c.

9.33.4.3 print_graph()

```
void print_graph (
            const Graph * graph )
```

Print Nodes, Edges and other information of the Graph.

Parameters

graph	Pointer to the Graph to be printed.

Definition at line 101 of file graph.c.

9.34 graph.h

Go to the documentation of this file.

```

#ifndef BRANCHANDBOUND_1TREE_GRAPH_H
#define BRANCHANDBOUND_1TREE_GRAPH_H
#include "./doubly_linked_list//linked_list.h"
#include "./doubly_linked_list/list_iterator.h"
#include "./doubly_linked_list/list_functions.h"
#include "../problem_settings.h"


typedef enum GraphKind{
    WEIGHTED_GRAPH,
    UNWEIGHTED_GRAPH
} GraphKind;


typedef struct Node {
    double x;
    double y;
    unsigned short positionInGraph;
    unsigned short num_neighbours;
    unsigned short neighbours [MAX_VERTEX_NUM - 1];
}Node;


typedef struct Edge {
    unsigned short src;
    unsigned short dest;
    unsigned short symbol;
    double weight;
    double prob;
    unsigned short positionInGraph;
}Edge;


typedef struct Graph {
    GraphKind kind;
    double cost;
    unsigned short num_nodes;
    unsigned short num_edges;
    bool orderedEdges;
    Node nodes [MAX_VERTEX_NUM];
    Edge edges [MAX_EDGES_NUM];
    Edge edges_matrix [MAX_VERTEX_NUM] [MAX_VERTEX_NUM];
}Graph;


void create_graph(Graph* graph, List *  nodes, List *  edges, GraphKind kind);


void create_euclidean_graph(Graph * graph, List * nodes);


void print_graph(const Graph * graph);


#endif //BRANCHANDBOUND_1TREE_GRAPH_H
```

9.35 GraphConvolutionalBranchandBound/src/HybridSolver/main/data↵ _structures/mfset.c File Reference

This file contains the implementation of the Merge-Find Set datastructure for the Minimum Spanning Tree problem.

```
#include "mfset.h"
```

Functions

- void create_forest_constrained (Forest *forest, const Node *nodes, unsigned short num_nodes, unsigned short candidateId)

    Create a new Forest with n Sets, each Set containing a Node, with constraints.
- void create_forest (Forest *forest, const Node *nodes, unsigned short num_nodes)

    Create a new Forest with n Sets, each Set containing a Node, without constraints.
- Set * find (Set *set)

    Find the root of a Set.
- void merge (Set *set1, Set *set2)

    Merge two Sets in the Forest if they are not already in the same Set.
- void print_forest (const Forest *forest)

    Print all the Forest.

9.35.1 Detailed Description

This file contains the implementation of the Merge-Find Set datastructure for the Minimum Spanning Tree problem.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Copyright (c) 2024, license MIT

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file mfset.c.

9.35.2 Function Documentation

9.35.2.1 create_forest()

```
void create_forest (
            Forest * forest,
            const Node * nodes,
            unsigned short num_nodes )
```

Create a new Forest with n Sets, each Set containing a Node, without constraints.

Parameters

nodes	Pointer to the List of Nodes.
num_nodes	Number of Nodes in the List.
forest	Pointer to the Forest to be initialized.

Definition at line 31 of file mfset.c.

9.35.2.2 create_forest_constrained()

```
void create_forest_constrained (
            Forest * forest,
            const Node * nodes,
            unsigned short num_nodes,
            unsigned short candidateId )
```

Create a new Forest with n Sets, each Set containing a Node, with constraints.

The candidateId Node is not added to the Forest because for the 1-tree I need a MST on the remaining Nodes.

Parameters

nodes	Pointer to the List of Nodes.
num_nodes	Number of Nodes in the List.
candidateId	Id of the Node in the List to be excluded from the Forest.
forest	Pointer to the Forest to be initialized.

Definition at line 17 of file mfset.c.

9.35.2.3 find()

```
Set * find (
            Set * set )
```

Find the root of a Set.

Complexity: O(log n), only a path in the tree is traversed. The parent Set of all the Nodes in the path are updated to point to the root, to reduce the complexity of the next find operations.

Parameters

set	Pointer to the Set.

Returns

Pointer to the root of the Set.

Definition at line 44 of file mfset.c.

9.35.2.4 merge()

```
void merge (
            Set * set1,
            Set * set2 )
```

Merge two Sets in the Forest if they are not already in the same Set.

The Set with the highest rank is the parent of the other. This is done to let the find operation run in O(log n) time. Complexity: O(log n_1 + log n_2)

Parameters

set1	Pointer to the first Set.
set2	Pointer to the second Set.

Definition at line 53 of file mfset.c.

9.35.2.5 print_forest()

```
void print_forest (
            const Forest * forest )
```

Print all the Forest.

Used for debugging purposes.

Parameters

forest	Pointer to the Forest.

Definition at line 72 of file mfset.c.

9.36 mfset.c

Go to the documentation of this file.

```
00001
00014 #include "mfset.h"
00015
00016
```

```
void create_forest_constrained(Forest *forest, const Node *nodes, unsigned short num_nodes, unsigned short candidateId) {
    forest->num_sets = num_nodes - 1;

    for (unsigned short i = 0; i < num_nodes; i++) {
        if (i != candidateId) {
            forest->sets[i].parentSet = NULL;
            forest->sets[i].rango = 0;
            forest->sets[i].curr = nodes[i];
            forest->sets[i].num_in_forest = i;
        }
    }
}


void create_forest(Forest *forest, const Node *nodes, unsigned short num_nodes) {

    forest->num_sets = num_nodes;
    for (unsigned short i = 0; i < num_nodes; i++) {
        forest->sets[i].parentSet = NULL;
        forest->sets[i].rango = 0;
        forest->sets[i].curr = nodes[i];
        forest->sets[i].num_in_forest = i;
    }

}


Set *find(Set *set) {
    if (set->parentSet != NULL) {
        set->parentSet = find(set->parentSet);
        return set->parentSet;
    }
    return set;
}


void merge(Set *set1, Set *set2) {

    Set *set1_root = find(set1);
    Set *set2_root = find(set2);

    //printf("\nThe root are %.2fd ,%d\n", set1_root->num_in_forest, set2_root->num_in_forest);
    if (set1_root->num_in_forest != set2_root->num_in_forest) {
        if (set1_root->rango > set2_root->rango) {
            set2_root->parentSet = set1_root;
        } else if (set1_root->rango < set2_root->rango) {
            set1_root->parentSet = set2_root;
        } else {
            set2_root->parentSet = set1_root;
            set1_root->rango++;
        }
    }
}


void print_forest(const Forest *forest) {
    for (unsigned short i = 0; i < forest->num_sets; i++) {
        Set set = forest->sets[i];

        printf("Set %i: ", set.curr.positionInGraph);
        if (set.parentSet != NULL) {
            printf("Parent: %i, ", set.parentSet->curr.positionInGraph);
        } else {
            printf("Parent: NULL, ");
        }
        printf("Rango: %d, ", set.rango);
        printf("Num in forest: %d\n", set.num_in_forest);

    }
}
```

9.37 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mfset.h File Reference

This file contains the declaration of the Merge-Find Set datastructure for the Minimum Spanning Tree problem.

```
#include "graph.h"
```

Classes

- struct Set

  A Set is a node in the Forest.
- struct Forest

  A Forest is a list of Sets.

Typedefs

- typedef struct Set Set

  A Set is a node in the Forest.
- typedef struct Forest Forest

  A Forest is a list of Sets.

Functions

- void create_forest (Forest ∗forest, const Node ∗nodes, unsigned short num_nodes)

  Create a new Forest with n Sets, each Set containing a Node, without constraints.
- void create_forest_constrained (Forest ∗forest, const Node ∗nodes, unsigned short num_nodes, unsigned short candidateId)

  Create a new Forest with n Sets, each Set containing a Node, with constraints.
- void merge (Set ∗set1, Set ∗set2)

  Merge two Sets in the Forest if they are not already in the same Set.
- Set ∗ find (Set ∗set)

  Find the root of a Set.
- void print_forest (const Forest ∗forest)

  Print all the Forest.

9.37.1 Detailed Description

This file contains the declaration of the Merge-Find Set datastructure for the Minimum Spanning Tree problem.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Repo: `https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound`

Definition in file mfset.h.

