Hybrid TSP Solver

v0.1.0

Generated by Doxygen 1.9.6

Chapter 1

Hybrid TSP Solver

This repository contains my implementation of a hybrid TSP solver for my master's thesis. I named it **hybrid** because it combines the classical 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 solver.

1.1 Idea

My approach involves using the Graph Conv Net 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. For each Node in the branch-and-bound tree, the associated 1-Tree is reformulated by performing a linear number of dual ascent steps to enhance the lower and upper bounds.

1.3 Graph Convolutional Network

I utilized 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.

1.4 Neural Grafting

After obtaining the probabilities for each Edge, I can assign to a 1-Tree the probability of being the optimal tour by averaging the probabilities of its edges. I then use these values as follows:

1. **Candidate node selection**: to construct a 1Tree, a **candidate Node** must be chosen. I try all nodes as the candidate node and select the one that yields the best lower bound. If multiple nodes produce the same lower bound, the one with the highest probability is chosen;
2. **Probabilistic nearest neighbor**: I need an initial feasible solution to reduce the search space using the bounding step. In the classical solver, this is accomplished by performing the nearest neighbor algorithm with each node as the starting node and selecting the lowest tour found as the initial tour. In the hybrid solver I also used a prob-nearest-neighbor algorithm. Starting from each node, I select at every step the unvisited Node 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 algorithm, and the best one is used as the initial feasible solution;
3. **Best-Prob-First search**: all subproblems generated by the branching step are stored and sorted based on their values. In the Hybrid Solver when two subproblems have the same value, the one with the highest probability is selected first. By adjusting some C macros, a trade-off between the value and probability of the 1Trees can be performed.

1.5 Code Documentation

All the code is documented with `Doxygen`, and it is available as a `GitHub Page` and as a PDF.

1.6 Results

Here are the mean values obtained on 100 instances for each Graph size. In each comparison the best value is highlighted in bold:

	Classic Solver	Hybrid Solver
20 nodes 100 instances max 10 minutes		
Total time (s)	**0.028**	1.494
B-and-B time (s)	0.025	**0.020**
B-and-B tree depth	4.72	**3.94**
Generated B-and-B nodes	228.69	**147.95**
Explored B-and-B nodes	170.04	**142.8**
Best value	3.805	3.805
Time to Best (s)	0.008	**0.002**
Depth of the best	1.49	**0.32**
B-and-B nodes before best	110.47	**19.62**
Probability of the best	-	0.974
Mandatory edges in best	3.37	0.72
Forbidden edges in best	1.49	0.32
50 nodes 100 instances max 10 minutes		
Total time (s)	24.931	**16.512**

	Classic Solver	Hybrid Solver
B-and-B time (s)	24.922	**14.633**
B-and-B tree depth	13.57	**12.44**
Generated B-and-B nodes	18384.37	**10519.63**
Explored B-and-B nodes	9850.52	**9225.95**
Best value	5.678	5.678
Time to Best (s)	17.555	**2.825**
Depth of the best	6.33	**1.6**
B-and-B nodes before best	13084.21	**2224.68**
Probability of the best	-	0.988
Mandatory edges in best	23.08	5.0
Forbidden edges in best	6.33	1.6
100 nodes 100 instances max 10 minutes		
Total time (s)	188.989	**103.150**
B-and-B time (s)	188.870	**98.586**
B-and-B tree depth	14.37	**12.49**
Generated B-and-B nodes	37802.4	**13214.54**
Explored B-and-B nodes	10199.05	**7207.29**
Best value	7.753	7.751
Time to Best (s)	128.527	**39.935**
Depth of the best	7.61	**3.49**
B-and-B nodes before best	26659.2	**6652.21**
Probability of the best	-	0.994
Mandatory edges in best	50.46	21.71
Forbidden edges in best	7.61	3.49

Chapter 2

An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem

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.

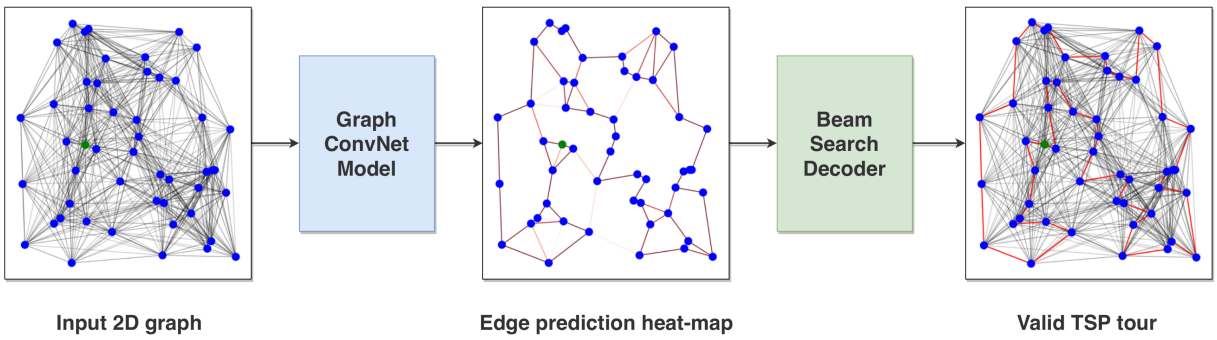


Figure 2.1 model-blocks

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 Hybrid 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

Hierarchical Index

5.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 6

Class Index

6.1 Class List

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

Chapter 7

File Index

7.1 File List

Here is a list of all files with brief descriptions:

Chapter 8

Namespace Documentation

8.1 config Namespace Reference

Classes

- class Settings

Functions

- def get_default_config ()
- def get_config (filepath)

8.1.1 Function Documentation

8.1.1.1 get_config()

```
def config.get_config (
              filepath )
```

```
Returns settings from json file.
```

Definition at line 68 of file config.py.

8.1.1.2 get_default_config()

```
def config.get_default_config ( )
```

```
Returns default settings object.
```

Definition at line 62 of file config.py.

8.2 HybridSolver Namespace Reference

Functions

- def build_c_program (build_directory, num_nodes, hyb_mode)
- def hybrid_solver (num_instances, num_nodes, hyb_mode)

Variables

- sys num_instances = sys.argv[1]
- int num_nodes = int(sys.argv[2])
- tuple hyb_mode = (sys.argv[3] == "y" or sys.argv[3] == "Y" or sys.argv[3] == "yes" or sys.argv[3] == "Yes")

8.2.1 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 0.1.0
@date 2023-04-18
@copyright Copyright (c) 2023, license MIT

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

8.2.2 Function Documentation

8.2.2.1 build_c_program()

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

```
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 22 of file HybridSolver.py.

8.2.2.2 hybrid_solver()

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


Args:
    num_instances: The range of instances to run on the Solver.
    num_nodes: The number of nodes to use in the C program.
    hyb_mode: True if the program is in hybrid mode, False otherwise.
```

Definition at line 53 of file HybridSolver.py.

8.2.3 Variable Documentation

8.2.3.1 hyb_mode

```
tuple HybridSolver.hyb_mode = (sys.argv[3] == "y" or sys.argv[3] == "Y" or sys.argv[3] ==
"yes" or sys.argv[3] == "Yes")
```

Definition at line 126 of file HybridSolver.py.

8.2.3.2 num_instances

```
sys HybridSolver.num_instances = sys.argv[1]
```

Definition at line 124 of file HybridSolver.py.

8.2.3.3 num_nodes

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

Definition at line 125 of file HybridSolver.py.

8.3 main Namespace Reference

Functions

- def compute_prob (net, config, dtypeLong, dtypeFloat, instance_number)
- def write_adjacency_matrix (y_probs, x_edges_values, filepath)
- def main (filepath, num_nodes, instance_number)

Variables

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

8.3.1 Detailed Description

```
@file main.py
@author Lorenzo Sciandra, by Chaitanya K. Joshi, Thomas Laurent and Xavier Bresson.
@brief A recombination of code take from: https://github.com/chaitjo/graph-convnet-tsp.
Some functions were created for the purpose of this project.
@version 0.1.0
@date 2023-04-18
@copyright Copyright (c) 2023, license MIT
Repo: https://github.com/LorenzoSciandra/HybridTSPSolver
```

8.3.2 Function Documentation

8.3.2.1 compute_prob()

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

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.
    instance_number: The number of the instance to be computed.
Returns:
    y_probs: The probability of the edges being in the optimal tour.
    x_edges_values: The distance between the nodes.
```

Definition at line 35 of file main.py.

8.3.2.2 main()

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

The function that calls the previous functions and first sets the parameters for the calculation.
Args:
    filepath: The path to the file where the adjacency matrix will be written.
    num_nodes: The number of nodes in the TSP instance.
    instance_number: The number of the instance to be computed.
```

Definition at line 139 of file main.py.

8.3.2.3 write_adjacency_matrix()

```
def main.write_adjacency_matrix (
              y_probs,
              x_edges_values,
              filepath )
```

```
This function simply writes the probabilistic adjacency matrix in a file, where each cell
is a tuple (distance, probability).
Args:
    y_probs: The probability of the edges being in the optimal tour.
    x_edges_values: The distance between the nodes.
    filepath: The path to the file where the adjacency matrix will be written.
```

Definition at line 107 of file main.py.

8.3.3 Variable Documentation

8.3.3.1 category

```
main.category
```

Definition at line 24 of file main.py.

8.3.3.2 filepath

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

Definition at line 202 of file main.py.

8.3.3.3 instance_number

```
int main.instance_number = int(sys.argv[3]) - 1
```

Definition at line 204 of file main.py.

8.3.3.4 num_nodes

```
sys main.num_nodes = sys.argv[2]
```

Definition at line 203 of file main.py.

8.4 models Namespace Reference

Namespaces

- namespace gcn_layers
- namespace gcn_model

8.5 models.gcn_layers Namespace Reference

Classes

- class BatchNormEdge
- class BatchNormNode
- class EdgeFeatures
- class MLP
- class NodeFeatures
- class ResidualGatedGCNLayer

8.6 models.gcn_model Namespace Reference

Classes

- class ResidualGatedGCNModel

8.7 utils Namespace Reference

Namespaces

- namespace beamsearch
- namespace google_tsp_reader
- namespace graph_utils
- namespace model_utils
- namespace plot_utils

8.8 utils.beamsearch Namespace Reference

Classes

- class Beamsearch

8.9 utils.google_tsp_reader Namespace Reference

Classes

- class DotDict
- class GoogleTSPReader

8.10 utils.graph_utils Namespace Reference

Functions

- def tour_nodes_to_W (nodes)
- def tour_nodes_to_tour_len (nodes, W_values)
- def W_to_tour_len (W, W_values)
- def is_valid_tour (nodes, num_nodes)
- def mean_tour_len_edges (x_edges_values, y_pred_edges)
- def mean_tour_len_nodes (x_edges_values, bs_nodes)
- def get_max_k (dataset, max_iter=1000)

8.10.1 Function Documentation

8.10.1.1 get_max_k()

```
def utils.graph_utils.get_max_k (
              dataset,
              max_iter = 1000 )
```

```
Given a TSP dataset, compute the maximum value of k for which the k'th nearest neighbor
of a node is connected to it in the groundtruth TSP tour.

For each node in all instances, compute the value of k for the next node in the tour,
and take the max of all ks.
```

Definition at line 95 of file graph_utils.py.

8.10.1.2 is_valid_tour()

```
def utils.graph_utils.is_valid_tour (
              nodes,
              num_nodes )
```

```
Sanity check: tour visits all nodes given.
```

Definition at line 47 of file graph_utils.py.

8.10.1.3 mean_tour_len_edges()

```
def utils.graph_utils.mean_tour_len_edges (
              x_edges_values,
              y_pred_edges )
```

```
Computes mean tour length for given batch prediction as edge adjacency matrices (for PyTorch tensors).

Args:
    x_edges_values: Edge values (distance) matrix (batch_size, num_nodes, num_nodes)
    y_pred_edges: Edge predictions (batch_size, num_nodes, num_nodes, voc_edges)

Returns:
    mean_tour_len: Mean tour length over batch
```

Definition at line 53 of file graph_utils.py.

8.10.1.4 mean_tour_len_nodes()

```
def utils.graph_utils.mean_tour_len_nodes (
              x_edges_values,
              bs_nodes )
```

```
Computes mean tour length for given batch prediction as node ordering after beamsearch (for Pytorch tensors).

Args:
    x_edges_values: Edge values (distance) matrix (batch_size, num_nodes, num_nodes)
    bs_nodes: Node orderings (batch_size, num_nodes)

Returns:
    mean_tour_len: Mean tour length over batch
```

Definition at line 72 of file graph_utils.py.

8.10.1.5 tour_nodes_to_tour_len()

```
def utils.graph_utils.tour_nodes_to_tour_len (
              nodes,
              W_values )
```

```
Helper function to calculate tour length from ordered list of tour nodes.
```

Definition at line 22 of file graph_utils.py.

8.10.1.6 tour_nodes_to_W()

```
def utils.graph_utils.tour_nodes_to_W (
              nodes )
```

Helper function to convert ordered list of tour nodes to edge adjacency matrix.

Definition at line 7 of file graph_utils.py.

8.10.1.7 W_to_tour_len()

```
def utils.graph_utils.W_to_tour_len (
              W,
              W_values )
```

Helper function to calculate tour length from edge adjacency matrix.

Definition at line 35 of file graph_utils.py.

8.11 utils.model_utils Namespace Reference

Functions

- def loss_nodes (y_pred_nodes, y_nodes, node_cw)
- def loss_edges (y_pred_edges, y_edges, edge_cw)
- def beamsearch_tour_nodes (y_pred_edges, beam_size, batch_size, num_nodes, dtypeFloat, dtypeLong, probs_type='raw', random_start=False)
- def beamsearch_tour_nodes_shortest (y_pred_edges, x_edges_values, beam_size, batch_size, num_↩ nodes, dtypeFloat, dtypeLong, probs_type='raw', random_start=False)
- def update_learning_rate (optimizer, lr)
- def edge_error (y_pred, y_target, x_edges)
- def _edge_error (y, y_target, mask)

8.11.1 Function Documentation

8.11.1.1 _edge_error()

```
def utils.model_utils._edge_error (
              y,
              y_target,
              mask )  [protected]
```

```
Helper method to compute edge errors.

Args:
    y: Edge predictions (batch_size, num_nodes, num_nodes)
    y_target: Edge targets (batch_size, num_nodes, num_nodes)
    mask: Edges which are not counted in error computation (batch_size, num_nodes, num_nodes)

Returns:
    err: Mean error over batch
    err_idx: One-hot array of shape (batch_size)- 1s correspond to indices which are not perfectly predicted
```

Definition at line 198 of file model_utils.py.