9.37.2 Typedef Documentation

9.37.2.1 Forest

```
typedef struct Forest Forest
```

A Forest is a list of Sets.

9.37.2.2 Set

```
typedef struct Set Set
```

A Set is a node in the Forest.

9.37.3 Function Documentation

9.37.3.1 create_forest()

```
void create_forest (
            Forest * forest,
            const Node * nodes,
            unsigned short num_nodes )
```

Create a new Forest with n Sets, each Set containing a Node, without constraints.

Parameters

nodes	Pointer to the List of Nodes.
num_nodes	Number of Nodes in the List.
forest	Pointer to the Forest to be initialized.

Definition at line 31 of file mfset.c.

9.37.3.2 create_forest_constrained()

```
void create_forest_constrained (
            Forest * forest,
            const Node * nodes,
```

```
            unsigned short num_nodes,
            unsigned short candidateId )
```

Create a new Forest with n Sets, each Set containing a Node, with constraints.

The candidateId Node is not added to the Forest because for the 1-tree I need a MST on the remaining Nodes.

Parameters

nodes	Pointer to the List of Nodes.
num_nodes	Number of Nodes in the List.
candidate↩ Id	Id of the Node in the List to be excluded from the Forest.
forest	Pointer to the Forest to be initialized.

Definition at line 17 of file mfset.c.

9.37.3.3 find()

```
Set * find (
            Set * set )
```

Find the root of a Set.

Complexity: O(log n), only a path in the tree is traversed. The parent Set of all the Nodes in the path are updated to point to the root, to reduce the complexity of the next find operations.

Parameters

set	Pointer to the Set.

Returns

Pointer to the root of the Set.

Definition at line 44 of file mfset.c.

9.37.3.4 merge()

```
void merge (
            Set * set1,
            Set * set2 )
```

Merge two Sets in the Forest if they are not already in the same Set.

The Set with the highest rank is the parent of the other. This is done to let the find operation run in O(log n) time. Complexity: O(log n_1 + log n_2)

Parameters

set1	Pointer to the first Set.
set2	Pointer to the second Set.

Definition at line 53 of file mfset.c.

9.37.3.5 print_forest()

```
void print_forest (
            const Forest * forest )
```

Print all the Forest.

Used for debugging purposes.

Parameters

forest	Pointer to the Forest.

Definition at line 72 of file mfset.c.

9.38 mfset.h

Go to the documentation of this file.

```
#ifndef BRANCHANDBOUND1TREE_MFSET_H
#define BRANCHANDBOUND1TREE_MFSET_H
#include "graph.h"


typedef struct Set {
    struct Set * parentSet;
    unsigned short rango;
    Node curr;
    unsigned short num_in_forest;
}Set;


typedef struct Forest {
    unsigned short num_sets;
    Set sets [MAX_VERTEX_NUM];
}Forest;


void create_forest(Forest * forest, const Node * nodes, unsigned short num_nodes);


void create_forest_constrained(Forest * forest, const Node * nodes, unsigned short num_nodes, unsigned
short candidateId);


void merge(Set * set1, Set * set2);


Set* find(Set * set);


void print_forest(const Forest * forest);


#endif //BRANCHANDBOUND1TREE_MFSET_H
```

9.39 GraphConvolutionalBranchandBound/src/HybridSolver/main/data←_structures/mst.c File Reference

This file contains the definition of the Minimum Spanning Tree operations.

```
#include "mst.h"
```

Functions

- void create_mst (MST ∗mst, const Node ∗nodes, unsigned short num_nodes)
  Create a Minimum Spanning Tree from a set of Nodes.
- void add_edge (MST ∗tree, const Edge ∗edge)
  Add an Edge to the MST.
- void print_mst (const MST ∗tree)
  Print the MST, printing all the information it contains.
- void print_mst_original_weight (const MST ∗tree, const Graph ∗graph)
  Print the MST, printing all the information it contains.

9.39.1 Detailed Description

This file contains the definition of the Minimum Spanning Tree operations.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Copyright (c) 2024, license MIT

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file mst.c.

9.39.2 Function Documentation

9.39.2.1 add_edge()

```
void add_edge (
            MST * tree,
            const Edge * edge )
```

Add an Edge to the MST.

Parameters

tree	The Minimum Spanning Tree.
edge	The Edge to add.

Definition at line 38 of file mst.c.

9.39.2.2 create_mst()

```
void create_mst (
            MST * mst,
            const Node * nodes,
            unsigned short num_nodes )
```

Create a Minimum Spanning Tree from a set of Nodes.

Parameters

mst	The Minimum Spanning Tree to be initialized.
nodes	The set of Nodes.
num_nodes	The number of Nodes.

Definition at line 17 of file mst.c.

9.39.2.3 print_mst()

```
void print_mst (
            const MST * mst )
```

Print the MST, printing all the information it contains.

Parameters

tree	The Minimum Spanning Tree.

Definition at line 70 of file mst.c.

9.39.2.4 print_mst_original_weight()

```
void print_mst_original_weight (
            const MST * mst,
            const Graph * graph )
```

Print the MST, printing all the information it contains.

This method is used to print a 1Tree with original. Edge weights, since in the branch and bound algorithm, with the dual procedure the Edge weights are changed.

Parameters

tree	The Minimum Spanning Tree.
graph	The Graph from which the MST was created.

Definition at line 92 of file mst.c.

9.40 mst.c

Go to the documentation of this file.

```
00001
00014 #include "mst.h"
00015
00016
00017 void create_mst(MST * mst, const Node * nodes, unsigned short num_nodes) {
00018     mst->isValid = false;
00019     mst->cost = 0;
00020     mst->num_nodes = num_nodes;
00021     mst->num_edges = 0;
00022     mst->prob = 0;
00023
00024     for (unsigned short i = 0; i < num_nodes; i++) {
00025         mst->nodes[i].positionInGraph = nodes[i].positionInGraph;
00026         mst->nodes[i].x = nodes[i].x;
00027         mst->nodes[i].y = nodes[i].y;
00028         mst->nodes[i].num_neighbours = 0;
00029
00030         for (unsigned short j = i; j < num_nodes; j++) {
00031             mst->edges_matrix[i][j] = -1;
00032             mst->edges_matrix[j][i] = -1;
00033         }
00034     }
00035 }
00036
00037
00038 void add_edge(MST * tree, const Edge * edge){
00039
00040     unsigned short src = edge->src;
00041     unsigned short dest = edge->dest;
00042
00043     tree->edges[tree->num_edges].src = src;
00044     tree->edges[tree->num_edges].dest = dest;
00045     tree->edges[tree->num_edges].weight = edge->weight;
00046     tree->edges[tree->num_edges].symbol = edge->symbol;
00047     tree->edges[tree->num_edges].prob = edge->prob;
00048     tree->edges[tree->num_edges].positionInGraph = tree->num_edges;
00049     tree->nodes[src].neighbours[tree->nodes[src].num_neighbours] = dest;
00050     tree->nodes[src].num_neighbours++;
00051     tree->nodes[dest].neighbours[tree->nodes[dest].num_neighbours] = src;
00052     tree->nodes[dest].num_neighbours++;
00053     tree->edges_matrix[src][dest] = (short) tree->num_edges;
00054     tree->edges_matrix[dest][src] = (short) tree->num_edges;
00055
00056     tree->num_edges++;
00057     tree->cost += edge->weight;
00058
00059     if(HYBRID){
00060         if(tree->num_edges == 1){
00061             tree->prob = edge->prob;
00062         }
00063         else{
00064             tree->prob  = ((tree->prob * ((float) tree->num_edges -1)) + edge->prob) / ((float)
      tree->num_edges);
00065         }
00066     }
00067 }
00068
00069
00070 void print_mst(const MST * tree){
```

```
    printf("\nMST or 1-Tree with cost: %lf and validity = %s\n", tree->cost, tree->isValid ? "TRUE" : "FALSE");

    double dim = (log(tree->num_nodes) / log(10) + 1) * 2 + 7;
    for (unsigned short i = 0; i < tree->num_edges; i++) {
        char edge_print [(int) dim] ;
        char edge_print_dest [(int) (dim-7)/2] ;
        const Edge * curr = &tree->edges[i];
        sprintf(edge_print, "%i", curr->src);
        strcat(edge_print, " <--> ");
        sprintf(edge_print_dest, "%i", curr->dest);
        strcat(edge_print, edge_print_dest);
        printf("Edge%i:\t%s\tweight = %lf\tprob = %lf\n",
               curr->symbol,
               edge_print,
               curr->weight,
               curr->prob);
    }

}


void print_mst_original_weight(const MST * tree, const Graph * graph){
    printf("\nMST or 1-Tree with cost: %f and validity = %s\n", tree->cost, tree->isValid ? "TRUE" : "FALSE");

    double dim = (log(tree->num_nodes) / log(10) + 1) * 2 + 7;
    for (unsigned short i = 0; i < tree->num_edges; i++) {
        char edge_print [(int) dim] ;
        char edge_print_dest [(int) (dim-7)/2] ;
        const Edge * curr = &tree->edges[i];
        sprintf(edge_print, "%i", curr->src);
        strcat(edge_print, " <--> ");
        sprintf(edge_print_dest, "%i", curr->dest);
        strcat(edge_print, edge_print_dest);
        printf("Edge%i: %s weight = %f prob = %f\n",
               curr->symbol,
               edge_print,
               graph->edges_matrix[curr->src][curr->dest].weight,
               curr->prob);
    }
}
```