8.11.1.2 beamsearch_tour_nodes()

```
def utils.model_utils.beamsearch_tour_nodes (
              y_pred_edges,
              beam_size,
              batch_size,
              num_nodes,
              dtypeFloat,
              dtypeLong,
              probs_type = 'raw',
              random_start = False )
```

```
Performs beamsearch procedure on edge prediction matrices and returns possible TSP tours.

Args:
    y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
    beam_size: Beam size
    batch_size: Batch size
    num_nodes: Number of nodes in TSP tours
    dtypeFloat: Float data type (for GPU/CPU compatibility)
    dtypeLong: Long data type (for GPU/CPU compatibility)
    random_start: Flag for using fixed (at node 0) vs. random starting points for beamsearch

Returns: TSP tours in terms of node ordering (batch_size, num_nodes)
```

Definition at line 49 of file model_utils.py.

8.11.1.3 beamsearch_tour_nodes_shortest()

```
def utils.model_utils.beamsearch_tour_nodes_shortest (
              y_pred_edges,
              x_edges_values,
              beam_size,
              batch_size,
              num_nodes,
              dtypeFloat,
              dtypeLong,
              probs_type = 'raw',
              random_start = False )
```

```
Performs beamsearch procedure on edge prediction matrices and returns possible TSP tours.

Final predicted tour is the one with the shortest tour length.
(Standard beamsearch returns the one with the highest probability and does not take length into account.)

Args:
    y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
    x_edges_values: Input edge distance matrix (batch_size, num_nodes, num_nodes)
    beam_size: Beam size
    batch_size: Batch size
    num_nodes: Number of nodes in TSP tours
    dtypeFloat: Float data type (for GPU/CPU compatibility)
    dtypeLong: Long data type (for GPU/CPU compatibility)
    probs_type: Type of probability values being handled by beamsearch (either 'raw'/'logits'/'argmax'(TODO))
    random_start: Flag for using fixed (at node 0) vs. random starting points for beamsearch

Returns:
    shortest_tours: TSP tours in terms of node ordering (batch_size, num_nodes)
```

Definition at line 87 of file model_utils.py.

8.11.1.4 edge_error()

```
def utils.model_utils.edge_error (
              y_pred,
              y_target,
              x_edges )
```

```
Computes edge error metrics for given batch prediction and targets.

Args:
    y_pred: Edge predictions (batch_size, num_nodes, num_nodes, voc_edges)
    y_target: Edge targets (batch_size, num_nodes, num_nodes)
    x_edges: Adjacency matrix (batch_size, num_nodes, num_nodes)

Returns:
    err_edges, err_tour, err_tsp, edge_err_idx, err_idx_tour, err_idx_tsp
```

Definition at line 166 of file model_utils.py.

8.11.1.5 loss_edges()

```
def utils.model_utils.loss_edges (
              y_pred_edges,
              y_edges,
              edge_cw )
```

```
Loss function for edge predictions.

Args:
    y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
    y_edges: Targets for edges (batch_size, num_nodes, num_nodes)
    edge_cw: Class weights for edges loss

Returns:
    loss_edges: Value of loss function
```

Definition at line 29 of file model_utils.py.

8.11.1.6 loss_nodes()

```
def utils.model_utils.loss_nodes (
              y_pred_nodes,
              y_nodes,
              node_cw )
```

```
Loss function for node predictions.

Args:
    y_pred_nodes: Predictions for nodes (batch_size, num_nodes)
    y_nodes: Targets for nodes (batch_size, num_nodes)
    node_cw: Class weights for nodes loss

Returns:
    loss_nodes: Value of loss function
```

Definition at line 9 of file model_utils.py.

8.11.1.7 update_learning_rate()

```
def utils.model_utils.update_learning_rate (
              optimizer,
              lr )
```

```
Updates learning rate for given optimizer.

Args:
    optimizer: Optimizer object
    lr: New learning rate

Returns:
    optimizer: Updated optimizer object
    s
```

Definition at line 149 of file model_utils.py.

8.12 utils.plot_utils Namespace Reference

Functions

- def plot_tsp (p, x_coord, W, W_val, W_target, title="default")
- def plot_tsp_heatmap (p, x_coord, W_val, W_pred, title="default")
- def plot_predictions (x_nodes_coord, x_edges, x_edges_values, y_edges, y_pred_edges, num_plots=3)
- def plot_predictions_beamsearch (x_nodes_coord, x_edges, x_edges_values, y_edges, y_pred_edges, bs↩ _nodes, num_plots=3)

8.12.1 Function Documentation

8.12.1.1 plot_predictions()

```
def utils.plot_utils.plot_predictions (
              x_nodes_coord,
              x_edges,
              x_edges_values,
              y_edges,
              y_pred_edges,
              num_plots = 3 )
```

```
Plots groundtruth TSP tour vs. predicted tours (without beamsearch).

Args:
    x_nodes_coord: Input node coordinates (batch_size, num_nodes, node_dim)
    x_edges: Input edge adjacency matrix (batch_size, num_nodes, num_nodes)
    x_edges_values: Input edge distance matrix (batch_size, num_nodes, num_nodes)
    y_edges: Groundtruth labels for edges (batch_size, num_nodes, num_nodes)
    y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
    num_plots: Number of figures to plot
```

Definition at line 88 of file plot_utils.py.

8.12.1.2 plot_predictions_beamsearch()

```
def utils.plot_utils.plot_predictions_beamsearch (
              x_nodes_coord,
              x_edges,
              x_edges_values,
              y_edges,
              y_pred_edges,
              bs_nodes,
              num_plots = 3 )
```

```
Plots groundtruth TSP tour vs. predicted tours (with beamsearch).

Args:
    x_nodes_coord: Input node coordinates (batch_size, num_nodes, node_dim)
    x_edges: Input edge adjacency matrix (batch_size, num_nodes, num_nodes)
    x_edges_values: Input edge distance matrix (batch_size, num_nodes, num_nodes)
    y_edges: Groundtruth labels for edges (batch_size, num_nodes, num_nodes)
    y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
    bs_nodes: Predicted node ordering in TSP tours after beamsearch (batch_size, num_nodes)
    num_plots: Number of figures to plot
```

Definition at line 119 of file plot_utils.py.

8.12.1.3 plot_tsp()

```
def utils.plot_utils.plot_tsp (
              p,
              x_coord,
              W,
              W_val,
              W_target,
              title = "default" )
```

```
Helper function to plot TSP tours.

Args:
    p: Matplotlib figure/subplot
    x_coord: Coordinates of nodes
    W: Edge adjacency matrix
    W_val: Edge values (distance) matrix
    W_target: One-hot matrix with 1s on groundtruth/predicted edges
    title: Title of figure/subplot

Returns:
    p: Updated figure/subplot
```

Definition at line 11 of file plot_utils.py.

8.12.1.4 plot_tsp_heatmap()

```
def utils.plot_utils.plot_tsp_heatmap (
              p,
              x_coord,
              W_val,
              W_pred,
              title = "default" )
```

```
Helper function to plot predicted TSP tours with edge strength denoting confidence of prediction.

Args:
    p: Matplotlib figure/subplot
    x_coord: Coordinates of nodes
    W_val: Edge values (distance) matrix
    W_pred: Edge predictions matrix
    title: Title of figure/subplot

Returns:
    p: Updated figure/subplot
```

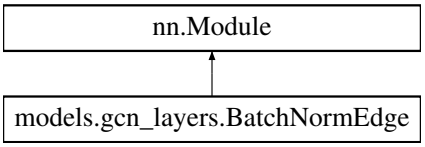
Definition at line 50 of file plot_utils.py.

Chapter 9

Class Documentation

9.1 models.gcn_layers.BatchNormEdge Class Reference

Inheritance diagram for models.gcn_layers.BatchNormEdge:



Public Member Functions

- def __init__ (self, hidden_dim)
- def forward (self, e)

Public Attributes

- batch_norm

9.1.1 Detailed Description

```
Batch normalization for edge features.
```

Definition at line 30 of file gcn_layers.py.

9.1.2 Constructor & Destructor Documentation

9.1.2.1 __init__()

```
def models.gcn_layers.BatchNormEdge.__init__ (
              self,
              hidden_dim )
```

Definition at line 34 of file gcn_layers.py.

9.1.3 Member Function Documentation

9.1.3.1 forward()

```
def models.gcn_layers.BatchNormEdge.forward (
              self,
              e )
```

```
Args:
    e: Edge features (batch_size, num_nodes, num_nodes, hidden_dim)

Returns:
    e_bn: Edge features after batch normalization (batch_size, num_nodes, num_nodes, hidden_dim)
```

Definition at line 38 of file gcn_layers.py.

9.1.4 Member Data Documentation

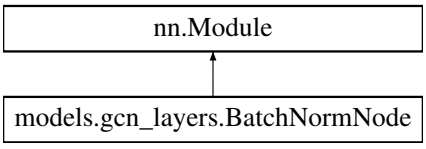
9.1.4.1 batch_norm

```
models.gcn_layers.BatchNormEdge.batch_norm
```

Definition at line 36 of file gcn_layers.py.

9.2 models.gcn_layers.BatchNormNode Class Reference

Inheritance diagram for models.gcn_layers.BatchNormNode:



Public Member Functions

- def __init__ (self, hidden_dim)
- def forward (self, x)

Public Attributes

- batch_norm

9.2.1 Detailed Description

```
Batch normalization for node features.
```

Definition at line 8 of file gcn_layers.py.

9.2.2 Constructor & Destructor Documentation

9.2.2.1 __init__()

```
def models.gcn_layers.BatchNormNode.__init__ (
              self,
              hidden_dim )
```

Definition at line 12 of file gcn_layers.py.

9.2.3 Member Function Documentation

9.2.3.1 forward()

```
def models.gcn_layers.BatchNormNode.forward (
              self,
              x )
```

```
Args:
    x: Node features (batch_size, num_nodes, hidden_dim)

Returns:
    x_bn: Node features after batch normalization (batch_size, num_nodes, hidden_dim)
```

Definition at line 16 of file gcn_layers.py.

9.2.4 Member Data Documentation

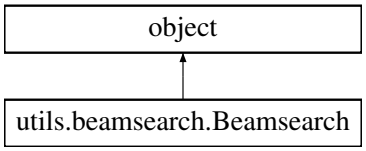
9.2.4.1 batch_norm

`models.gcn_layers.BatchNormNode.batch_norm`

Definition at line 14 of file gcn_layers.py.

9.3 utils.beamsearch.Beamsearch Class Reference

Inheritance diagram for utils.beamsearch.Beamsearch:



Public Member Functions

- def __init__ (self, beam_size, batch_size, num_nodes, dtypeFloat=torch.FloatTensor, dtypeLong=torch.↩ LongTensor, probs_type='raw', random_start=False)
- def get_current_state (self)
- def get_current_origin (self)
- def advance (self, trans_probs)
- def update_mask (self, new_nodes)
- def sort_best (self)
- def get_best (self)
- def get_hypothesis (self, k)

Public Attributes

- batch_size
- beam_size
- num_nodes
- probs_type
- dtypeFloat
- dtypeLong
- start_nodes
- mask
- scores
- all_scores
- prev_Ks
- next_nodes

9.3.1 Detailed Description

```
Class for managing internals of beamsearch procedure.

References:
    General: https://github.com/OpenNMT/OpenNMT-py/blob/master/onmt/translate/beam.py
    For TSP: https://github.com/alexnowakvila/QAP_pt/blob/master/src/tsp/beam_search.py
```

Definition at line 5 of file beamsearch.py.

9.3.2 Constructor & Destructor Documentation

9.3.2.1 __init__()

```
def utils.beamsearch.Beamsearch.__init__ (
              self,
              beam_size,
              batch_size,
              num_nodes,
              dtypeFloat = torch.FloatTensor,
              dtypeLong = torch.LongTensor,
              probs_type = 'raw',
              random_start = False )
```

```
Args:
    beam_size: Beam size
    batch_size: Batch size
    num_nodes: Number of nodes in TSP tours
    dtypeFloat: Float data type (for GPU/CPU compatibility)
    dtypeLong: Long data type (for GPU/CPU compatibility)
    probs_type: Type of probability values being handled by beamsearch (either 'raw'/'logits'/'argmax'(TODO))
    random_start: Flag for using fixed (at node 0) vs. random starting points for beamsearch
```

Definition at line 13 of file beamsearch.py.

9.3.3 Member Function Documentation

9.3.3.1 advance()

```
def utils.beamsearch.Beamsearch.advance (
              self,
              trans_probs )
```

```
Advances the beam based on transition probabilities.

Args:
    trans_probs: Probabilities of advancing from the previous step (batch_size, beam_size, num_nodes)
```

Definition at line 62 of file beamsearch.py.

9.3.3.2 get_best()

```
def utils.beamsearch.Beamsearch.get_best (
              self )
```

Get the score and index of the best hypothesis in the beam.

Definition at line 117 of file beamsearch.py.

9.3.3.3 get_current_origin()

```
def utils.beamsearch.Beamsearch.get_current_origin (
              self )
```

Get the backpointers for the current timestep.

Definition at line 57 of file beamsearch.py.

9.3.3.4 get_current_state()

```
def utils.beamsearch.Beamsearch.get_current_state (
              self )
```

Get the output of the beam at the current timestep.

Definition at line 50 of file beamsearch.py.

9.3.3.5 get_hypothesis()

```
def utils.beamsearch.Beamsearch.get_hypothesis (
              self,
              k )
```

Walk back to construct the full hypothesis.

```
Args:
    k: Position in the beam to construct (usually 0s for most probable hypothesis)
```

Definition at line 123 of file beamsearch.py.

9.3.3.6 sort_best()

```
def utils.beamsearch.Beamsearch.sort_best (
              self )
```

Sort the beam.

Definition at line 112 of file beamsearch.py.

9.3.3.7 update_mask()

```
def utils.beamsearch.Beamsearch.update_mask (
              self,
              new_nodes )
```

Sets new_nodes to zero in mask.

Definition at line 100 of file beamsearch.py.

9.3.4 Member Data Documentation

9.3.4.1 all_scores

```
utils.beamsearch.Beamsearch.all_scores
```

Definition at line 44 of file beamsearch.py.

9.3.4.2 batch_size

```
utils.beamsearch.Beamsearch.batch_size
```

Definition at line 27 of file beamsearch.py.

9.3.4.3 beam_size

```
utils.beamsearch.Beamsearch.beam_size
```

Definition at line 28 of file beamsearch.py.

9.3.4.4 dtypeFloat

`utils.beamsearch.Beamsearch.dtypeFloat`

Definition at line 32 of file beamsearch.py.