9.41 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mst.h File Reference

This file contains the declaration of the Minimum Spanning Tree datastructure.

```
#include "mfset.h"
#include "fibonacci_heap.h"
```

Classes

- struct ConstrainedEdge

  A reduced form of an Edge in the Graph, with only the source and destination Nodes.
- struct MST

  Minimum Spanning Tree, or MST, and also a 1-Tree.

Macros

- #define BRANCHANDBOUND1TREE_MST_H

Typedefs

- typedef struct ConstrainedEdge ConstrainedEdge

    A reduced form of an Edge in the Graph, with only the source and destination Nodes.
- typedef struct MST MST

    Minimum Spanning Tree, or MST, and also a 1-Tree.

Functions

- void create_mst (MST ∗mst, const Node ∗nodes, unsigned short num_nodes)

    Create a Minimum Spanning Tree from a set of Nodes.
- void add_edge (MST ∗tree, const Edge ∗edge)

    Add an Edge to the MST.
- void print_mst (const MST ∗mst)

    Print the MST, printing all the information it contains.
- void print_mst_original_weight (const MST ∗mst, const Graph ∗graph)

    Print the MST, printing all the information it contains.

9.41.1 Detailed Description

This file contains the declaration of the Minimum Spanning Tree datastructure.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file mst.h.

9.41.2 Macro Definition Documentation

9.41.2.1 BRANCHANDBOUND1TREE_MST_H

```
#define BRANCHANDBOUND1TREE_MST_H
```

Definition at line 15 of file mst.h.

9.41.3 Typedef Documentation

9.41.3.1 ConstrainedEdge

```
typedef struct ConstrainedEdge ConstrainedEdge
```

A reduced form of an Edge in the Graph, with only the source and destination Nodes.

9.41.3.2 MST

```
typedef struct MST MST
```

Minimum Spanning Tree, or MST, and also a 1-Tree.

9.41.4 Function Documentation

9.41.4.1 add_edge()

```
void add_edge (
            MST * tree,
            const Edge * edge )
```

Add an Edge to the MST.

Parameters

tree	The Minimum Spanning Tree.
edge	The Edge to add.

Definition at line 38 of file mst.c.

9.41.4.2 create_mst()

```
void create_mst (
            MST * mst,
            const Node * nodes,
            unsigned short num_nodes )
```

Create a Minimum Spanning Tree from a set of Nodes.

Parameters

mst	The Minimum Spanning Tree to be initialized.
nodes	The set of Nodes.
num_nodes	The number of Nodes.

Definition at line 17 of file mst.c.

9.41.4.3 print_mst()

```
void print_mst (
            const MST * mst )
```

Print the MST, printing all the information it contains.

Parameters

tree	The Minimum Spanning Tree.

Definition at line 70 of file mst.c.

9.41.4.4 print_mst_original_weight()

```
void print_mst_original_weight (
            const MST * mst,
            const Graph * graph )
```

Print the MST, printing all the information it contains.

This method is used to print a 1Tree with original. Edge weights, since in the branch and bound algorithm, with the dual procedure the Edge weights are changed.

Parameters

tree	The Minimum Spanning Tree.
graph	The Graph from which the MST was created.

Definition at line 92 of file mst.c.

9.42 mst.h

Go to the documentation of this file.

```
00001
```

```
#pragma once
#ifndef BRANCHANDBOUND1TREE_MST_H
#define BRANCHANDBOUND1TREE_MST_H
#include "mfset.h"
#include "fibonacci_heap.h"


typedef struct ConstrainedEdge{
    unsigned short src;
    unsigned short dest;
}ConstrainedEdge;


typedef struct MST{
    bool isValid;
    double cost;
    double prob;
    unsigned short num_nodes;
    unsigned short num_edges;
    Node nodes [MAX_VERTEX_NUM];
    Edge edges [MAX_VERTEX_NUM];
    short edges_matrix [MAX_VERTEX_NUM][MAX_VERTEX_NUM];
}MST;


void create_mst(MST* mst, const Node * nodes, unsigned short num_nodes);


void add_edge(MST * tree, const Edge * edge);


void print_mst(const MST * mst);


void print_mst_original_weight(const MST * mst, const Graph * graph);


#endif //BRANCHANDBOUND1TREE_MST_H
```

9.43 GraphConvolutionalBranchandBound/src/HybridSolver/main/graph-convnet-tsp/main.py File Reference

Namespaces

- namespace main

Functions

- def main.compute_prob (net, config, dtypeLong, dtypeFloat)
- def main.write_adjacency_matrix (graph, y_probs, x_edges_values, nodes_coord, filepath, num_nodes, kmedoids_labels=None)
- def main.add_dummy_cities (num_nodes, model_size)
- def main.create_temp_file (num_nodes, str_grap)
- def main.cluster_nodes (graph, k)
- def main.fix_instance_size (graph, num_nodes, model_size=100)
- def main.get_instance (num_nodes)
- def main.main (filepath, num_nodes, model_size)

Variables

- main.category
- sys main.filepath = sys.argv[1]
- int main.num_nodes = int(sys.argv[2])
- int main.model_size = int(sys.argv[3])

9.44 main.py

Go to the documentation of this file.

```
"""
    @file main.py
    @author Lorenzo Sciandra
    @brief A recombination of code take from: https://github.com/chaitjo/graph-convnet-tsp.
    Some functions were created for the purpose of the paper.
    @version 1.0.0
    @data 2024-05-1
    @copyright Copyright (c) 2024, license MIT
    Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound
"""
import errno
import os
import sys
import time
import numpy as np
import torch
from torch.autograd import Variable
import torch.nn.functional as F
import torch.nn as nn
from sklearn.utils.class_weight import compute_class_weight
# Remove warning
import warnings

warnings.filterwarnings("ignore", category=UserWarning)
from scipy.sparse import SparseEfficiencyWarning

warnings.simplefilter('ignore', SparseEfficiencyWarning)
from config import *
from utils.graph_utils import *
from utils.google_tsp_reader import GoogleTSPReader
from utils.plot_utils import *
from models.gcn_model import ResidualGatedGCNModel
from sklearn_extra.cluster import KMedoids
from utils.model_utils import *


def compute_prob(net, config, dtypeLong, dtypeFloat):
    """
    This function computes the probability of the edges being in the optimal tour, by running the GCN.
    Args:
        net: The Graph Convolutional Network.
        config: The configuration file, from which the parameters are taken.
        dtypeLong: The data type for the long tensors.
        dtypeFloat: The data type for the float tensors.
    Returns:
        y_probs: The probability of the edges being in the optimal tour.
        x_edges_values: The distance between the nodes.
    """
    # Set evaluation mode
    net.eval()

    # Assign parameters
    num_nodes = config.num_nodes
    num_neighbors = config.num_neighbors
    batch_size = config.batch_size
    test_filepath = config.test_filepath

    # Load TSP data
    dataset = GoogleTSPReader(num_nodes, num_neighbors, batch_size=batch_size, filepath=test_filepath)

    # Convert dataset to iterable
    dataset = iter(dataset)

    # Initially set loss class weights as None
    edge_cw = None

    y_probs = []

    # read the instance number line from the test_filepath
    instance = None
    lines = []
    with open(test_filepath, 'r') as f:
        lines = f.readlines()

    if lines is None:
        raise Exception("The input file is empty.")

    instance = lines[0]

    if instance is None:
        raise Exception("The instance does not exist.")

```

```python
    instance = instance.split(" output")[0]
    instance = [float(x) for x in instance.split(" ")]
    # print(config)

    with torch.no_grad():

        batch = next(dataset)

        while batch.nodes_coord.flatten().tolist() != instance:
            batch = next(dataset)

        x_edges = Variable(torch.LongTensor(batch.edges).type(dtypeLong), requires_grad=False)
        x_edges_values = Variable(torch.FloatTensor(batch.edges_values).type(dtypeFloat), requires_grad=False)
        x_nodes = Variable(torch.LongTensor(batch.nodes).type(dtypeLong), requires_grad=False)
        x_nodes_coord = Variable(torch.FloatTensor(batch.nodes_coord).type(dtypeFloat), requires_grad=False)
        y_edges = Variable(torch.LongTensor(batch.edges_target).type(dtypeLong), requires_grad=False)

        # Compute class weights (if uncomputed)
        if type(edge_cw) != torch.Tensor:
            edge_labels = y_edges.cpu().numpy().flatten()
            edge_cw = compute_class_weight("balanced", classes=np.unique(edge_labels), y=edge_labels)

        y_preds, _ = net.forward(x_edges, x_edges_values, x_nodes, x_nodes_coord, y_edges, edge_cw)
        y = F.softmax(y_preds, dim=3)
        # y_bins = y.argmax(dim=3)
        y_probs = y[:, :, :, 1]

    nodes_coord = batch.nodes_coord.flatten().tolist()

    return y_probs, x_edges_values, nodes_coord


def write_adjacency_matrix(graph, y_probs, x_edges_values, nodes_coord, filepath, num_nodes, kmedoids_labels=None):
    """
    This function writes the adjacency matrix to a file.
    The file is in the format:
            cities: (x1, y1);(x2, y2);...;(xn, yn)
            adjacency matrix:
            (0.0, 0.0);(0.23, 0.9);...;(0.15, 0.56)
            ...
            (0.23, 0.9);(0.3, 0.59);...;(0.0, 0.0)
            where each entry is (distance, probability)
    If needed adjusts the size of the graph when the model size is different from the number of nodes in the instance.
    Args:
        graph: The set of nodes in the graph.
        y_probs: The probability of the edges being in the optimal tour.
        x_edges_values: The weight of the edges.
        nodes_coord: The nodes coordinates used in the GCN.
        filepath: The path to the file where the adjacency matrix will be written.
        num_nodes: The number of nodes in the TSP instance.
        kmedoids_labels: The labels of the k-medoids clustering.
    """

    model_size = y_probs.shape[1]
    y_probs = y_probs.flatten().numpy()
    x_edges_values = x_edges_values.flatten().numpy()

    # stack the arrays horizontally and convert to string data type
    arr_combined = np.stack((x_edges_values, y_probs), axis=1).astype('U')

    if num_nodes < model_size:
        nodes_coord = nodes_coord[:num_nodes*2]
        final_arr = []
        for i in range(num_nodes):
            for j in range(num_nodes):
                final_arr.append(arr_combined[i * model_size + j])

            for j in range(num_nodes, model_size):
                if i != j-num_nodes:
                    if arr_combined[i * model_size + j][1] > final_arr[i * num_nodes + j - num_nodes][1]:
                        final_arr[i * num_nodes + j - num_nodes][1] = arr_combined[i * model_size + j][1]

        arr_combined = np.array(final_arr)

    elif num_nodes > model_size:
        nodes_coord = [[nodes_coord[i], nodes_coord[i + 1]] for i in range(0, len(nodes_coord), 2)]
        arr_combined = arr_combined.flatten()
        arr_combined = arr_combined.reshape(model_size, model_size, 2).tolist()
        j = 0
        for node in graph:
            if node not in nodes_coord:
```

```
                new_row = []
                label = kmedoids_labels[j]

                for i in range(len(nodes_coord)):
                    distance = np.linalg.norm(np.array(node) - np.array(nodes_coord[i]))
                    prob = 0.0 #arr_combined[label][i][1]
                    arr_combined[i].append([distance, prob])
                    new_row.append([distance, prob])

                new_row.append([0.0, 0.0])
                arr_combined.append(new_row)
                nodes_coord.append(node)

            j += 1

        arr_combined = np.array(arr_combined).flatten().tolist()
        arr_combined = [[arr_combined[i], arr_combined[i + 1]] for i in range(0, len(arr_combined), 2)]
        nodes_coord = np.array(nodes_coord).flatten().tolist()

    nodes_coord = ";".join([f"({nodes_coord[i]}, {nodes_coord[i + 1]})" for i in range(0, len(nodes_coord), 2)])
    arr_strings = np.array(['({}, {});'.format(x[0], x[1]) for x in arr_combined])

    with open(filepath, 'w') as f:
        f.write("%s\n" % nodes_coord)
        edge = 0
        for item in arr_strings:
            if (edge + 1) % num_nodes == 0:
                f.write("%s\n" % item)
            else:
                f.write("%s" % item)
            edge += 1
        f.flush()
        os.fsync(f.fileno())


def add_dummy_cities(num_nodes, model_size):
    """
    This function adds dummy cities to the graph instance. The dummy cities are randomly generated are
    added to the graph instance and the new instance is saved in a temporary file.
    Args:
        num_nodes: The number of nodes of the graph instance.
        model_size: The size of the Graph Convolutional Network to use.
    """

    num_dummy_cities = model_size - num_nodes
    filepath = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"
    graph_str = None

    with open(filepath, "r") as f:
        lines = f.readlines()
        graph_str = lines[0]

    graph_str = graph_str.split(" output")[0]
    if graph_str is None:
        raise Exception("The input file is empty.")

    nodes = graph_str.split(" ")
    graph = [[float(nodes[i]), float(nodes[i + 1])] for i in range(0, len(nodes), 2)]
    rr_index = 0
    for i in range(num_dummy_cities):
        find = False
        x = None
        y = None
        while not find:
            values = np.random.randint(1, 9, 2)
            signs = np.random.choice([-1, 1], 2)
            x = graph[rr_index][0] + signs[0] * values[0] * 0.000000000000001
            y = graph[rr_index][1] + signs[1] * values[1] * 0.000000000000001
            if [x, y] not in graph:
                find = True

        graph.append([x, y])
        graph_str += " " + str(x) + " " + str(y)
        rr_index = (rr_index + 1) % num_nodes

    seq = np.linspace(1,model_size, model_size, dtype=int)
    seq_str = ""
    for s in seq:
        seq_str += str(s) + " "

    seq_str += "1"
    graph_str += " output " + seq_str

    with open(filepath, 'w+') as file:
        file.writelines(graph_str)
```

```
        file.flush()
        os.fsync(file.fileno())


def create_temp_file(num_nodes, str_grap):
    """
    Creates a temporary file with the graph instance.
    Args:
        num_nodes: The number of nodes of the graph instance.
        str_grap: The graph instance.
    """

    filepath = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"

    with open(filepath, 'w+') as file:
        file.writelines(str_grap)
        file.flush()
        os.fsync(file.fileno())


def cluster_nodes(graph, k):
    """
    Applies the k-medoids clustering to the graph.
    Args:
        graph: The graph to cluster.
        k: The number of clusters to create.
    Returns:
        medoids_str: The medoids of the clusters.
        kmedoids.labels_: The labels of the clusters.
    """
    graph = np.array(graph)
    kmedoids = KMedoids(n_clusters=k, method='pam', random_state=42).fit(graph)
    medoids = kmedoids.cluster_centers_
    medoids_str = " ".join(f"{x} {y}" for x, y in medoids)

    return medoids_str, kmedoids.labels_


def fix_instance_size(graph, num_nodes, model_size=100):
    """
    The function that fixes the instance size with clustering.
    It applies the k-medoids clustering to the graph and creates a new instance with the medoids as the new nodes.
    Args:
        graph: The graph to fix.
        num_nodes: The number of nodes of the graph instance.
        model_size: The size of the Graph Convolutional Network to use.
    """

    new_graph_str = ""
    end_str = " output "

    print("Need to fix the instance size with clustering")
    new_graph_str, kmedoids_labels = cluster_nodes(graph, model_size)

    for i in range(1, model_size + 1):
        end_str += str(i) + " "

    end_str += "1"

    create_temp_file(num_nodes, new_graph_str + end_str)
    return kmedoids_labels


def get_instance(num_nodes):
    """
    The function that reads the current instance from the file.
    Args:
        num_nodes: The number of nodes of the graph instance.
    Returns:
        graph: The graph instance.
    """

    lines = None
    file_path = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"

    with open(file_path, "r") as f:
        lines = f.readlines()

    if lines is None or len(lines) == 0:
        raise Exception(
            "The current instance for the number of nodes " + str(num_nodes) + " does not exist.")

    str_graph = lines[0]

    if "output" in str_graph:
        str_graph = str_graph.split(" output")[0]
```

```
    str_graph = str_graph.replace("\n", "").strip()
    nodes = str_graph.split(" ")
    graph = [float(x) for x in nodes]
    graph = [[graph[i], graph[i + 1]] for i in range(0, len(graph), 2)]

    return graph



def main(filepath, num_nodes, model_size):
    """
    The function that runs the Graph Convolutional Network and writes the adjacency matrix to a file
    for the given input instance.
    Args:
        filepath: The path to the file where the adjacency matrix will be written.
        num_nodes: The number of nodes in the TSP instance.
        model_size: The size of the Graph Convolutional Network to use.
    """

    graph = None
    kmedoids_labels = None

    if num_nodes < model_size:
        add_dummy_cities(num_nodes, model_size)
    elif num_nodes > model_size:
        graph = get_instance(num_nodes)
        kmedoids_labels = fix_instance_size(graph, num_nodes)

    config_path = "./logs/tsp" + str(model_size) + "/config.json"
    config = get_config(config_path)

    config.gpu_id = "0"
    config.accumulation_steps = 1

    config.val_filepath = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"
    config.test_filepath = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"

    os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
    os.environ["CUDA_VISIBLE_DEVICES"] = str(config.gpu_id)

    if torch.cuda.is_available():
        # print("CUDA available, using GPU ID {}".format(config.gpu_id))
        dtypeFloat = torch.cuda.FloatTensor
        dtypeLong = torch.cuda.LongTensor
        torch.cuda.manual_seed(1)
    else:
        # print("CUDA not available")
        dtypeFloat = torch.FloatTensor
        dtypeLong = torch.LongTensor
        torch.manual_seed(1)

    net = nn.DataParallel(ResidualGatedGCNModel(config, dtypeFloat, dtypeLong))
    if torch.cuda.is_available():
        net.cuda()

    log_dir = f"./logs/{config.expt_name}/"
    if torch.cuda.is_available():
        checkpoint = torch.load(log_dir + "best_val_checkpoint.tar")
    else:
        checkpoint = torch.load(log_dir + "best_val_checkpoint.tar", map_location='cpu')
    # Load network state
    net.load_state_dict(checkpoint['model_state_dict'])
    config.batch_size = 1
    probs, edges_value, nodes_coord = compute_prob(net, config, dtypeLong, dtypeFloat)
    write_adjacency_matrix(graph, probs, edges_value, nodes_coord, filepath, num_nodes,
kmedoids_labels)


if __name__ == "__main__":
    """
    Args:
        sys.argv[1]: The path to the file where the adjacency matrix will be written.
        sys.argv[2]: The number of nodes in the TSP instance.
        sys.argv[3]: The dimension of the model.
    """

    if len(sys.argv) != 4:
        print("\nPlease provide the path to the output file to write in, the number of nodes in the
tsp and the "
              "instance number to analyze. The format is: "
              "<filepath> <number of nodes> <model size>\n")
        sys.exit(1)

    if not isinstance(sys.argv[1], str) or not isinstance(sys.argv[2], str) or not
isinstance(sys.argv[3], str):
        print("Error: The arguments must be strings.")
```

```
        sys.exit(1)

    filepath = sys.argv[1]
    num_nodes = int(sys.argv[2])
    model_size = int(sys.argv[3])
    main(filepath, num_nodes, model_size)
```