9.3.4.5 dtypeLong

`utils.beamsearch.Beamsearch.dtypeLong`

Definition at line 33 of file beamsearch.py.

9.3.4.6 mask

`utils.beamsearch.Beamsearch.mask`

Definition at line 40 of file beamsearch.py.

9.3.4.7 next_nodes

`utils.beamsearch.Beamsearch.next_nodes`

Definition at line 48 of file beamsearch.py.

9.3.4.8 num_nodes

`utils.beamsearch.Beamsearch.num_nodes`

Definition at line 29 of file beamsearch.py.

9.3.4.9 prev_Ks

`utils.beamsearch.Beamsearch.prev_Ks`

Definition at line 46 of file beamsearch.py.

9.3.4.10 probs_type

`utils.beamsearch.Beamsearch.probs_type`

Definition at line 30 of file beamsearch.py.

9.3.4.11 scores

`utils.beamsearch.Beamsearch.scores`

Definition at line 43 of file beamsearch.py.

9.3.4.12 start_nodes

`utils.beamsearch.Beamsearch.start_nodes`

Definition at line 35 of file beamsearch.py.

9.4 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.

9.4.1 Detailed Description

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

Definition at line 20 of file mst.h.

9.4.2 Member Data Documentation

9.4.2.1 dest

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

The destination Node of the Edge.

Definition at line 22 of file mst.h.

9.4.2.2 src

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

The source Node of the Edge.

Definition at line 21 of file mst.h.

9.5 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.

9.5.1 Detailed Description

The double linked List element.

Definition at line 27 of file linked_list.h.

9.5.2 Member Data Documentation

9.5.2.1 next

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

The next element in the List.

Definition at line 29 of file linked_list.h.

9.5.2.2 prev

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

The previous element in the List.

Definition at line 30 of file linked_list.h.

9.5.2.3 value

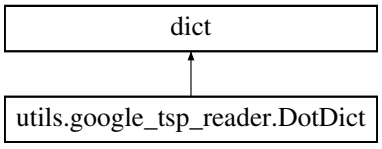
```
void* DllElem::value
```

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.

9.6 utils.google_tsp_reader.DotDict Class Reference

Inheritance diagram for utils.google_tsp_reader.DotDict:



Public Member Functions

- def __init__ (self, **kwds)

Private Attributes

- __dict__

9.6.1 Detailed Description

```
Wrapper around in-built dict class to access members through the dot operation.
```

Definition at line 7 of file google_tsp_reader.py.

9.6.2 Constructor & Destructor Documentation

9.6.2.1 __init__()

```
def utils.google_tsp_reader.DotDict.__init__ (
              self,
             ** kwds )
```

Definition at line 11 of file google_tsp_reader.py.

9.6.3 Member Data Documentation

9.6.3.1 __dict__

```
utils.google_tsp_reader.DotDict.__dict__  [private]
```

Definition at line 13 of file google_tsp_reader.py.

9.7 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.
- float weight
  Weigth of the Edge, 1 if the data_structures is not weighted.
- float 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.

9.7.1 Detailed Description

Structure of an Edge.

Definition at line 40 of file graph.h.

9.7.2 Member Data Documentation

9.7.2.1 dest

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

ID of the destination vertex.

Definition at line 42 of file graph.h.

9.7.2.2 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.

9.7.2.3 prob

```
float Edge::prob
```

Probability of the Edge to be in an optimal tour.

Definition at line 45 of file graph.h.

9.7.2.4 src

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

ID of the source vertex.

Definition at line 41 of file graph.h.

9.7.2.5 symbol

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

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

Definition at line 43 of file graph.h.

9.7.2.6 weight

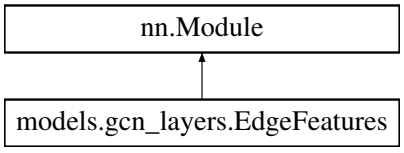
```
float Edge::weight
```

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

Definition at line 44 of file graph.h.

9.8 models.gcn_layers.EdgeFeatures Class Reference

Inheritance diagram for models.gcn_layers.EdgeFeatures:



Public Member Functions

- def __init__ (self, hidden_dim)
- def forward (self, x, e)

Public Attributes

- U
- V

9.8.1 Detailed Description

```
Convnet features for edges.

e_ij = U*e_ij + V*(x_i + x_j)
```

Definition at line 88 of file gcn_layers.py.

9.8.2 Constructor & Destructor Documentation

9.8.2.1 __init__()

```
def models.gcn_layers.EdgeFeatures.__init__ (
              self,
              hidden_dim )
```

Definition at line 94 of file gcn_layers.py.

9.8.3 Member Function Documentation

9.8.3.1 forward()

```
def models.gcn_layers.EdgeFeatures.forward (
              self,
              x,
              e )
```

```
Args:
    x: Node features (batch_size, num_nodes, hidden_dim)
    e: Edge features (batch_size, num_nodes, num_nodes, hidden_dim)

Returns:
    e_new: Convolved edge features (batch_size, num_nodes, num_nodes, hidden_dim)
```

Definition at line 99 of file gcn_layers.py.

9.8.4 Member Data Documentation

9.8.4.1 U

```
models.gcn_layers.EdgeFeatures.U
```

Definition at line 96 of file gcn_layers.py.

9.8.4.2 V

```
models.gcn_layers.EdgeFeatures.V
```

Definition at line 97 of file gcn_layers.py.

9.9 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.

9.9.1 Detailed Description

A Forest is a list of Sets.

Definition at line 28 of file mfset.h.

9.9.2 Member Data Documentation

9.9.2.1 num_sets

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

Number of Sets in the Forest.

Definition at line 29 of file mfset.h.

9.9.2.2 sets

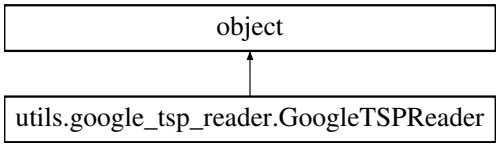
```
Set Forest::sets[MAX_VERTEX_NUM]
```

Array of Sets.

Definition at line 30 of file mfset.h.

9.10 utils.google_tsp_reader.GoogleTSPReader Class Reference

Inheritance diagram for utils.google_tsp_reader.GoogleTSPReader:



Public Member Functions

- def __init__ (self, num_nodes, num_neighbors, batch_size, filepath)
- def __iter__ (self)
- def process_batch (self, lines)

Public Attributes

- num_nodes
- num_neighbors
- batch_size
- filepath
- filedata
- max_iter

9.10.1 Detailed Description

```
Iterator that reads TSP dataset files and yields mini-batches.

Format expected as in Vinyals et al., 2015: https://arxiv.org/abs/1506.03134, http://goo.gl/NDcOIG
```

Definition at line 16 of file google_tsp_reader.py.

9.10.2 Constructor & Destructor Documentation

9.10.2.1 __init__()

```
def utils.google_tsp_reader.GoogleTSPReader.__init__ (
              self,
              num_nodes,
              num_neighbors,
              batch_size,
              filepath )

Args:
    num_nodes: Number of nodes in TSP tours
    num_neighbors: Number of neighbors to consider for each node in graph
    batch_size: Batch size
    filepath: Path to dataset file (.txt file)
```

Definition at line 22 of file google_tsp_reader.py.

9.10.3 Member Function Documentation

9.10.3.1 __iter__()

```
def utils.google_tsp_reader.GoogleTSPReader.__iter__ (
            self )
```

Definition at line 37 of file google_tsp_reader.py.

9.10.3.2 process_batch()

```
def utils.google_tsp_reader.GoogleTSPReader.process_batch (
            self,
            lines )
```

Helper function to convert raw lines into a mini-batch as a DotDict.

Definition at line 43 of file google_tsp_reader.py.

9.10.4 Member Data Documentation

9.10.4.1 batch_size

```
utils.google_tsp_reader.GoogleTSPReader.batch_size
```

Definition at line 32 of file google_tsp_reader.py.

9.10.4.2 filedata

```
utils.google_tsp_reader.GoogleTSPReader.filedata
```

Definition at line 34 of file google_tsp_reader.py.

9.10.4.3 filepath

`utils.google_tsp_reader.GoogleTSPReader.filepath`

Definition at line 33 of file google_tsp_reader.py.

9.10.4.4 max_iter

`utils.google_tsp_reader.GoogleTSPReader.max_iter`

Definition at line 35 of file google_tsp_reader.py.

9.10.4.5 num_neighbors

`utils.google_tsp_reader.GoogleTSPReader.num_neighbors`

Definition at line 31 of file google_tsp_reader.py.

9.10.4.6 num_nodes

`utils.google_tsp_reader.GoogleTSPReader.num_nodes`

Definition at line 30 of file google_tsp_reader.py.

9.11 Graph Struct Reference

Structure of a Graph.

`#include <graph.h>`

Public Attributes

- GraphKind kind
  Type of the Graph.
- float 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.

9.11.1 Detailed Description

Structure of a Graph.

Definition at line 51 of file graph.h.

9.11.2 Member Data Documentation

9.11.2.1 cost

```
float Graph::cost
```

Sum of the weights of the Edges in the Graph.

Definition at line 53 of file graph.h.

9.11.2.2 edges

```
Edge Graph::edges[MAX_EDGES_NUM]
```

Array of Edges.

Definition at line 58 of file graph.h.

9.11.2.3 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.

9.11.2.4 kind

```
GraphKind Graph::kind
```

Type of the Graph.

Definition at line 52 of file graph.h.

9.11.2.5 nodes

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

Array of Nodes.

Definition at line 57 of file graph.h.

9.11.2.6 num_edges

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

Number of Edges in the Graph.

Definition at line 55 of file graph.h.

9.11.2.7 num_nodes

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

Number of Nodes in the Graph.

Definition at line 54 of file graph.h.

9.11.2.8 orderedEdges

```
bool Graph::orderedEdges
```

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

Definition at line 56 of file graph.h.

9.12 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.

9.12.1 Detailed Description

The double linked list.

Definition at line 35 of file linked_list.h.

9.12.2 Member Data Documentation

9.12.2.1 head

```
DllElem* List::head
```

The head of the list as a DllElem.

Definition at line 36 of file linked_list.h.

9.12.2.2 size

```
size_t List::size
```

The current size of the List.

Definition at line 38 of file linked_list.h.

9.12.2.3 tail

```
DllElem* List::tail
```

The tail of the list as a DllElem.

Definition at line 37 of file linked_list.h.

9.13 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.

9.13.1 Detailed Description

The iterator for the List.

Definition at line 43 of file linked_list.h.

9.13.2 Member Data Documentation

9.13.2.1 curr

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

The current DllElem (element) of the List.

Definition at line 45 of file linked_list.h.

9.13.2.2 index

```
size_t ListIterator::index
```

The current index of the element in the List.

Definition at line 46 of file linked_list.h.

9.13.2.3 list

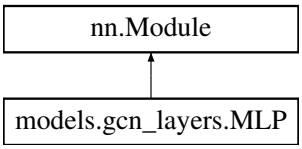
```
List* ListIterator::list
```

The List to iterate.

Definition at line 44 of file linked_list.h.

9.14 models.gcn_layers.MLP Class Reference

Inheritance diagram for models.gcn_layers.MLP:



Public Member Functions

- def __init__ (self, hidden_dim, output_dim, L=2)
- def forward (self, x)

Public Attributes

- L
- U
- V

9.14.1 Detailed Description

```
Multi-layer Perceptron for output prediction.
```

Definition at line 157 of file gcn_layers.py.

9.14.2 Constructor & Destructor Documentation

9.14.2.1 __init__()

```
def models.gcn_layers.MLP.__init__ (
              self,
              hidden_dim,
              output_dim,
              L = 2 )
```

Definition at line 161 of file gcn_layers.py.

9.14.3 Member Function Documentation

9.14.3.1 forward()

```
def models.gcn_layers.MLP.forward (
              self,
              x )
```

```
Args:
    x: Input features (batch_size, hidden_dim)

Returns:
    y: Output predictions (batch_size, output_dim)
```

Definition at line 170 of file gcn_layers.py.

9.14.4 Member Data Documentation

9.14.4.1 L

```
models.gcn_layers.MLP.L
```

Definition at line 163 of file gcn_layers.py.

9.14.4.2 U

```
models.gcn_layers.MLP.U
```

Definition at line 167 of file gcn_layers.py.

9.14.4.3 V

```
models.gcn_layers.MLP.V
```

Definition at line 168 of file gcn_layers.py.

9.15 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.
- float cost

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

9.15.1 Detailed Description

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

Definition at line 27 of file mst.h.

9.15.2 Member Data Documentation

9.15.2.1 cost

`float MST::cost`

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

Definition at line 29 of file mst.h.

9.15.2.2 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 34 of file mst.h.

9.15.2.3 isValid

```
bool MST::isValid
```

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

Definition at line 28 of file mst.h.

9.15.2.4 nodes

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

The set of Nodes in the MST.

Definition at line 33 of file mst.h.

9.15.2.5 num_edges

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

The number of Edges in the MST.

Definition at line 32 of file mst.h.

9.15.2.6 num_nodes

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

The number of Nodes in the MST.

Definition at line 31 of file mst.h.

9.15.2.7 prob

```
float MST::prob
```

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

Definition at line 30 of file mst.h.

9.16 Node Struct Reference

Structure of a Node.

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

Public Attributes

- float x
  x coordinate of the Node.
- float y
  y coordinate of the Node.
- unsigned short positionInGraph
  Position of the Node in the list of Nodes of the Graph.
- 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.

9.16.1 Detailed Description

Structure of a Node.

Definition at line 30 of file graph.h.

9.16.2 Member Data Documentation

9.16.2.1 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.

9.16.2.2 num_neighbours

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

Number of neighbours of the Node.

Definition at line 34 of file graph.h.

9.16.2.3 positionInGraph

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

Position of the Node in the list of Nodes of the Graph.

Definition at line 33 of file graph.h.

9.16.2.4 x

```
float Node::x
```

x coordinate of the Node.

Definition at line 31 of file graph.h.

9.16.2.5 y

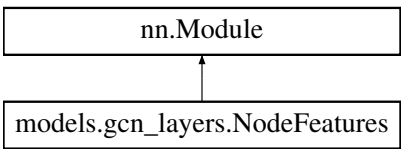
```
float Node::y
```

y coordinate of the Node.

Definition at line 32 of file graph.h.