9.45 GraphConvolutionalBranchandBound/README.md File Reference

9.46 GraphConvolutionalBranchandBound/src/Hybrid↩Solver/main/graph-convnet-tsp/README.md File Reference

9.47 GraphConvolutionalBranchandBound/src/HybridSolver/main/↩HybridSolver.py File Reference

Namespaces

- namespace HybridSolver

Functions

- def HybridSolver.adjacency_matrix (orig_graph)
- def HybridSolver.create_temp_file (num_nodes, str_grap)
- def HybridSolver.get_nodes (graph)
- def HybridSolver.get_instance (instance, num_nodes)
- def HybridSolver.build_c_program (build_directory, num_nodes, hyb_mode)
- def HybridSolver.hybrid_solver (num_instances, num_nodes, hyb_mode, gen_matrix)

Variables

- argparse HybridSolver.parser = argparse.ArgumentParser()
- HybridSolver.type
- HybridSolver.str
- HybridSolver.default
- HybridSolver.int
- HybridSolver.action
- argparse HybridSolver.opts = parser.parse_args()
- argparse HybridSolver.gen_matrix = False

9.48 HybridSolver.py

Go to the documentation of this file.

```
"""
    @file: HybridSolver.py
    @author Lorenzo Sciandra
    @brief First it builds the program in C, specifying the number of nodes to use and whether it is in hybrid mode or not.
    Then it runs the graph conv net on the instance, and finally it runs the Branch and Bound.
    It can be run on a single instance or a range of instances.
    The input matrix is generated by the neural network and stored in the data folder. The output is stored in the results folder.
    @version 1.0.0
    @data 2024-05-1
    @copyright Copyright (c) 2024, license MIT

    Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound
"""
import subprocess
import argparse
import pprint as pp
import os
import time
import numpy as np


def adjacency_matrix(orig_graph):
    """
    Calculates the adjacency matrix of the graph.
    Args:
        orig_graph: The original graph.

    Returns:
        The adjacency matrix of the graph.
    """
    adj_matrix = np.zeros((len(orig_graph), len(orig_graph)))

    for i in range(0, len(orig_graph)):
        for j in range(i + 1, len(orig_graph)):
            adj_matrix[i][j] = np.linalg.norm(np.array(orig_graph[i]) - np.array(orig_graph[j]))
            adj_matrix[j][i] = adj_matrix[i][j]

    return adj_matrix


def create_temp_file(num_nodes, str_grap):
    """
    Creates a temporary file to store the current instance of the TSP for the neural network.
    Args:
        num_nodes: The number of nodes in the TSP instance.
        str_grap: The string representation of the graph.
    """
    filepath = "graph-convnet-tsp/data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"

    if not os.path.exists(os.path.dirname(filepath)):
        try:
            os.makedirs(os.path.dirname(filepath))
        except OSError as exc:  # Guard against race condition
            if exc.errno != errno.EEXIST:
                raise

    with open(filepath, 'w+') as file:
        file.writelines(str_grap)
        file.flush()
        os.fsync(file.fileno())


def get_nodes(graph):
    """
    From a graph, it returns the nodes in a string format.
    Args:
        graph: The graph to get the nodes from.

    Returns:
        The nodes in a string format.
    """
    nodes = ""
    i = 0
    for node in graph:
        nodes += "\t" + str(i) + " : " + str(node[0]) + " " + str(node[1]) + "\n"
        i += 1

    return nodes