9.17 models.gcn_layers.NodeFeatures Class Reference

Inheritance diagram for models.gcn_layers.NodeFeatures:



Public Member Functions

- def __init__ (self, hidden_dim, aggregation="mean")
- def forward (self, x, edge_gate)

Public Attributes

- aggregation
- U
- V

9.17.1 Detailed Description

```
Convnet features for nodes.

Using `sum` aggregation:
    x_i = U*x_i +  sum_j [ gate_ij * (V*x_j) ]

Using `mean` aggregation:
    x_i = U*x_i + ( sum_j [ gate_ij * (V*x_j) ] / sum_j [ gate_ij] )
```

Definition at line 52 of file gcn_layers.py.

9.17.2 Constructor & Destructor Documentation

9.17.2.1 __init__()

```
def models.gcn_layers.NodeFeatures.__init__ (
              self,
              hidden_dim,
              aggregation = "mean" )
```

Definition at line 62 of file gcn_layers.py.

9.17.3 Member Function Documentation

9.17.3.1 forward()

```
def models.gcn_layers.NodeFeatures.forward (
              self,
              x,
              edge_gate )

Args:
    x: Node features (batch_size, num_nodes, hidden_dim)
    edge_gate: Edge gate values (batch_size, num_nodes, num_nodes, hidden_dim)

Returns:
    x_new: Convolved node features (batch_size, num_nodes, hidden_dim)
```

Definition at line 68 of file gcn_layers.py.

9.17.4 Member Data Documentation

9.17.4.1 aggregation

`models.gcn_layers.NodeFeatures.aggregation`

Definition at line 64 of file gcn_layers.py.

9.17.4.2 U

`models.gcn_layers.NodeFeatures.U`

Definition at line 65 of file gcn_layers.py.

9.17.4.3 V

`models.gcn_layers.NodeFeatures.V`

Definition at line 66 of file gcn_layers.py.

9.18 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.
- float 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.
- 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.

9.18.1 Detailed Description

The struct used to represent the overall problem.

Definition at line 59 of file b_and_b_data.h.

9.18.2 Member Data Documentation

9.18.2.1 bestSolution

```
SubProblem Problem::bestSolution
```

The best solution found so far.

Definition at line 64 of file b_and_b_data.h.

9.18.2.2 bestValue

```
float Problem::bestValue
```

The cost of the best solution found so far.

Definition at line 65 of file b_and_b_data.h.

9.18.2.3 candidateNodeId

```
unsigned short Problem::candidateNodeId
```

The id of the candidate node.

Definition at line 62 of file b_and_b_data.h.

9.18.2.4 end

```
clock_t Problem::end
```

The time when the algorithm ended.

Definition at line 70 of file b_and_b_data.h.

9.18.2.5 exploredBBNodes

```
unsigned int Problem::exploredBBNodes
```

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

Definition at line 67 of file b_and_b_data.h.

9.18.2.6 generatedBBNodes

```
unsigned int Problem::generatedBBNodes
```

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

Definition at line 66 of file b_and_b_data.h.

9.18.2.7 graph

```
Graph Problem::graph
```

The Graph of the problem.

Definition at line 60 of file b_and_b_data.h.

9.18.2.8 interrupted

```
bool Problem::interrupted
```

True if the algorithm has been interrupted by timeout.

Definition at line 68 of file b_and_b_data.h.

9.18.2.9 reformulationGraph

```
Graph Problem::reformulationGraph
```

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

Definition at line 61 of file b_and_b_data.h.

9.18.2.10 start

```
clock_t Problem::start
```

The time when the algorithm started.

Definition at line 69 of file b_and_b_data.h.

9.18.2.11 totTreeLevels

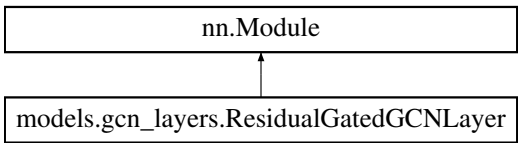
```
unsigned short Problem::totTreeLevels
```

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

Definition at line 63 of file b_and_b_data.h.

9.19 models.gcn_layers.ResidualGatedGCNLayer Class Reference

Inheritance diagram for models.gcn_layers.ResidualGatedGCNLayer:



Public Member Functions

- def __init__ (self, hidden_dim, aggregation="sum")
- def forward (self, x, e)

Public Attributes

- node_feat
- edge_feat
- bn_node
- bn_edge

9.19.1 Detailed Description

```
Convnet layer with gating and residual connection.
```

Definition at line 116 of file gcn_layers.py.

9.19.2 Constructor & Destructor Documentation

9.19.2.1 __init__()

```
def models.gcn_layers.ResidualGatedGCNLayer.__init__ (
              self,
              hidden_dim,
              aggregation = "sum" )
```

Definition at line 120 of file gcn_layers.py.

9.19.3 Member Function Documentation

9.19.3.1 forward()

```
def models.gcn_layers.ResidualGatedGCNLayer.forward (
              self,
              x,
              e )
```

```
Args:
    x: Node features (batch_size, num_nodes, hidden_dim)
    e: Edge features (batch_size, num_nodes, num_nodes, hidden_dim)

Returns:
    x_new: Convolved node features (batch_size, num_nodes, hidden_dim)
    e_new: Convolved edge features (batch_size, num_nodes, num_nodes, hidden_dim)
```

Definition at line 127 of file gcn_layers.py.

9.19.4 Member Data Documentation

9.19.4.1 bn_edge

```
models.gcn_layers.ResidualGatedGCNLayer.bn_edge
```

Definition at line 125 of file gcn_layers.py.

9.19.4.2 bn_node

`models.gcn_layers.ResidualGatedGCNLayer.bn_node`

Definition at line 124 of file gcn_layers.py.

9.19.4.3 edge_feat

`models.gcn_layers.ResidualGatedGCNLayer.edge_feat`

Definition at line 123 of file gcn_layers.py.

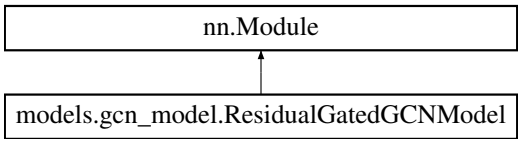
9.19.4.4 node_feat

`models.gcn_layers.ResidualGatedGCNLayer.node_feat`

Definition at line 122 of file gcn_layers.py.

9.20 models.gcn_model.ResidualGatedGCNModel Class Reference

Inheritance diagram for models.gcn_model.ResidualGatedGCNModel:



Public Member Functions

- def __init__ (self, config, dtypeFloat, dtypeLong)
- def forward (self, x_edges, x_edges_values, x_nodes, x_nodes_coord, y_edges, edge_cw)

Public Attributes

- dtypeFloat
- dtypeLong
- num_nodes
- node_dim
- voc_nodes_in
- voc_nodes_out
- voc_edges_in
- voc_edges_out
- hidden_dim
- num_layers
- mlp_layers
- aggregation
- nodes_coord_embedding
- edges_values_embedding
- edges_embedding
- gcn_layers
- mlp_edges

9.20.1 Detailed Description

```
Residual Gated GCN Model for outputting predictions as edge adjacency matrices.

References:
    Paper: https://arxiv.org/pdf/1711.07553v2.pdf
    Code: https://github.com/xbresson/spatial_graph_convnets
```

Definition at line 9 of file gcn_model.py.

9.20.2 Constructor & Destructor Documentation

9.20.2.1 __init__()

```
def models.gcn_model.ResidualGatedGCNModel.__init__ (
              self,
              config,
              dtypeFloat,
              dtypeLong )
```

Definition at line 17 of file gcn_model.py.

9.20.3 Member Function Documentation

9.20.3.1 forward()

```
def models.gcn_model.ResidualGatedGCNModel.forward (
              self,
              x_edges,
              x_edges_values,
              x_nodes,
              x_nodes_coord,
              y_edges,
              edge_cw )


Args:
    x_edges: Input edge adjacency matrix (batch_size, num_nodes, num_nodes)
    x_edges_values: Input edge distance matrix (batch_size, num_nodes, num_nodes)
    x_nodes: Input nodes (batch_size, num_nodes)
    x_nodes_coord: Input node coordinates (batch_size, num_nodes, node_dim)
    y_edges: Targets for edges (batch_size, num_nodes, num_nodes)
    edge_cw: Class weights for edges loss
    # y_nodes: Targets for nodes (batch_size, num_nodes, num_nodes)
    # node_cw: Class weights for nodes loss

Returns:
    y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
    # y_pred_nodes: Predictions for nodes (batch_size, num_nodes)
    loss: Value of loss function
```

Definition at line 45 of file gcn_model.py.

9.20.4 Member Data Documentation

9.20.4.1 aggregation

`models.gcn_model.ResidualGatedGCNModel.aggregation`

Definition at line 31 of file gcn_model.py.

9.20.4.2 dtypeFloat

`models.gcn_model.ResidualGatedGCNModel.dtypeFloat`

Definition at line 19 of file gcn_model.py.

9.20.4.3 dtypeLong

`models.gcn_model.ResidualGatedGCNModel.dtypeLong`

Definition at line 20 of file gcn_model.py.

9.20.4.4 edges_embedding

`models.gcn_model.ResidualGatedGCNModel.edges_embedding`

Definition at line 35 of file gcn_model.py.

9.20.4.5 edges_values_embedding

`models.gcn_model.ResidualGatedGCNModel.edges_values_embedding`

Definition at line 34 of file gcn_model.py.

9.20.4.6 gcn_layers

`models.gcn_model.ResidualGatedGCNModel.gcn_layers`

Definition at line 40 of file gcn_model.py.

9.20.4.7 hidden_dim

`models.gcn_model.ResidualGatedGCNModel.hidden_dim`

Definition at line 28 of file gcn_model.py.

9.20.4.8 mlp_edges

`models.gcn_model.ResidualGatedGCNModel.mlp_edges`

Definition at line 42 of file gcn_model.py.

9.20.4.9 mlp_layers

`models.gcn_model.ResidualGatedGCNModel.mlp_layers`

Definition at line 30 of file gcn_model.py.

9.20.4.10 node_dim

`models.gcn_model.ResidualGatedGCNModel.node_dim`

Definition at line 23 of file gcn_model.py.

9.20.4.11 nodes_coord_embedding

`models.gcn_model.ResidualGatedGCNModel.nodes_coord_embedding`

Definition at line 33 of file gcn_model.py.

9.20.4.12 num_layers

`models.gcn_model.ResidualGatedGCNModel.num_layers`

Definition at line 29 of file gcn_model.py.

9.20.4.13 num_nodes

`models.gcn_model.ResidualGatedGCNModel.num_nodes`

Definition at line 22 of file gcn_model.py.

9.20.4.14 voc_edges_in

`models.gcn_model.ResidualGatedGCNModel.voc_edges_in`

Definition at line 26 of file gcn_model.py.

9.20.4.15 voc_edges_out

`models.gcn_model.ResidualGatedGCNModel.voc_edges_out`

Definition at line 27 of file gcn_model.py.

9.20.4.16 voc_nodes_in

`models.gcn_model.ResidualGatedGCNModel.voc_nodes_in`

Definition at line 24 of file gcn_model.py.

9.20.4.17 voc_nodes_out

`models.gcn_model.ResidualGatedGCNModel.voc_nodes_out`

Definition at line 25 of file gcn_model.py.

9.21 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.

9.21.1 Detailed Description

A Set is a node in the Forest.

Definition at line 19 of file mfset.h.

9.21.2 Member Data Documentation

9.21.2.1 curr

`Node Set::curr`

Current Node.

Definition at line 22 of file mfset.h.

9.21.2.2 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.

9.21.2.3 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.

9.21.2.4 rango

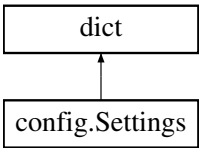
```
unsigned short Set::rango
```

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

Definition at line 21 of file mfset.h.

9.22 config.Settings Class Reference

Inheritance diagram for config.Settings:



Public Member Functions

- def __init__ (self, config_dict)
- def __getattr__ (self, attr)
- def __setitem__ (self, key, value)
- def __setattr__ (self, key, value)

Static Private Attributes

- dict __delattr__ = dict.__delitem__

9.22.1 Detailed Description

```
Experiment configuration options.

Wrapper around in-built dict class to access members through the dot operation.

Experiment parameters:
    "expt_name": Name/description of experiment, used for logging.
    "gpu_id": Available GPU ID(s)

    "train_filepath": Training set path
    "val_filepath": Validation set path
    "test_filepath": Test set path

    "num_nodes": Number of nodes in TSP tours
    "num_neighbors": Number of neighbors in k-nearest neighbor input graph (-1 for fully connected)

    "node_dim": Number of dimensions for each node
    "voc_nodes_in": Input node signal vocabulary size
    "voc_nodes_out": Output node prediction vocabulary size
    "voc_edges_in": Input edge signal vocabulary size
    "voc_edges_out": Output edge prediction vocabulary size

    "beam_size": Beam size for beamsearch procedure (-1 for disabling beamsearch)

    "hidden_dim": Dimension of model's hidden state
    "num_layers": Number of GCN layers
    "mlp_layers": Number of MLP layers
    "aggregation": Node aggregation scheme in GCN (`mean` or `sum`)

    "max_epochs": Maximum training epochs
    "val_every": Interval (in epochs) at which validation is performed
    "test_every": Interval (in epochs) at which testing is performed

    "batch_size": Batch size
    "batches_per_epoch": Batches per epoch (-1 for using full training set)
    "accumulation_steps": Number of steps for gradient accumulation (DO NOT USE: BUGGY)

    "learning_rate": Initial learning rate
    "decay_rate": Learning rate decay parameter
```

Definition at line 4 of file config.py.

9.22.2 Constructor & Destructor Documentation

9.22.2.1 __init__()

```
def config.Settings.__init__ (
              self,
              config_dict )
```

Definition at line 45 of file config.py.

9.22.3 Member Function Documentation

9.22.3.1 __getattr__()

```
def config.Settings.__getattr__ (
              self,
              attr )
```

Definition at line 50 of file config.py.

9.22.3.2 __setattr__()

```
def config.Settings.__setattr__ (
              self,
              key,
              value )
```

Definition at line 56 of file config.py.

9.22.3.3 __setitem__()

```
def config.Settings.__setitem__ (
              self,
              key,
              value )
```

Definition at line 53 of file config.py.

9.22.4 Member Data Documentation

9.22.4.1 __delattr__

```
dict config.Settings.__delattr__ = dict.__delitem__  [static], [private]
```

Definition at line 59 of file config.py.

9.23 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.
- float 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.
- float 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.
- ConstrainedEdge forbiddenEdges [MAX_EDGES_NUM]
  The forbidden edges in the SubProblem.
- unsigned short num_mandatory_edges
  The number of mandatory edges in the SubProblem.
- ConstrainedEdge mandatoryEdges [MAX_EDGES_NUM]
  The mandatory edges in the SubProblem.
- ConstraintType constraints [MAX_VERTEX_NUM][MAX_VERTEX_NUM]
  The constraints of the edges in the SubProblem.