```

```
def get_instance(instance, num_nodes):
    """
    Gets the instance of the TSP from the file.
    Args:
        instance: The instance to get.
        num_nodes: The number of nodes in the TSP instance.

    Returns:
        The graph in a list and string format.
    """

    lines = None
    file_path = "graph-convnet-tsp/data/hyb_tsp/test_" + str(num_nodes) + "_nodes.txt"
    with open(file_path, "r") as f:
        lines = f.readlines()

    if lines is None or len(lines) < instance - 1:
        raise Exception(
            "The instance " + str(instance) + " for the number of nodes " + str(num_nodes) + " does
not exist.")

    str_graph = lines[instance - 1]
    orig_graph = str_graph

    if "output" in str_graph:
        str_graph = str_graph.split(" output")[0]

    str_graph = str_graph.replace("\n", "").strip()
    nodes = str_graph.split(" ")
    graph = [float(x) for x in nodes]
    graph = [[graph[i], graph[i + 1]] for i in range(0, len(graph), 2)]

    return graph, orig_graph


def build_c_program(build_directory, num_nodes, hyb_mode):
    """
    Builds the C program with the specified number of nodes and whether it is in hybrid mode or not.
    Args:
        build_directory: The directory where the CMakeLists.txt file is located and where the
executable will be built.
        num_nodes: The number of nodes to use in the C program.
        hyb_mode: 1 if the program is in hybrid mode, 0 otherwise.
    """
    source_directory = "../"
    cmake_command = [
        "cmake",
        "-S" + source_directory,
        "-B" + build_directory,
        "-DCMAKE_BUILD_TYPE=Release",
        "-DMAX_VERTEX_NUM=" + str(num_nodes),
        "-DHYBRID=" + str(hyb_mode)
    ]
    # print(cmake_command)
    make_command = [
        "make",
        "-C" + build_directory,
        "-j"
    ]
    try:
        subprocess.check_call(cmake_command)
        subprocess.check_call(make_command)
    except subprocess.CalledProcessError as e:
        print("Build failed:")
        print(e.output)
        raise Exception("Build failed")


def hybrid_solver(num_instances, num_nodes, hyb_mode, gen_matrix):
    """
    The Graph Convolutional Branch-and-Bound Solver.
    Args:
        num_instances: The range of instances to run on the Solver.
        num_nodes: The number of nodes in each TSP instance.
        hyb_mode: True if the program is in hybrid mode, False otherwise.
        gen_matrix: True if the adjacency matrix is already generated, False otherwise.
    """

    model_size = 0
    adj_matrix = None

    if hyb_mode:
        if num_nodes <= 1:
            raise Exception("The number of nodes must be greater than 1.")
        elif num_nodes <= 20:
            model_size = 20
        elif num_nodes <= 50:
```

```
            model_size = 50
        else:
            model_size = 100
    else:
        model_size = num_nodes

    build_directory = "../cmake-build/CMakeFiles/BranchAndBound1Tree.dir"
    hybrid = 1 if hyb_mode else 0
    build_c_program(build_directory, num_nodes, hybrid)

    if "-" in num_instances:
        instances = num_instances.split("-")
        start_instance = 1 if int(instances[0]) == 0 else int(instances[0])
        end_instance = int(instances[1])
    else:
        start_instance = 1
        end_instance = int(num_instances)

    print("Starting instance: " + str(start_instance))
    print("Ending instance: " + str(end_instance))

    for i in range(start_instance, end_instance + 1):
        start_time = time.time()
        graph, str_graph = get_instance(i, num_nodes)
        input_file = "../data/AdjacencyMatrix/tsp_" + str(num_nodes) + "_nodes/tsp_test_" + str(i) + ".csv"
        absolute_input_path = os.path.abspath(input_file)

        if not os.path.exists(os.path.dirname(absolute_input_path)):
            try:
                os.makedirs(os.path.dirname(absolute_input_path))
            except OSError as exc:  # Guard against race condition
                if exc.errno != errno.EEXIST:
                    raise

        result_mode = "hybrid" if hyb_mode else "classic"
        output_file = "../results/AdjacencyMatrix/tsp_" + str(num_nodes) + "_nodes_" + result_mode \
                      + "/tsp_result_" + str(i) + ".txt"

        if hyb_mode:
            create_temp_file(num_nodes, str_graph)
            absolute_python_path = os.path.abspath("./graph-convnet-tsp/main.py")
            result = subprocess.run(
                ['python3', absolute_python_path, absolute_input_path, str(num_nodes), str(model_size)],
                cwd="./graph-convnet-tsp", check=True)
            if result.returncode == 0:
                print('Neural Network completed successfully on instance ' + str(i) + ' / ' + str(end_instance))
            else:
                print('Neural Network failed on instance ' + str(i) + ' / ' + str(end_instance))

        elif gen_matrix:
            adj_matrix = adjacency_matrix(graph)
            with open(absolute_input_path, "w") as f:
                nodes_coord = ";".join([f"({graph[i][0]}, {graph[i][1]})" for i in range(len(graph))])
                f.write(nodes_coord + "\n")
                for k in range(len(adj_matrix)):
                    for j in range(len(adj_matrix[k])):
                        f.write(f"({adj_matrix[k][j]}, 0);")
                    f.write("\n")

        absolute_output_path = os.path.abspath(output_file)
        if not os.path.exists(os.path.dirname(absolute_output_path)):
            try:
                os.makedirs(os.path.dirname(absolute_output_path))
            except OSError as exc:  # Guard against race condition
                if exc.errno != errno.EEXIST:
                    raise
        cmd = [build_directory + "/BranchAndBound1Tree", absolute_input_path, absolute_output_path]
        result = subprocess.run(cmd)
        if result.returncode == 0:
            print('Branch-and-Bound completed successfully on instance ' + str(i) + ' / ' + str(end_instance))
        else:
            print('Branch-and-Bound failed on instance ' + str(i) + ' / ' + str(end_instance))

        end_time = time.time()
        cities = get_nodes(graph)

        if hyb_mode:
            os.remove("graph-convnet-tsp/data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt")

        with open(output_file, "a") as f:
            f.write("\nNodes: \n" + cities)
            f.write("\nTime taken: " + str(end_time - start_time) + "s\n")
            f.flush()
```

```
            os.fsync(f.fileno())


if __name__ == "__main__":
    """
    Args:
        --range_instances: The range of instances to run on the Solver.
        --num_nodes: The number of nodes in each TSP instance.
        --hybrid_mode: If present, the program is in hybrid mode, otherwise it is in classic mode.
    """

    parser = argparse.ArgumentParser()
    parser.add_argument("--range_instances", type=str, default="1-1")
    parser.add_argument("--num_nodes", type=int, default=20)
    parser.add_argument("--hybrid", action="store_true")
    opts = parser.parse_args()

    pp.pprint(vars(opts))

    gen_matrix = opts.hybrid == False

    hybrid_solver(opts.range_instances, opts.num_nodes, opts.hybrid, gen_matrix)
```

9.49 GraphConvolutionalBranchandBound/src/Hybrid↩ Solver/main/main.c File Reference

Project main file, where you start the program, read the input file and print/write the results.

```
#include "../test/main_test.h"
```

Functions

- int main (int argc, char ∗argv[ ])

  Main function, where you start the program, read the input file and print/write the results.

9.49.1 Detailed Description

Project main file, where you start the program, read the input file and print/write the results.

Author

Lorenzo Sciandra

Version

1.0.0 @data 2024-05-1

Copyright

Copyright (c) 2024, license MIT

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file main.c.

9.49.2 Function Documentation

9.49.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Main function, where you start the program, read the input file and print/write the results.

Parameters

argc	The number of arguments passed to the program.
argv	The arguments passed to the program.

Returns

0 if the program ends correctly, 1 otherwise.

Definition at line 23 of file main.c.

9.50 main.c

Go to the documentation of this file.

```
00001
00014 #include "../test/main_test.h"
00015
00016
00023 int main(int argc, char *argv[]) {
00024
00025     if (argc != 3) {
00026         perror("Wrong number of arguments");
00027         printf("\nYou need to pass 2 arguments: <input file> <output file>\n");
00028         exit(1);
00029     }
00030
00031
00032     char *input_file = argv[1];
00033     char *output_file = argv[2];
00034
00035     //printf("\nNodes:%d \t\tMode: %s\t\t\n", MAX_VERTEX_NUM, HYBRID ? "Hybrid" : "Classic");
00036
00037     //printf("\nReading from file '%s'\n", input_file);
00038     //printf("\nWriting to file '%s'\n", output_file);
00039
00040     freopen(output_file, "w+", stdout);
00041
00042     //run_all_tests();
00043
00044     static Problem new_problem;
00045
00046     //read_tsp_lib_file(&new_problem.graph, input_file);
00047     read_tsp_csv_file(&new_problem.graph, input_file);
00048
00049     //print_graph(&new_problem.graph);
00050
00051     branch_and_bound(&new_problem);
00052
00053     print_problem();
00054
00055     fclose(stdout);
00056
00057     return 0;
00058 }
```

9.51 GraphConvolutionalBranchandBound/src/Hybrid↩ Solver/main/problem_settings.h File Reference

Contains all the execution settings.

```
#include <stdio.h>
#include <limits.h>
#include <float.h>
#include <string.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <errno.h>
#include <pthread.h>
```

Macros

- #define INFINITE DBL_MAX

  The maximum number to set the initial value of Problem and SubProblem.
- #define PRIM 1

  The maximum number of Node in the Graph.
- #define MAX_EDGES_NUM (MAX_VERTEX_NUM ∗ (MAX_VERTEX_NUM - 1) / 2)

  The maximum number of edges in the Graph.
- #define GHOSH_UB (sqrt(MAX_VERTEX_NUM) ∗ 1.27f)

  The first upper bound for the problem, see: `https://www.semanticscholar.org/paper/↩ Expected-Travel-Among-Random-Points-in-a-Region-Ghosh/4c395ab42054f4312ad24cb500fb8ca6f7ad`
- #define TRACE() fprintf(stderr, "%s (%d): %s\n", __FILE__, __LINE__, __func__)

  Used to debug the code, to check if the execution reaches a certain point.
- #define EPSILON (GHOSH_UB / 1000)

  The first constant used to compare two SubProblem in the branch and bound algorithm.
- #define EPSILON2 (0.1f ∗ EPSILON)

  The second constant used to compare two SubProblem in the branch and bound algorithm.
- #define BETTER_PROB 0.05f

  The third constant used to compare two SubProblem in the branch and bound algorithm.
- #define PROB_BRANCH 0.5f

  The way with generate the children of a SubProblem.
- #define TIME_LIMIT_SECONDS 600

  The maximum time to run the algorithm. Default: 10 minutes.
- #define NUM_HK_INITIAL_ITERATIONS (((((float) MAX_VERTEX_NUM ∗ MAX_VERTEX_NUM)/50) + 0.5f) + MAX_VERTEX_NUM + 15)

  The maximum number of dual iterations for the root of the branch and bound tree.
- #define NUM_HK_ITERATIONS (((float) MAX_VERTEX_NUM / 4) + 5)

  The maximum number of dual iterations for nodes of the branch and bound tree that are not the root.

9.51.1 Detailed Description

Contains all the execution settings.

Author

Lorenzo Sciandra

Not only MACROs for branch-and-bound, but also for testing and debugging. The two MACROs MAX_VERTEX_↩ NUM and HYBRID that are used to set the maximum number of Node in the Graph and to choose the algorithm to use are now in the CMakeLists.txt file, so that they can be changed from the command line.

Version

1.0.0 @data 2024-05-1

Copyright

Repo: `https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound`

Definition in file problem_settings.h.

9.51.2 Macro Definition Documentation

9.51.2.1 BETTER_PROB

```
#define BETTER_PROB 0.05f
```

The third constant used to compare two SubProblem in the branch and bound algorithm.

If two SubProblem are within EPSILON2 (and therefore equal), the one that has a greater probability than the other of at least BETTER_PROB is considered better.

See also

branch_and_bound.c::compare_subproblems()

Definition at line 82 of file problem_settings.h.

9.51.2.2 EPSILON

```
#define EPSILON (GHOSH_UB / 1000)
```

The first constant used to compare two SubProblem in the branch and bound algorithm.

Two SubProblem are considered equal if their lower bound is within EPSILON of each other.

See also

branch_and_bound.c::compare_subproblems()

Definition at line 65 of file problem_settings.h.

9.51.2.3 EPSILON2

```
#define EPSILON2 (0.1f * EPSILON)
```

The second constant used to compare two SubProblem in the branch and bound algorithm.

If two SubProblem are equal and their lower bound is within EPSILON2 of each other, their probability is compared.

See also

branch_and_bound.c::compare_subproblems()

Definition at line 73 of file problem_settings.h.

9.51.2.4 GHOSH_UB

```
#define GHOSH_UB (sqrt(MAX_VERTEX_NUM) * 1.27f)
```

The first upper bound for the problem, see: https://www.semanticscholar.org/paper/↩
Expected-Travel-Among-Random-Points-in-a-Region-Ghosh/4c395ab42054f4312ad24cb500fb8ca6f7

Definition at line 53 of file problem_settings.h.

9.51.2.5 INFINITE

```
#define INFINITE DBL_MAX
```

The maximum number to set the initial value of Problem and SubProblem.

Definition at line 33 of file problem_settings.h.

9.51.2.6 MAX_EDGES_NUM

```
#define MAX_EDGES_NUM (MAX_VERTEX_NUM * (MAX_VERTEX_NUM - 1) / 2)
```

The maximum number of edges in the Graph.

Definition at line 49 of file problem_settings.h.

9.51.2.7 NUM_HK_INITIAL_ITERATIONS

```
#define NUM_HK_INITIAL_ITERATIONS (((((float) MAX_VERTEX_NUM * MAX_VERTEX_NUM)/50) + 0.5f) +
MAX_VERTEX_NUM + 15)
```

The maximum number of dual iterations for the root of the branch and bound tree.

Definition at line 100 of file problem_settings.h.

9.51.2.8 NUM_HK_ITERATIONS

```
#define NUM_HK_ITERATIONS (((float) MAX_VERTEX_NUM / 4) + 5)
```

The maximum number of dual iterations for nodes of the branch and bound tree that are not the root.

Definition at line 104 of file problem_settings.h.

9.51.2.9 PRIM

```
#define PRIM 1
```

The maximum number of Node in the Graph.

The parameter to choose the algorithm to use, 0 for Classic B&B, 1 for Hybrid B&B. 1 if the MST is solved with the Prim algorithm using Fibonacci Heap, 0 if it is solved with the Kruskal algorithm using MFSets.

Definition at line 45 of file problem_settings.h.

9.51.2.10 PROB_BRANCH

```
#define PROB_BRANCH 0.5f
```

The way with generate the children of a SubProblem.

If the probability of a SubProblem is greater than PROB_BRANCH, the children are generated by first imposing the constraint that the edge (i, j) is in the solution and then the constraint that it is not. Otherwise, the children are generated by imposing the constraint that the edge (i, j) is not in the solution and then the constraint that it is.

See also

branch_and_bound.c::branch()

Definition at line 92 of file problem_settings.h.

9.51.2.11 TIME_LIMIT_SECONDS

```
#define TIME_LIMIT_SECONDS 600
```

The maximum time to run the algorithm. Default: 10 minutes.

Definition at line 96 of file problem_settings.h.

9.51.2.12 TRACE

```
#define TRACE( ) fprintf(stderr, "%s (%d):  %s\n", __FILE__, __LINE__, __func__)
```

Used to debug the code, to check if the execution reaches a certain point.

Definition at line 57 of file problem_settings.h.

9.52 problem_settings.h

Go to the documentation of this file.

```c

#ifndef BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
#define BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
#include <stdio.h>
#include <limits.h>
#include <float.h>
#include <string.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <errno.h>
#include <pthread.h>


#define INFINITE DBL_MAX


//#define MAX_VERTEX_NUM 20//-- no longer in this file, but in the CMakeLists.txt to be able to change it from the command line.


//#define HYBRID 0 //-- no longer in this file, but in the CMakeLists.txt to be able to change it from the command line.


#define PRIM 1


#define MAX_EDGES_NUM (MAX_VERTEX_NUM * (MAX_VERTEX_NUM - 1) / 2)


#define GHOSH_UB (sqrt(MAX_VERTEX_NUM) * 1.27f)


#define TRACE() fprintf(stderr, "%s (%d): %s\n", __FILE__, __LINE__, __func__)



#define EPSILON (GHOSH_UB / 1000)



#define EPSILON2 (0.1f * EPSILON)



#define BETTER_PROB 0.05f



#define PROB_BRANCH 0.5f


#define TIME_LIMIT_SECONDS 600


#define NUM_HK_INITIAL_ITERATIONS ((((float) MAX_VERTEX_NUM * MAX_VERTEX_NUM)/50) + 0.5f) + MAX_VERTEX_NUM + 15)


#define NUM_HK_ITERATIONS (((float) MAX_VERTEX_NUM / 4) + 5)


#endif //BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
```

9.53 GraphConvolutionalBranchandBound/src/HybridSolver/main/ReadME.md File Reference

9.54 GraphConvolutionalBranchandBound/src/HybridSolver/main/tsp_instance_reader.c File Reference

The definition of the function to read input files.

```
#include "tsp_instance_reader.h"
```

Functions

- void read_tsp_lib_file (Graph *graph, char *filename)
  Reads a .tsp file and stores the data in the Graph.
- void read_tsp_csv_file (Graph *graph, char *filename)
  Reads a .csv file and stores the data in the Graph.