9.23.1 Detailed Description

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

Definition at line 40 of file b_and_b_data.h.

9.23.2 Member Data Documentation

9.23.2.1 constraints

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

The constraints of the edges in the SubProblem.

Definition at line 54 of file b_and_b_data.h.

9.23.2.2 cycleEdges

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

The edges in the cycle of the SubProblem.

Definition at line 49 of file b_and_b_data.h.

9.23.2.3 forbiddenEdges

```
ConstrainedEdge SubProblem::forbiddenEdges[MAX_EDGES_NUM]
```

The forbidden edges in the SubProblem.

Definition at line 51 of file b_and_b_data.h.

9.23.2.4 id

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

The id of the SubProblem, an incremental number.

Definition at line 42 of file b_and_b_data.h.

9.23.2.5 mandatoryEdges

```
ConstrainedEdge SubProblem::mandatoryEdges[MAX_EDGES_NUM]
```

The mandatory edges in the SubProblem.

Definition at line 53 of file b_and_b_data.h.

9.23.2.6 num_edges_in_cycle

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

The number of edges in the cycle of the SubProblem.

Definition at line 47 of file b_and_b_data.h.

9.23.2.7 num_forbidden_edges

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

The number of forbidden edges in the SubProblem.

Definition at line 50 of file b_and_b_data.h.

9.23.2.8 num_mandatory_edges

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

The number of mandatory edges in the SubProblem.

Definition at line 52 of file b_and_b_data.h.

9.23.2.9 oneTree

```
MST SubProblem::oneTree
```

The 1Tree of the SubProblem.

Definition at line 46 of file b_and_b_data.h.

9.23.2.10 prob

```
float SubProblem::prob
```

The probability of the SubProblem to be the best tour.

Definition at line 48 of file b_and_b_data.h.

9.23.2.11 timeToReach

```
float SubProblem::timeToReach
```

The time needed to reach the SubProblem, in seconds.

Definition at line 45 of file b_and_b_data.h.

9.23.2.12 treeLevel

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

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

Definition at line 44 of file b_and_b_data.h.

9.23.2.13 type

```
BBNodeType SubProblem::type
```

The label of the SubProblem.

Definition at line 41 of file b_and_b_data.h.

9.23.2.14 value

```
float SubProblem::value
```

The cost of the SubProblem.

Definition at line 43 of file b_and_b_data.h.

9.24 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.

9.24.1 Detailed Description

The element of the list of SubProblems.

Definition at line 75 of file b_and_b_data.h.

9.24.2 Member Data Documentation

9.24.2.1 next

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

The next element of the list.

Definition at line 77 of file b_and_b_data.h.

9.24.2.2 prev

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

The previous element of the list.

Definition at line 78 of file b_and_b_data.h.

9.24.2.3 subProblem

```
SubProblem SubProblemElem::subProblem
```

The SubProblem.

Definition at line 76 of file b_and_b_data.h.

9.25 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.

9.25.1 Detailed Description

The list of open SubProblems.

Definition at line 83 of file b_and_b_data.h.

9.25.2 Member Data Documentation

9.25.2.1 head

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

The head of the list.

Definition at line 84 of file b_and_b_data.h.

9.25.2.2 size

```
size_t SubProblemsList::size
```

The size of the list.

Definition at line 86 of file b_and_b_data.h.

9.25.2.3 tail

```
SubProblemElem* SubProblemsList::tail
```

The tail of the list.

Definition at line 85 of file b_and_b_data.h.

9.26 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.

9.26.1 Detailed Description

The iterator of the list of SubProblems.

Definition at line 91 of file b_and_b_data.h.

9.26.2 Member Data Documentation

9.26.2.1 curr

```
SubProblemElem* SubProblemsListIterator::curr
```

The current element of the list.

Definition at line 93 of file b_and_b_data.h.

9.26.2.2 index

```
size_t SubProblemsListIterator::index
```

The index of the current element of the list.

Definition at line 94 of file b_and_b_data.h.

9.26.2.3 list

```
SubProblemsList* SubProblemsListIterator::list
```

The list to iterate.

Definition at line 92 of file b_and_b_data.h.

Chapter 10

File Documentation

10.1 HybridTSPSolver/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)

  This function is used to check if the 1Tree of a SubProblem is a tour.
- BBNodeType mst_to_one_tree (SubProblem ∗currentSubproblem, Graph ∗graph)

  This is the function that transforms a MST into a 1Tree.
- void clean_matrix (SubProblem ∗subProblem)

  This function is used to initialize the matrix of ConstraintType for a SubProblem.
- void copy_constraints (SubProblem ∗subProblem, const SubProblem ∗otherSubProblem)

  This function is used to copy the ConstraintType of a SubProblem into another.
- bool compare_subproblems (const SubProblem ∗a, const SubProblem ∗b)

  This function is used to sort the SubProblems in the open list.
- void branch (SubProblemsList ∗openSubProblems, const SubProblem ∗currentSubProblem)

  This function is used to branch a SubProblem into n new SubProblems.
- void held_karp_bound (SubProblem ∗currentSubProb)

  The bound function used to calculate lower and upper bounds.
- bool time_limit_reached (void)

  This function is used to 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)

  This function is used to find the candidate Node for the 1Tree.
- bool check_feasibility (Graph ∗graph)

  This function is used to check if the Graph associated to the Problem is feasible.

- void branch_and_bound (Problem *current_problem)
  This is the main function of the Branch and Bound algorithm.
- void set_problem (Problem *current_problem)
  This function is used to set the pointer to the problem to solve.
- void print_subProblem (const SubProblem *subProblem)
  This function is used to print all the information of a SubProblem.

10.1.1 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

0.1.0

Date

2023-04-18

Copyright

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

Definition in file branch_and_bound.c.

10.1.2 Function Documentation

10.1.2.1 branch()

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

This function is used to branch a SubProblem into n new SubProblems.

The number of new SubProblems is equal to the number of edges in the cycle passing through the candidate Node in the 1Tree.

Parameters

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

Definition at line 199 of file branch_and_bound.c.

10.1.2.2 branch_and_bound()

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

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 551 of file branch_and_bound.c.

10.1.2.3 check_feasibility()

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

This function is used to 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 536 of file branch_and_bound.c.

10.1.2.4 check_hamiltonian()

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

This function is used to check 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.

10.1.2.5 clean_matrix()

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

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

Parameters

subProblem	The SubProblem with no ConstraintType.

Definition at line 160 of file branch_and_bound.c.

10.1.2.6 compare_subproblems()

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

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

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 189 of file branch_and_bound.c.

10.1.2.7 copy_constraints()

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

This function is used to copy the ConstraintType of a SubProblem into another.

Parameters

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

Definition at line 170 of file branch_and_bound.c.

10.1.2.8 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.

10.1.2.9 find_candidate_node()

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

This function is used to find the candidate Node for the 1Tree.

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 503 of file branch_and_bound.c.

10.1.2.10 held_karp_bound()

```
void held_karp_bound (
            SubProblem * currentSubProb )
```

The bound function used to calculate lower and upper bounds.

This function has a primal and dual behaviour. 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 264 of file branch_and_bound.c.

10.1.2.11 mst_to_one_tree()

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

This is the function that 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.

10.1.2.12 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 397 of file branch_and_bound.c.

10.1.2.13 print_subProblem()

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

This function is used to print all the information of a SubProblem.

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

Parameters

subProblem	The SubProblem to print.

Definition at line 605 of file branch_and_bound.c.

10.1.2.14 set_problem()

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

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 600 of file branch_and_bound.c.

10.1.2.15 time_limit_reached()

```
bool time_limit_reached (
            void  )
```

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 392 of file branch_and_bound.c.