9.54.1 Detailed Description

The definition of the function to read input files.

Author

Lorenzo Sciandra

There are two functions to read the input files, one for the .tsp format and one for the .csv format.

Version

1.0.0 @data 2024-05-1

Copyright

Copyright (c) 2024, license MIT

Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound

Definition in file tsp_instance_reader.c.

9.54.2 Function Documentation

9.54.2.1 read_tsp_csv_file()

```
void read_tsp_csv_file (
            Graph * graph,
            char * filename )
```

Reads a .csv file and stores the data in the Graph.

Parameters

graph	The Graph where the data will be stored.
filename	The name of the file to read.

Definition at line 79 of file tsp_instance_reader.c.

9.54.2.2 read_tsp_lib_file()

```
void read_tsp_lib_file (
            Graph * graph,
            char * filename )
```

Reads a .tsp file and stores the data in the Graph.

Parameters

graph	The Graph where the data will be stored.
filename	The name of the file to read.

Definition at line 18 of file tsp_instance_reader.c.

9.55 tsp_instance_reader.c

Go to the documentation of this file.

```
00001
00015 #include "tsp_instance_reader.h"
00016
00017
00018 void read_tsp_lib_file(Graph *graph, char *filename) {
00019     FILE *fp = fopen(filename, "r");
00020     if (fp == NULL) {
00021         perror("Error while opening the file.\n");
00022         printf("\nFile: %s\n", filename);
00023         exit(1);
00024     }
00025
00026     char *line = NULL;
00027     size_t len = 0;
00028     bool check_euc_2d = false;
00029     while (getline(&line, &len, fp) != -1 &&
00030            strstr(line, "NODE_COORD_SECTION") == NULL) {
00031         if (strstr(line, "EDGE_WEIGHT_TYPE : EUC_2D") == NULL) {
00032             check_euc_2d = true;
00033         }
00034     }
00035
00036     if (!check_euc_2d) {
00037         perror("The current TSP file is not an euclidean one.\n");
00038         printf("\nFile: %s\n", filename);
00039         exit(1);
00040     }
00041
00042     unsigned short i = 0;
00043     Node nodes[MAX_VERTEX_NUM];
00044     graph->kind = WEIGHTED_GRAPH;
00045     List *nodes_list = new_list();
00046     bool end_of_file = false;
00047     while (getline(&line, &len, fp) != -1 && !end_of_file) {
00048         if (strstr(line, "EOF") == NULL) {
00049             unsigned short id;
00050             float x;
00051             float y;
00052
00053             int result = sscanf(line, "%hu %f %f", &id, &x, &y);
00054             if (result != 3) {
00055                 perror("Error while reading the file.\n");
00056                 printf("\nFile: %s\n", filename);
00057                 exit(1);
00058             }
```

```
            nodes[i].positionInGraph = i;
            nodes[i].x = x;
            nodes[i].y = y;
            nodes[i].num_neighbours = 0;
            add_elem_list_bottom(nodes_list, &nodes[i]);
            i++;
        } else {
            end_of_file = true;
        }
    }
    free(line);
    if (fclose(fp) == EOF) {
        perror("Error while closing the file.\n");
        printf("\nFile: %s\n", filename);
        exit(1);
    }
    create_euclidean_graph(graph, nodes_list);
}


void read_tsp_csv_file(Graph *graph, char *filename) {
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        perror("Error while opening the file.\n");
        printf("\nFile: %s\n", filename);
        exit(1);
    }
    graph->cost = 0;
    graph->num_edges = 0;
    graph->num_nodes = 0;
    graph->kind = WEIGHTED_GRAPH;
    graph->orderedEdges = false;
    unsigned short i = 0;
    unsigned short z = 0;
    char *line = NULL;
    size_t len = 0;
    bool first = true;
    while (getline(&line, &len, fp) != -1) {
        if (first) {
            first = false;

            char *token = strtok(line, ";");
            unsigned short node_num = 0;
            while (token != NULL && strcmp(token, "\n") != 0) {
                double x = 0, y = 0;
                int result = sscanf(token, "(%lf, %lf)", &x, &y);
                if (result != 2) {
                    perror("Error while reading the file.\n");
                    printf("\nFile: %s\n", filename);
                    exit(1);
                }
                graph->nodes[node_num].positionInGraph = node_num;
                graph->nodes[node_num].x = x;
                graph->nodes[node_num].y = y;
                graph->nodes[node_num].num_neighbours = 0;
                node_num++;
                token = strtok(NULL, ";");
            }

            continue;
        }
        char *token = strtok(line, ";");
        unsigned short j = 0;
        while (token != NULL && strcmp(token, "\n") != 0) {
            if (j != i) {
                double weight = 0, prob = 0;

                int result = sscanf(token, "(%lf, %lf)", &weight, &prob);
                if (result != 2) {
                    perror("Error while reading the file.\n");
                    printf("\nFile: %s\n", filename);
                    exit(1);
                }

                if (weight > 0) {
                    if (j > i) {
                        graph->nodes[i].neighbours[graph->nodes[i].num_neighbours] = j;
                        graph->nodes[i].num_neighbours++;
                        graph->num_edges++;
                        graph->edges[z].src = i;
                        graph->edges[z].dest = j;
                        graph->edges[z].prob = HYBRID ? prob : 0;
                        graph->edges[z].symbol = z + 1;
                        graph->edges[z].positionInGraph = z;
                        graph->edges[z].weight = weight;
                        graph->cost += graph->edges[z].weight;
                        graph->nodes[j].positionInGraph = j;
```

```
                        graph->edges_matrix[i][j] = graph->edges[z];
                        graph->edges_matrix[j][i] = graph->edges[z];
                        z++;
                    } else {
                        graph->nodes[i].neighbours[graph->nodes[i].num_neighbours] = j;
                        graph->nodes[i].num_neighbours++;
                    }
                }
            }
            token = strtok(NULL, ";");
            j++;
        }
        graph->num_nodes++;
        i++;
    }
    free(line);
    if (fclose(fp) == EOF) {
        perror("Error while closing the file.\n");
        printf("\nFile: %s\n", filename);
        exit(1);
    }
}
```

9.56 GraphConvolutionalBranchandBound/src/HybridSolver/main/tsp_instance_reader.h File Reference

The declaration of the function to read input files.

`#include "data_structures/graph.h"`

Functions

- void read_tsp_lib_file (Graph *graph, char *filename)

  Reads a .tsp file and stores the data in the Graph.
- void read_tsp_csv_file (Graph *graph, char *filename)

  Reads a .csv file and stores the data in the Graph.

9.56.1 Detailed Description

The declaration of the function to read input files.

Author

Lorenzo Sciandra

There are two functions to read the input files, one for the .tsp format and one for the .csv format.

Version

1.0.0 @data 2024-05-1

Copyright

Copyright (c) 2024, license MIT

Repo: `https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound`

Definition in file tsp_instance_reader.h.

9.56.2 Function Documentation

9.56.2.1 read_tsp_csv_file()

```
void read_tsp_csv_file (
            Graph * graph,
            char * filename )
```

Reads a .csv file and stores the data in the Graph.

Parameters

graph	The Graph where the data will be stored.
filename	The name of the file to read.

Definition at line 79 of file tsp_instance_reader.c.

9.56.2.2 read_tsp_lib_file()

```
void read_tsp_lib_file (
            Graph * graph,
            char * filename )
```

Reads a .tsp file and stores the data in the Graph.

Parameters

graph	The Graph where the data will be stored.
filename	The name of the file to read.

Definition at line 18 of file tsp_instance_reader.c.

9.57 tsp_instance_reader.h

Go to the documentation of this file.

```
#ifndef BRANCHANDBOUND1TREE_TSP_INSTANCE_READER_H
#define BRANCHANDBOUND1TREE_TSP_INSTANCE_READER_H
#include "data_structures/graph.h"


void read_tsp_lib_file(Graph * graph, char * filename);


void read_tsp_csv_file(Graph * graph, char * filename);


#endif //BRANCHANDBOUND1TREE_TSP_INSTANCE_READER_H
```

Index