10.2 branch_and_bound.c

Go to the documentation of this file.

```
00001
00015 #include "branch_and_bound.h"
00016
00017
00018 void dfs(SubProblem *subProblem) {
00019     List *stack = new_list();
00020     unsigned short num_nodes = subProblem->oneTree.num_nodes;
00021     Node start;
00022     int parentId[num_nodes];
00023     unsigned short pathLength[num_nodes];
00024     bool visited[num_nodes];
00025
00026     for (unsigned short i = 0; i < num_nodes; i++) {
00027         Node current = subProblem->oneTree.nodes[i];
00028         if (current.positionInGraph == problem->graph.nodes[problem->candidateNodeId].positionInGraph)
      {
00029             start = current;
00030         }
00031         parentId[i] = -1;
00032         pathLength[i] = 0;
00033         visited[i] = false;
00034     }
00035
00036     add_elem_list_bottom(stack, &start);
00037
00038     while (stack->size > 0 && parentId[start.positionInGraph] == -1) {
00039         Node *currentNode = get_list_elem_index(stack, 0);
00040         delete_list_elem_index(stack, 0);
00041         if (!visited[currentNode->positionInGraph]) {
00042             visited[currentNode->positionInGraph] = true;
00043             for (unsigned short i = 0; i < currentNode->num_neighbours; i++) {
00044                 unsigned short dest = currentNode->neighbours[i];
00045                 if (!visited[dest]) {
00046                     pathLength[dest] = pathLength[currentNode->positionInGraph] + 1;
00047                     parentId[dest] = currentNode->positionInGraph;
00048                     Node *neighbour = &subProblem->oneTree.nodes[dest];
00049                     add_elem_list_index(stack, neighbour, 0);
00050                 } else if (parentId[dest] == -1) {
00051                     // start node
00052                     unsigned short path = pathLength[currentNode->positionInGraph] + 1;
00053                     if (path > 2) {
00054                         parentId[dest] = currentNode->positionInGraph;
00055                         pathLength[dest] = path;
00056                     }
00057                 }
00058             }
00059         }
00060     }
00061     del_list(stack);
00062
00063     int fromNode = -1;
00064     int toNode = start.positionInGraph;
00065
00066     //printf("Path Length: %d\n", pathLength[toNode]);
00067
00068     if (pathLength[toNode] > 2) {
00069         while (fromNode != start.positionInGraph) {
00070             fromNode = parentId[toNode];
```

```
00071             Edge current_in_cycle = problem->graph.edges_matrix[fromNode][toNode];
00072             subProblem->cycleEdges[subProblem->num_edges_in_cycle].src = current_in_cycle.src;
00073             subProblem->cycleEdges[subProblem->num_edges_in_cycle].dest = current_in_cycle.dest;
00074             subProblem->num_edges_in_cycle++;
00075             toNode = fromNode;
00076         }
00077     }
00078
00079 }
00080
00081
00082 bool check_hamiltonian(SubProblem *subProblem) {
00083     dfs(subProblem);
00084     return subProblem->num_edges_in_cycle == subProblem->oneTree.num_edges;
00085 }
00086
00087
00088 BBNodeType mst_to_one_tree(SubProblem *currentSubproblem, Graph *graph) {
00089     Node candidate = graph->nodes[problem->candidateNodeId];
00090     int bestEdgePos[candidate.num_neighbours];
00091     float bestNewEdgeWeight[candidate.num_neighbours];
00092     unsigned short src = candidate.positionInGraph;
00093     unsigned short others[candidate.num_neighbours];
00094     unsigned short num_others = 0;
00095     unsigned short add = 0;
00096     for (unsigned short i = 0; i < candidate.num_neighbours; i++) {
00097         unsigned short dest = candidate.neighbours[i];
00098
00099         if (currentSubproblem->constraints[src][dest] == MANDATORY) {
00100             Edge mandatoryEdge = graph->edges_matrix[src][dest];
00101             bestEdgePos[add] = mandatoryEdge.positionInGraph;
00102             bestNewEdgeWeight[add] = mandatoryEdge.weight;
00103             add++;
00104         } else if (currentSubproblem->constraints[src][dest] == NOTHING) {
00105             others[num_others] = dest;
00106             num_others++;
00107         }
00108     }
00109     if (add > 2) {
00110         return CLOSED_UNFEASIBLE;
00111     } else {
00112         if (add < 2) {
00113             if (add == 0) {
00114                 bestNewEdgeWeight[0] = INFINITE;
00115                 bestEdgePos[0] = -1;
00116             }
00117             bestNewEdgeWeight[1] = INFINITE;
00118             bestEdgePos[1] = -1;
00119
00120             for (unsigned short j = 0; j < num_others; j++) {
00121                 unsigned short dest = others[j];
00122                 Edge candidateEdge = graph->edges_matrix[src][dest];
00123
00124                 if (add == 0) {
00125                     if ((bestNewEdgeWeight[0] - candidateEdge.weight) > APPROXIMATION) {
00126                         unsigned short temp_pos = bestEdgePos[0];
00127                         float temp_weight = bestNewEdgeWeight[0];
00128                         bestEdgePos[0] = candidateEdge.positionInGraph;
00129                         bestNewEdgeWeight[0] = candidateEdge.weight;
00130                         bestEdgePos[1] = temp_pos;
00131                         bestNewEdgeWeight[1] = temp_weight;
00132
00133                     } else if ((bestNewEdgeWeight[1] - candidateEdge.weight) > APPROXIMATION) {
00134                         bestEdgePos[1] = candidateEdge.positionInGraph;
00135                         bestNewEdgeWeight[1] = candidateEdge.weight;
00136                     }
00137                 } else if (add == 1) {
00138                     if ((bestNewEdgeWeight[1] - candidateEdge.weight) > APPROXIMATION) {
00139                         bestEdgePos[1] = candidateEdge.positionInGraph;
00140                         bestNewEdgeWeight[1] = candidateEdge.weight;
00141                     }
00142                 }
00143             }
00144             if (bestNewEdgeWeight[0] == INFINITE || bestNewEdgeWeight[1] == INFINITE) {
00145                 return CLOSED_UNFEASIBLE;
00146             }
00147         }
00148
00149         Edge best_first = graph->edges[bestEdgePos[0]];
00150         add_edge(&currentSubproblem->oneTree, &best_first);
00151
00152         Edge best_second = graph->edges[bestEdgePos[1]];
00153         add_edge(&currentSubproblem->oneTree, &best_second);
00154
00155         return OPEN;
00156     }
00157 }
```

```
00158
00159
00160 void clean_matrix(SubProblem *subProblem) {
00161     for (short i = 0; i < MAX_VERTEX_NUM; i++) {
00162         for (short j = i; j < MAX_VERTEX_NUM; j++) {
00163             subProblem->constraints[i][j] = NOTHING;
00164             subProblem->constraints[j][i] = NOTHING;
00165         }
00166     }
00167 }
00168
00169
00170 void copy_constraints(SubProblem *subProblem, const SubProblem *otherSubProblem) {
00171     for (short i = 0; i < MAX_VERTEX_NUM; i++) {
00172         for (short j = i; j < MAX_VERTEX_NUM; j++) {
00173             subProblem->constraints[i][j] = otherSubProblem->constraints[i][j];
00174             subProblem->constraints[j][i] = otherSubProblem->constraints[j][i];
00175         }
00176     }
00177
00178     for (short j = 0; j < otherSubProblem->num_forbidden_edges; j++) {
00179         subProblem->forbiddenEdges[j] = otherSubProblem->forbiddenEdges[j];
00180     }
00181
00182     for (short k = 0; k < otherSubProblem->num_mandatory_edges; k++) {
00183         subProblem->mandatoryEdges[k] = otherSubProblem->mandatoryEdges[k];
00184     }
00185 }
00186
00187
00188 // a better than b?
00189 bool compare_subproblems(const SubProblem *a, const SubProblem *b) {
00190     if (HYBRID) {
00191         return ((b->value - a->value) > EPSILON) ||
00192                (((b->value - a->value) > EPSILON2) && ((a->prob - b->prob) >= BETTER_PROB));
00193     } else {
00194         return (b->value - a->value) > EPSILON;
00195     }
00196 }
00197
00198
00199 void branch(SubProblemsList *openSubProblems, const SubProblem *currentSubProblem) {
00200
00201     if (currentSubProblem->treeLevel + 1 > problem->totTreeLevels) {
00202         problem->totTreeLevels = currentSubProblem->treeLevel + 1;
00203     }
00204
00205     for (unsigned short i = 0; i < currentSubProblem->num_edges_in_cycle; i++) {
00206
00207         ConstrainedEdge current_cycle_edge = currentSubProblem->cycleEdges[i];
00208
00209         if (currentSubProblem->constraints[current_cycle_edge.src][current_cycle_edge.dest] ==
    NOTHING) {
00210             problem->generatedBBNodes++;
00211             SubProblem child;
00212             child.num_edges_in_cycle = 0;
00213             child.type = OPEN;
00214             child.prob = currentSubProblem->prob;
00215             child.id = problem->generatedBBNodes;
00216             child.value = currentSubProblem->value;
00217             child.treeLevel = currentSubProblem->treeLevel + 1;
00218             child.num_forbidden_edges = currentSubProblem->num_forbidden_edges;
00219             child.num_mandatory_edges = currentSubProblem->num_mandatory_edges;
00220             copy_constraints(&child, currentSubProblem);
00221
00222             child.forbiddenEdges[currentSubProblem->num_forbidden_edges].src = current_cycle_edge.src;
00223             child.forbiddenEdges[currentSubProblem->num_forbidden_edges].dest =
    current_cycle_edge.dest;
00224             child.constraints[current_cycle_edge.src][current_cycle_edge.dest] = FORBIDDEN;
00225             child.constraints[current_cycle_edge.dest][current_cycle_edge.src] = FORBIDDEN;
00226
00227             child.num_forbidden_edges++;
00228
00229             for (unsigned short z = 0; z < i; z++) {
00230                 ConstrainedEdge mandatory = currentSubProblem->cycleEdges[z];
00231
00232                 if (currentSubProblem->constraints[mandatory.src][mandatory.dest] == NOTHING) {
00233                     child.mandatoryEdges[child.num_mandatory_edges].src = mandatory.src;
00234                     child.mandatoryEdges[child.num_mandatory_edges].dest = mandatory.dest;
00235                     child.num_mandatory_edges++;
00236                     child.constraints[mandatory.src][mandatory.dest] = MANDATORY;
00237                     child.constraints[mandatory.dest][mandatory.src] = MANDATORY;
00238                 }
00239             }
00240
00241             long position = -1;
00242
```

```
00243             SubProblemsListIterator *subProblem_iterators = create_SubProblemList_iterator(
00244                     openSubProblems);
00245             for (size_t j = 0; j < openSubProblems->size && position == -1; j++) {
00246                 SubProblem *open_subProblem = SubProblemList_iterator_get_next(subProblem_iterators);
00247                 if (compare_subproblems(&child, open_subProblem)) {
00248                     position = (long) j;
00249                 }
00250             }
00251             delete_SubProblemList_iterator(subProblem_iterators);
00252             if (position == -1) {
00253                 add_elem_SubProblemList_bottom(openSubProblems, &child);
00254             } else {
00255                 add_elem_SubProblemList_index(openSubProblems, &child, position);
00256             }
00257
00258         }
00259     }
00260
00261 }
00262
00263
00264 void held_karp_bound(SubProblem *currentSubProb) {
00265
00266     if (compare_subproblems(currentSubProb, &problem->bestSolution) || currentSubProb->treeLevel == 0)
    {
00267         problem->exploredBBNodes++;
00268         float pi[MAX_VERTEX_NUM] = {0};
00269         float v[MAX_VERTEX_NUM] = {0};
00270         float v_old[MAX_VERTEX_NUM] = {0};
00271         float total_pi = 0;
00272         int max_iter = currentSubProb->treeLevel == 0 ? (int) NUM_HK_INITIAL_ITERATIONS : (int)
    NUM_HK_ITERATIONS;
00273         float best_lower_bound = currentSubProb->value;
00274         BBNodeType type = currentSubProb->type;
00275         float t_0;
00276         SubProblemsList generatedSubProblems;
00277         new_SubProblemList(&generatedSubProblems);
00278         Graph *used_graph = &problem->graph;
00279         bool first_iter = true;
00280         currentSubProb->timeToReach = ((float) (clock() - problem->start)) / CLOCKS_PER_SEC;
00281
00282         for (int iter = 1; iter <= max_iter && type == OPEN; iter++) {
00283
00284             SubProblem analyzedSubProblem = *currentSubProb;
00285
00286             for (unsigned short j = 0; j < problem->graph.num_edges; j++) {
00287                 if ((pi[used_graph->edges[j].src] +
00288                      pi[used_graph->edges[j].dest]) != 0) {
00289                     used_graph->edges[j].weight += (pi[used_graph->edges[j].src] +
00290                                                     pi[used_graph->edges[j].dest]);
00291
    used_graph->edges_matrix[used_graph->edges[j].src][used_graph->edges[j].dest].weight =
    used_graph->edges[j].weight;
00292
    used_graph->edges_matrix[used_graph->edges[j].dest][used_graph->edges[j].src].weight =
    used_graph->edges[j].weight;
00293                     used_graph->orderedEdges = false;
00294                 }
00295             }
00296
00297             kruskal_constrained(used_graph, &analyzedSubProblem.oneTree, problem->candidateNodeId,
00298                                 analyzedSubProblem.forbiddenEdges,
    analyzedSubProblem.num_forbidden_edges,
00299                                 analyzedSubProblem.mandatoryEdges,
    analyzedSubProblem.num_mandatory_edges);
00300
00301             if (analyzedSubProblem.oneTree.isValid) {
00302                 type = mst_to_one_tree(&analyzedSubProblem, used_graph);
00303
00304                 if (type == OPEN) {
00305
00306                     analyzedSubProblem.value = 0;
00307                     analyzedSubProblem.prob = analyzedSubProblem.oneTree.prob;
00308                     analyzedSubProblem.type = type;
00309
00310                     for (int e = 0; e < problem->graph.num_nodes; e++) {
00311                         Edge *edge = &analyzedSubProblem.oneTree.edges[e];
00312                         analyzedSubProblem.value +=
    problem->graph.edges_matrix[edge->src][edge->dest].weight;
00313                     }
00314
00315                     bool better_value = compare_subproblems(&analyzedSubProblem,
    &problem->bestSolution);
00316                     if (!better_value) {
00317                         analyzedSubProblem.type = CLOSED_BOUND;
00318                     } else {
00319                         analyzedSubProblem.num_edges_in_cycle = 0;
```

```
00320                         if (check_hamiltonian(&analyzedSubProblem)) {
00321                             problem->bestValue = analyzedSubProblem.value;
00322                             analyzedSubProblem.type = CLOSED_NEW_BEST;
00323                             problem->bestSolution = analyzedSubProblem;
00324                         } else {
00325                             analyzedSubProblem.type = OPEN;
00326                         }
00327                     }
00328 
00329                     float current_value = analyzedSubProblem.oneTree.cost - (2 * total_pi);
00330 
00331                     if (current_value > best_lower_bound || first_iter) {
00332                         best_lower_bound = current_value;
00333                         t_0 = best_lower_bound / (2 * MAX_VERTEX_NUM);
00334                         if (first_iter) {
00335                             first_iter = false;
00336                             used_graph = &problem->reformulationGraph;
00337                         }
00338 
00339                     }
00340                     // change the graph to the original one, because the dual variables are calculated
     on the original graph
00341                     *used_graph = problem->graph;
00342                     add_elem_SubProblemList_bottom(&generatedSubProblems, &analyzedSubProblem);
00343 
00344                     for (unsigned short i = 0; i < problem->graph.num_nodes; i++) {
00345                         v[i] = (float) (analyzedSubProblem.oneTree.nodes[i].num_neighbours - 2);
00346                     }
00347 
00348                     float t =
00349                             (((float) (iter - 1)) * (((float) (2 * max_iter) - 5) / (2 * (float)
     (max_iter - 1))) * t_0)
00350                             - (((float) iter - 2) * t_0) +
00351                             ((t_0 * ((float) iter - 1) * ((float) iter - 2)) /
00352                              (2 * ((float) max_iter - 1) * ((float) max_iter - 2)));
00353 
00354                     total_pi = 0;
00355 
00356                     for (unsigned short j = 0; j < problem->graph.num_nodes; j++) {
00357                         if (v[j] != 0) {
00358                             pi[j] += (float) ((0.6 * t * v[j]) + (0.4 * t * v_old[j]));
00359                         }
00360                         v_old[j] = v[j];
00361                         total_pi += pi[j];
00362                     }
00363                 }
00364             } else {
00365                 analyzedSubProblem.type = CLOSED_UNFEASIBLE;
00366                 type = CLOSED_UNFEASIBLE;
00367             }
00368 
00369         }
00370 
00371         float best_value = -1;
00372         SubProblem *best_found = NULL;
00373         SubProblemsListIterator *subProblem_iterators =
     create_SubProblemList_iterator(&generatedSubProblems);
00374         for (size_t j = 0; j < generatedSubProblems.size; j++) {
00375             SubProblem *generatedSubProblem = SubProblemList_iterator_get_next(subProblem_iterators);
00376             if (generatedSubProblem->value > best_value &&
00377                 generatedSubProblem->value <= best_lower_bound &&
00378                 generatedSubProblem->type != CLOSED_UNFEASIBLE) {
00379                 best_value = generatedSubProblem->value;
00380                 best_found = generatedSubProblem;
00381             }
00382         }
00383         *currentSubProb = best_found == NULL ? *currentSubProb : *best_found;
00384         delete_SubProblemList_iterator(subProblem_iterators);
00385         delete_SubProblemList(&generatedSubProblems);
00386     } else {
00387         currentSubProb->type = CLOSED_BOUND;
00388     }
00389 }
00390 
00391 
00392 bool time_limit_reached(void) {
00393     return ((clock() - problem->start) / CLOCKS_PER_SEC) > TIME_LIMIT_SECONDS;
00394 }
00395 
00396 
00397 void nearest_prob_neighbour(unsigned short start_node) {
00398     SubProblem nn_subProblem;
00399     nn_subProblem.num_forbidden_edges = 0;
00400     nn_subProblem.num_mandatory_edges = 0;
00401     nn_subProblem.num_edges_in_cycle = 0;
00402     nn_subProblem.timeToReach = ((float) (clock() - problem->start)) / CLOCKS_PER_SEC;
00403     create_mst(&nn_subProblem.oneTree, problem->graph.nodes, problem->graph.num_nodes);
```

```
00404     unsigned short current_node = start_node;
00405     bool visited[MAX_VERTEX_NUM] = {false};
00406     ConstrainedEdge cycleEdge;
00407 
00408     for (unsigned short visited_count = 0; visited_count < problem->graph.num_nodes; visited_count++)
      {
00409 
00410         if (visited_count == problem->graph.num_nodes - 1) {
00411             add_edge(&nn_subProblem.oneTree, &problem->graph.edges_matrix[current_node][start_node]);
00412             cycleEdge.src = current_node;
00413             cycleEdge.dest = start_node;
00414             nn_subProblem.cycleEdges[nn_subProblem.num_edges_in_cycle] = cycleEdge;
00415             nn_subProblem.num_edges_in_cycle++;
00416         } else {
00417             float best_edge_value = INFINITE;
00418             unsigned short best_neighbour = current_node;
00419             for (unsigned short i = 0; i < problem->graph.nodes[current_node].num_neighbours; i++) {
00420 
00421                 if
      (problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].weight <
00422                     best_edge_value
00423                     && !visited[problem->graph.nodes[current_node].neighbours[i]]) {
00424                     best_edge_value =
      problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].weight;
00425                     best_neighbour = problem->graph.nodes[current_node].neighbours[i];
00426                 }
00427             }
00428             add_edge(&nn_subProblem.oneTree,
      &problem->graph.edges_matrix[current_node][best_neighbour]);
00429             cycleEdge.src = current_node;
00430             cycleEdge.dest = best_neighbour;
00431             nn_subProblem.cycleEdges[nn_subProblem.num_edges_in_cycle] = cycleEdge;
00432             nn_subProblem.num_edges_in_cycle++;
00433             visited[current_node] = true;
00434             current_node = best_neighbour;
00435         }
00436     }
00437     nn_subProblem.value = nn_subProblem.oneTree.cost;
00438     nn_subProblem.oneTree.isValid = true;
00439     nn_subProblem.type = CLOSED_HAMILTONIAN;
00440     nn_subProblem.prob = nn_subProblem.oneTree.prob;
00441 
00442     if (HYBRID) {
00443         SubProblem prob_nn_subProblem;
00444         prob_nn_subProblem.num_forbidden_edges = 0;
00445         prob_nn_subProblem.num_mandatory_edges = 0;
00446         prob_nn_subProblem.num_edges_in_cycle = 0;
00447         create_mst(&prob_nn_subProblem.oneTree, problem->graph.nodes, problem->graph.num_nodes);
00448         bool prob_visited[MAX_VERTEX_NUM] = {false};
00449         current_node = start_node;
00450 
00451         for (unsigned short visited_count = 0; visited_count < problem->graph.num_nodes;
      visited_count++) {
00452 
00453             if (visited_count == problem->graph.num_nodes - 1) {
00454                 add_edge(&prob_nn_subProblem.oneTree,
      &problem->graph.edges_matrix[current_node][start_node]);
00455                 cycleEdge.src = current_node;
00456                 cycleEdge.dest = start_node;
00457                 prob_nn_subProblem.cycleEdges[prob_nn_subProblem.num_edges_in_cycle] = cycleEdge;
00458                 prob_nn_subProblem.num_edges_in_cycle++;
00459             } else {
00460                 float best_edge_prob = -1;
00461                 unsigned short best_neighbour = current_node;
00462                 for (unsigned short i = 0; i < problem->graph.nodes[current_node].num_neighbours; i++)
      {
00463 
00464                     if
      (problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].prob >
00465                         best_edge_prob
00466                         && !prob_visited[problem->graph.nodes[current_node].neighbours[i]]) {
00467                         best_edge_prob =
      problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].prob;
00468                         best_neighbour = problem->graph.nodes[current_node].neighbours[i];
00469                     }
00470                 }
00471                 add_edge(&prob_nn_subProblem.oneTree,
      &problem->graph.edges_matrix[current_node][best_neighbour]);
00472                 cycleEdge.src = current_node;
00473                 cycleEdge.dest = best_neighbour;
00474                 prob_nn_subProblem.cycleEdges[prob_nn_subProblem.num_edges_in_cycle] = cycleEdge;
00475                 prob_nn_subProblem.num_edges_in_cycle++;
00476                 prob_visited[current_node] = true;
00477                 current_node = best_neighbour;
00478             }
00479         }
00480         prob_nn_subProblem.value = prob_nn_subProblem.oneTree.cost;
```

```
00481         prob_nn_subProblem.oneTree.isValid = true;
00482         prob_nn_subProblem.type = CLOSED_HAMILTONIAN;
00483         prob_nn_subProblem.prob = prob_nn_subProblem.oneTree.prob;
00484 
00485         bool better_prob = prob_nn_subProblem.value < nn_subProblem.value;
00486         SubProblem *best = better_prob ? &prob_nn_subProblem : &nn_subProblem;
00487 
00488         if (best->value < problem->bestValue) {
00489             problem->bestValue = best->value;
00490             problem->bestSolution = *best;
00491         }
00492 
00493     } else {
00494         if (nn_subProblem.value < problem->bestValue) {
00495             problem->bestValue = nn_subProblem.value;
00496             problem->bestSolution = nn_subProblem;
00497         }
00498     }
00499 
00500 }
00501 
00502 
00503 unsigned short find_candidate_node(void) {
00504     SubProblemsList findCandidateSubProblems;
00505     new_SubProblemList(&findCandidateSubProblems);
00506     ConstrainedEdge *forbiddenEdges = NULL;
00507     ConstrainedEdge *mandatoryEdges = NULL;
00508 
00509     for (unsigned short i = 0; i < problem->graph.num_nodes; i++) {
00510         SubProblem currentCandidate;
00511         clean_matrix(&currentCandidate);
00512         nearest_prob_neighbour(i);
00513         kruskal_constrained(&problem->graph, &currentCandidate.oneTree, i, forbiddenEdges, 0,
    mandatoryEdges, 0);
00514         mst_to_one_tree(&currentCandidate, &problem->graph);
00515         currentCandidate.value = currentCandidate.oneTree.cost;
00516         currentCandidate.prob = currentCandidate.oneTree.prob;
00517         add_elem_SubProblemList_bottom(&findCandidateSubProblems, &currentCandidate);
00518     }
00519 
00520     unsigned short best_candidate = 0;
00521     SubProblem *best_subProblem = NULL;
00522     SubProblemsListIterator *subProblems_iterators =
    create_SubProblemList_iterator(&findCandidateSubProblems);
00523     for (unsigned short j = 0; j < problem->graph.num_nodes; j++) {
00524         SubProblem *current_subProblem = SubProblemList_iterator_get_next(subProblems_iterators);
00525         if (best_subProblem == NULL || compare_subproblems(best_subProblem, current_subProblem)) {
00526             best_candidate = j;
00527             best_subProblem = current_subProblem;
00528         }
00529     }
00530     delete_SubProblemList_iterator(subProblems_iterators);
00531     delete_SubProblemList(&findCandidateSubProblems);
00532     return best_candidate;
00533 }
00534 
00535 
00536 bool check_feasibility(Graph *graph) {
00537 
00538     bool feasible = true;
00539     for (short i = 0; feasible && i < graph->num_nodes; i++) {
00540         Node current_node = graph->nodes[i];
00541         if (current_node.num_neighbours < 2) {
00542             feasible = false;
00543             printf("\nThe graph is not feasible for the BB algorithm. Node %i has less than 2
    neighbors.",
00544                    current_node.positionInGraph);
00545         }
00546     }
00547     return feasible;
00548 }
00549 
00550 
00551 void branch_and_bound(Problem *current_problem) {
00552 
00553     problem = current_problem;
00554 
00555     if (check_feasibility(&problem->graph)) {
00556         problem->start = clock();
00557         problem->bestValue = INFINITE;
00558         problem->candidateNodeId = find_candidate_node();
00559         problem->exploredBBNodes = 0;
00560         problem->generatedBBNodes = 0;
00561         problem->totTreeLevels = 0;
00562         problem->interrupted = false;
00563         problem->reformulationGraph = problem->graph;
00564 
```

```
00565         SubProblem subProblem;
00566         subProblem.treeLevel = 0;
00567         subProblem.id = problem->generatedBBNodes;
00568         subProblem.type = OPEN;
00569         subProblem.prob = 0;
00570         subProblem.value = INFINITE;
00571         subProblem.num_edges_in_cycle = 0;
00572         subProblem.num_forbidden_edges = 0;
00573         subProblem.num_mandatory_edges = 0;
00574         problem->generatedBBNodes++;
00575         clean_matrix(&subProblem);
00576 
00577         SubProblemsList subProblems;
00578         new_SubProblemList(&subProblems);
00579         add_elem_SubProblemList_bottom(&subProblems, &subProblem);
00580 
00581         while (subProblems.size != 0 && !time_limit_reached()) {
00582             SubProblem current_sub_problem = *get_SubProblemList_elem_index(&subProblems, 0);
00583             delete_SubProblemList_elem_index(&subProblems, 0);
00584             held_karp_bound(&current_sub_problem);
00585             if (current_sub_problem.type == OPEN) {
00586                 branch(&subProblems, &current_sub_problem);
00587             }
00588         }
00589 
00590         if (time_limit_reached()) {
00591             problem->interrupted = true;
00592         }
00593 
00594         problem->end = clock();
00595         delete_SubProblemList(&subProblems);
00596     }
00597 }
00598 
00599 
00600 void set_problem(Problem *current_problem) {
00601     problem = current_problem;
00602 }
00603 
00604 
00605 void print_subProblem(const SubProblem *subProblem) {
00606 
00607     char *type;
00608     if (subProblem->type == OPEN) {
00609         type = "OPEN";
00610     } else if (subProblem->type == CLOSED_UNFEASIBLE) {
00611         type = "CLOSED_UNFEASIBLE";
00612     } else if (subProblem->type == CLOSED_BOUND) {
00613         type = "CLOSED_BOUND";
00614     } else if (subProblem->type == CLOSED_HAMILTONIAN) {
00615         type = "CLOSED_HAMILTONIAN";
00616     } else {
00617         type = "CLOSED_NEW_BEST";
00618     }
00619     printf("\nSUBPROBLEM with cost = %lf, type = %s, level of the BB tree = %i, prob = %lf, BBNode
      number = %u and time to obtain = %lfs",
00620            subProblem->value, type, subProblem->treeLevel, subProblem->oneTree.prob, subProblem->id,
00621            subProblem->timeToReach);
00622 
00623     print_mst_original_weight(&subProblem->oneTree, &problem->graph);
00624 
00625     printf("\nCycle with %i edges:", subProblem->num_edges_in_cycle);
00626     for (unsigned short i = 0; i < subProblem->num_edges_in_cycle; i++) {
00627         ConstrainedEdge edge_cycle = subProblem->cycleEdges[i];
00628         unsigned short src = edge_cycle.src;
00629         unsigned short dest = edge_cycle.dest;
00630         printf("  %i <-> %i ", subProblem->oneTree.nodes[src].positionInGraph,
00631                subProblem->oneTree.nodes[dest].positionInGraph);
00632     }
00633 
00634     printf("\n%i Mandatory edges:", subProblem->num_mandatory_edges);
00635     for (unsigned short j = 0; j < subProblem->num_mandatory_edges; j++) {
00636         ConstrainedEdge mandatory = subProblem->mandatoryEdges[j];
00637         printf("  %i <-> %i ", mandatory.src, mandatory.dest);
00638     }
00639 
00640     printf("\n%i Forbidden edges:", subProblem->num_forbidden_edges);
00641     for (unsigned short z = 0; z < subProblem->num_forbidden_edges; z++) {
00642         ConstrainedEdge forbidden = subProblem->forbiddenEdges[z];
00643         printf("  %i <-> %i ", forbidden.src, forbidden.dest);
00644     }
00645     printf("\n");
00646 }
```

10.3 HybridTSPSolver/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 "../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)

  This function is used to check if the 1Tree of a SubProblem is a tour.
- BBNodeType mst_to_one_tree (SubProblem *currentSubproblem, Graph *graph)

  This is the function that transforms a MST into a 1Tree.
- void clean_matrix (SubProblem *subProblem)

  This function is used to initialize the matrix of ConstraintType for a SubProblem.
- void copy_constraints (SubProblem *subProblem, const SubProblem *otherSubProblem)

  This function is used to copy the ConstraintType of a SubProblem into another.
- bool compare_subproblems (const SubProblem *a, const SubProblem *b)

  This function is used to sort the SubProblems in the open list.
- void branch (SubProblemsList *openSubProblems, const SubProblem *subProblem)

  This function is used to branch a SubProblem into n new SubProblems.
- void held_karp_bound (SubProblem *currentSubProb)

  The bound function used to calculate lower and upper bounds.
- bool time_limit_reached (void)

  This function is used to 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)

  This function is used to find the candidate Node for the 1Tree.
- void branch_and_bound (Problem *current_problem)

  This is the main function of the Branch and Bound algorithm.
- bool check_feasibility (Graph *graph)

  This function is used to check if the Graph associated to the Problem is feasible.
- void set_problem (Problem *current_problem)

  This function is used to set the pointer to the problem to solve.
- void print_subProblem (const SubProblem *subProblem)

  This function is used to print all the information of a SubProblem.

Variables

- static Problem * problem

  The pointer to the problem to solve.

10.3.1 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

0.1.0

Date

2023-04-18

Copyright

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

Definition in file branch_and_bound.h.

10.3.2 Function Documentation

10.3.2.1 branch()

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

This function is used to branch a SubProblem into n new SubProblems.

The number of new SubProblems is equal to the number of edges in the cycle passing through the candidate Node in the 1Tree.

Parameters

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

Definition at line 199 of file branch_and_bound.c.

10.3.2.2 branch_and_bound()

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

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 551 of file branch_and_bound.c.

10.3.2.3 check_feasibility()

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

This function is used to 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 536 of file branch_and_bound.c.

10.3.2.4 check_hamiltonian()

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

This function is used to check 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.

10.3.2.5 clean_matrix()

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

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

Parameters

subProblem	The SubProblem with no ConstraintType.

Definition at line 160 of file branch_and_bound.c.

10.3.2.6 compare_subproblems()

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

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

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 189 of file branch_and_bound.c.

10.3.2.7 copy_constraints()

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

This function is used to copy the ConstraintType of a SubProblem into another.

Parameters

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

Definition at line 170 of file branch_and_bound.c.

10.3.2.8 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.

10.3.2.9 find_candidate_node()

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

This function is used to find the candidate Node for the 1Tree.

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 503 of file branch_and_bound.c.

10.3.2.10 held_karp_bound()

```
void held_karp_bound (
            SubProblem * currentSubProb )
```

The bound function used to calculate lower and upper bounds.

This function has a primal and dual behaviour. 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 264 of file branch_and_bound.c.

10.3.2.11 mst_to_one_tree()

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

This is the function that 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.

10.3.2.12 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 397 of file branch_and_bound.c.

10.3.2.13 print_subProblem()

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

This function is used to print all the information of a SubProblem.

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

Parameters

subProblem	The SubProblem to print.

Definition at line 605 of file branch_and_bound.c.

10.3.2.14 set_problem()

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

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 600 of file branch_and_bound.c.

10.3.2.15 time_limit_reached()

```
bool time_limit_reached (
            void  )
```

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 392 of file branch_and_bound.c.

10.3.3 Variable Documentation

10.3.3.1 problem

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

The pointer to the problem to solve.

Definition at line 21 of file branch_and_bound.h.

10.4 branch_and_bound.h

Go to the documentation of this file.

```
00001
00014 #ifndef BRANCHANDBOUND1TREE_BRANCH_AND_BOUND_H
00015 #define BRANCHANDBOUND1TREE_BRANCH_AND_BOUND_H
00016 #include "kruskal.h"
00017 #include "../data_structures/b_and_b_data.h"
00018
00019
00021 static Problem * problem;
00022
00023
00025
00029 void dfs(SubProblem *subProblem);
00030
00031
00033
00039 bool check_hamiltonian(SubProblem *subProblem);
00040
00041
00043
00049 BBNodeType mst_to_one_tree(SubProblem *currentSubproblem, Graph *graph);
00050
00051
00056 void clean_matrix(SubProblem *subProblem);
00057
00058
00064 void copy_constraints(SubProblem *subProblem, const SubProblem *otherSubProblem);
00065
00066
00073 bool compare_subproblems(const SubProblem *a, const SubProblem *b);
00074
00075
00077
00082 void branch(SubProblemsList *openSubProblems, const SubProblem *subProblem);
00083
00084
00086
00090 void held_karp_bound(SubProblem *currentSubProb);
00091
00092
00097 bool time_limit_reached(void);
00098
00099
00101
00107 void nearest_prob_neighbour(unsigned short start_node);
00108
00109
00111
00116 unsigned short find_candidate_node(void);
00117
00118
00124 void branch_and_bound(Problem * current_problem);
00125
00126
00128
00133 bool check_feasibility(Graph * graph);
00134
00135
00140 void set_problem(Problem * current_problem);
```

```
00141
00142
00144
00148 void print_subProblem(const SubProblem *subProblem);
00149
00150
00151 #endif //BRANCHANDBOUND1TREE_BRANCH_AND_BOUND_H
```

10.5 HybridTSPSolver/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)

  If the Graph is not sorted, this function calls the quick sort algorithm to sort the edges of the Graph.
- void kruskal (Graph *graph, MST *mst)

  The Kruskal algorithm to find the Minimum Spanning Tree O(|E| log |V|)
- void kruskal_constrained (Graph *graph, MST *oneTree, unsigned short candidateId, const ConstrainedEdge *forbiddenEdges, unsigned short numForbidden, const ConstrainedEdge *mandatoryEdges, unsigned short numMandatory)

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

10.5.1 Detailed Description

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

Author

Lorenzo Sciandra

There is also the implementation of the constrained version of Kruskal's algorithm with mandatory and forbidden edges.

Version

0.1.0

Date

2023-04-18

Copyright

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

Definition in file kruskal.c.

10.5.2 Function Documentation

10.5.2.1 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 119 of file kruskal.c.

10.5.2.2 kruskal_constrained()

```
void kruskal_constrained (
            Graph * graph,
            MST * oneTree,
            unsigned short candidateId,
            const ConstrainedEdge * forbiddenEdges,
            unsigned short numForbidden,
            const ConstrainedEdge * mandatoryEdges,
            unsigned short numMandatory )
```

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

The mandatory edges are first added to the MST and then the algorithm continues as the classic Kruskal, but the forbidden edges are not considered.

Parameters

graph	The Graph from which we want to find the Constrained MST.
oneTree	The Constrained Minimum Spanning Tree.
candidateId	The id of the candidate Node.
forbiddenEdges	The list of forbidden edges.
numForbidden	The number of forbidden edges.
mandatoryEdges	The list of mandatory edges.
numMandatory	The number of mandatory edges.

Definition at line 150 of file kruskal.c.

10.5.2.3 pivot_quicksort()

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

Definition at line 39 of file kruskal.c.

10.5.2.4 quick_sort()

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

Definition at line 98 of file kruskal.c.

10.5.2.5 swap()

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

Definition at line 18 of file kruskal.c.

10.5.2.6 wrap_quick_sort()

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

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 111 of file kruskal.c.

10.6 kruskal.c

Go to the documentation of this file.

```
00001
00015 #include "kruskal.h"
00016
00017
00018 static void swap(Graph *graph, unsigned short swap_1, unsigned short swap_2) {
00019
00020     Edge * edges = graph->edges;
00021
00022     //printf("\nswap values %lf - %lf, at %d - %d\n", edges[swap_1].weight, edges[swap_2].weight,
      swap_1, swap_2);
00023
00024     graph->edges_matrix[edges[swap_1].src][edges[swap_1].dest].positionInGraph = swap_2;
00025     graph->edges_matrix[edges[swap_2].src][edges[swap_2].dest].positionInGraph = swap_1;
00026
00027
00028     graph->edges_matrix[edges[swap_1].dest][edges[swap_1].src].positionInGraph = swap_2;
00029     graph->edges_matrix[edges[swap_2].dest][edges[swap_2].src].positionInGraph = swap_1;
00030
00031     edges[swap_1].positionInGraph = swap_2;
00032     edges[swap_2].positionInGraph = swap_1;
00033     Edge temp = edges[swap_1];
00034     edges[swap_1] = edges[swap_2];
00035     edges[swap_2] = temp;
00036 }
00037
00038
00039 static int pivot_quicksort(Graph * graph, unsigned short first, unsigned short last) {
00040     Edge * edges = graph->edges;
00041     Edge last_edge = edges[last];
00042     Edge first_edge = edges[first];
00043     unsigned short middle = (first + last) / 2;
00044     Edge middle_edge = edges[middle];
00045     float pivot = first_edge.weight;
00046
00047     if ((last_edge.weight - first_edge.weight) > APPROXIMATION) {
00048         if ((last_edge.weight - middle_edge.weight) > APPROXIMATION) {
00049             if ((middle_edge.weight - first_edge.weight) > APPROXIMATION) {
00050                 pivot = middle_edge.weight;
00051                 swap(graph, first, middle);
00052             }
00053         } else {
00054             pivot = last_edge.weight;
00055             swap(graph, first, last);
00056         }
00057     } else {
00058         if ((last_edge.weight - middle_edge.weight ) > APPROXIMATION) {
00059             pivot = last_edge.weight;
00060             swap(graph, first, last);
00061
00062         } else if ((first_edge.weight - middle_edge.weight) > APPROXIMATION) {
00063             pivot = middle_edge.weight;
00064             swap(graph, first, middle);
00065         }
00066     }
00067     unsigned short j = last;
00068     unsigned short i = first + 1;
00069     bool condition = true;
00070     while (condition) {
00071         Edge i_edge = edges[i];
00072         while (i <= j && (pivot - i_edge.weight) >= -APPROXIMATION) {
00073             i += 1;
00074             i_edge = edges[i];
00075         }
00076         Edge j_edge = edges[j];
00077         while (i <= j && (j_edge.weight - pivot) > APPROXIMATION) {
00078             j -= 1;
00079             j_edge = edges[j];
00080         }
00081         if (i <= j) {
00082             swap(graph, i,  j);
00083         } else {
00084             condition = false;
00085         }
00086
00087     }
00088
00089     if(j != first){
00090         swap( graph,  first,  j);
00091     }
00092
00093     return j;
00094
```

```
00095 }
00096
00097
00098 static void quick_sort(Graph * graph, unsigned short first, unsigned short last) {
00099     if (first < last) {
00100         unsigned short pivot = pivot_quicksort(graph, first, last);
00101         if(pivot -1 > first) {
00102             quick_sort(graph, first, pivot - 1);
00103         }
00104         if(pivot + 1 < last) {
00105             quick_sort(graph, pivot + 1, last);
00106         }
00107     }
00108 }
00109
00110
00111 void wrap_quick_sort(Graph * graph) {
00112     if (!graph->orderedEdges) {
00113         graph->orderedEdges = true;
00114         quick_sort(graph, 0, graph->num_edges - 1);
00115     }
00116 }
00117
00118
00119 void kruskal(Graph * graph, MST * mst) {
00120     create_mst(mst, graph->nodes, graph->num_nodes);
00121     Forest forest;
00122     create_forest(&forest, graph->nodes, graph->num_nodes);
00123     wrap_quick_sort(graph);
00124     unsigned short num_edges_inG = 0;
00125     unsigned short num_edges_inMST = 0;
00126
00127     while (num_edges_inG < graph->num_edges && num_edges_inMST < graph->num_nodes - 1) {
00128         // get the edge with the minimum weight
00129         Edge current_edge = graph->edges[num_edges_inG];
00130         unsigned short src = current_edge.src;
00131         unsigned short dest = current_edge.dest;
00132
00133         Set *set1_root = find(&forest.sets[src]);
00134         Set *set2_root = find(&forest.sets[dest]);
00135
00136         if (set1_root->num_in_forest != set2_root->num_in_forest) {
00137             merge(set1_root, set2_root);
00138             // add the edge to the MST
00139             add_edge(mst, &current_edge);
00140             num_edges_inMST++;
00141         }
00142         num_edges_inG++;
00143     }
00144     if (num_edges_inMST == graph->num_nodes - 1) {
00145         mst->isValid = true;
00146     }
00147 }
00148
00149
00150 void kruskal_constrained(Graph * graph, MST * oneTree, unsigned short candidateId, const
    ConstrainedEdge * forbiddenEdges,
00151                          unsigned short numForbidden, const ConstrainedEdge * mandatoryEdges, unsigned
    short numMandatory) {
00152     create_mst(oneTree,graph->nodes, graph->num_nodes);
00153     Forest forest;
00154     create_forest_constrained(&forest, graph->nodes, graph->num_nodes, candidateId);
00155     wrap_quick_sort(graph);
00156
00157     unsigned short num_edges_inMST = 0;
00158     for (unsigned short i = 0; i < numMandatory; i++) {
00159         ConstrainedEdge current_mandatory = mandatoryEdges[i];
00160         Edge mandatory_edge = graph->edges_matrix[current_mandatory.src][current_mandatory.dest];
00161         unsigned short src = mandatory_edge.src;
00162         unsigned short dest = mandatory_edge.dest;
00163
00164         if (src != candidateId && dest != candidateId) {
00165
00166             Set *set1_root = find(&forest.sets[src]);
00167             Set *set2_root = find(&forest.sets[dest]);
00168             if (set1_root->num_in_forest != set2_root->num_in_forest) {
00169                 merge(set1_root, set2_root);
00170                 // add the edge to the MST
00171                 add_edge(oneTree, &mandatory_edge);
00172                 num_edges_inMST++;
00173             }
00174         }
00175     }
00176
00177     bool isForbidden = false;
00178     unsigned short num_edges_inG = 0;
00179
```

```
00180     while (num_edges_inG < graph->num_edges && num_edges_inMST < graph->num_nodes - 2) {
00181
00182         Edge current_edge = graph->edges[num_edges_inG];
00183
00184         unsigned short src_id = current_edge.src;
00185         unsigned short dest_id = current_edge.dest;
00186
00187         if (src_id != candidateId && dest_id != candidateId) {
00188
00189             for (unsigned short j = 0; !isForbidden && j < numForbidden; j++) {
00190                 ConstrainedEdge current_mandatory = forbiddenEdges[j];
00191                 Edge forbidden_edge =
      graph->edges_matrix[current_mandatory.src][current_mandatory.dest];
00192
00193                 if (forbidden_edge.src != candidateId && forbidden_edge.dest != candidateId) {
00194                     if (current_edge.symbol == forbidden_edge.symbol) {
00195                         isForbidden = true;
00196                     }
00197                 }
00198             }
00199
00200             if (!isForbidden) {
00201                 // get the edge with the minimum weight
00202                 unsigned short src = current_edge.src;
00203                 unsigned short dest = current_edge.dest;
00204
00205                 Set *set1_root = find(&forest.sets[src]);
00206                 Set *set2_root = find(&forest.sets[dest]);
00207
00208                 if (set1_root->num_in_forest != set2_root->num_in_forest) {
00209                     merge(set1_root, set2_root);
00210                     // add the edge to the MST
00211                     add_edge(oneTree, &current_edge);
00212                     num_edges_inMST++;
00213                 }
00214             }
00215             isForbidden = false;
00216         }
00217
00218         num_edges_inG++;
00219     }
00220
00221     if (num_edges_inMST == graph->num_nodes - 2) {
00222         oneTree->isValid = true;
00223     }
00224 }
```

10.7 HybridTSPSolver/src/HybridSolver/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)

  This function is used to swap 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)

  If the Graph is not sorted, this function calls the quick sort algorithm to sort the edges of the Graph.
- void kruskal (Graph *graph, MST *mst)

  The Kruskal algorithm to find the Minimum Spanning Tree O(|E| log |V|)
- void kruskal_constrained (Graph *graph, MST *oneTree, unsigned short candidateId, const ConstrainedEdge *forbiddenEdges, unsigned short numForbidden, const ConstrainedEdge *mandatoryEdges, unsigned short numMandatory)

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

10.7.1 Detailed Description

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

Author

Lorenzo Sciandra

There is also the implementation of the constrained version of Kruskal's algorithm with mandatory and forbidden edges.

Version

0.1.0

Date

2023-04-18

Copyright

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

Definition in file kruskal.h.

10.7.2 Function Documentation

10.7.2.1 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 119 of file kruskal.c.

10.7.2.2 kruskal_constrained()

```
void kruskal_constrained (
            Graph * graph,
            MST * oneTree,
            unsigned short candidateId,
            const ConstrainedEdge * forbiddenEdges,
            unsigned short numForbidden,
            const ConstrainedEdge * mandatoryEdges,
            unsigned short numMandatory )
```

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

The mandatory edges are first added to the MST and then the algorithm continues as the classic Kruskal, but the forbidden edges are not considered.

Parameters

graph	The Graph from which we want to find the Constrained MST.
oneTree	The Constrained Minimum Spanning Tree.
candidateId	The id of the candidate Node.
forbiddenEdges	The list of forbidden edges.
numForbidden	The number of forbidden edges.
mandatoryEdges	The list of mandatory edges.
numMandatory	The number of mandatory edges.

Definition at line 150 of file kruskal.c.

10.7.2.3 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 all the edges with weight greater than the pivot.

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.

10.7.2.4 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.

10.7.2.5 swap()

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

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.

10.7.2.6 wrap_quick_sort()

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

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 111 of file kruskal.c.

10.8 kruskal.h

Go to the documentation of this file.

```
00001
00015 #ifndef BRANCHANDBOUND1TREE_KRUSKAL_H
00016 #define BRANCHANDBOUND1TREE_KRUSKAL_H
00017 #include "../data_structures/mst.h"
00018
00019
00026 static void swap(Graph * graph, unsigned short swap_1, unsigned short swap_2);
00027
00028
00030
00038 static int pivot_quicksort(Graph * graph, unsigned short first, unsigned short last);
00039
00040
00042
00048 static void quick_sort(Graph * graph, unsigned short first, unsigned short last);
00049
00050
00055 void wrap_quick_sort(Graph * graph);
00056
00057
00059
00064 void kruskal(Graph * graph, MST * mst);
00065
00066
00068
00079 void kruskal_constrained(Graph * graph, MST * oneTree, unsigned short candidateId, const
      ConstrainedEdge * forbiddenEdges,
00080                          unsigned short numForbidden, const ConstrainedEdge * mandatoryEdges, unsigned
      short numMandatory);
00081
00082
00083 #endif //BRANCHANDBOUND1TREE_KRUSKAL_H
```

10.9 HybridTSPSolver/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.

10.9.1 Detailed Description

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

Author

Lorenzo Sciandra

Version

0.1.0

Date

2023-04-18

Copyright

Copyright (c) 2023, license MIT

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

Definition in file b_and_b_data.c.

10.9.2 Function Documentation

10.9.2.1 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.

10.9.2.2 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.

10.9.2.3 build_list_elem()

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

Definition at line 44 of file b_and_b_data.c.

10.9.2.4 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.

10.9.2.5 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.

10.9.2.6 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.

10.9.2.7 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.

10.9.2.8 get_current_openSubProblemList_iterator_element()

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

Definition at line 174 of file b_and_b_data.c.

10.9.2.9 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.

10.9.2.10 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.

10.9.2.11 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.

10.9.2.12 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.

10.9.2.13 list_openSubProblemList_next()

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

Definition at line 178 of file b_and_b_data.c.

10.9.2.14 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.

10.9.2.15 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.

10.10 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;
```

```
00019     list->head = list->tail = NULL;
00020 }
00021
00022
00023 void delete_SubProblemList(SubProblemsList * list){
00024     if (!list) {
00025         return;
00026     }
00027
00028     SubProblemElem *current = list->head;
00029     SubProblemElem *next;
00030
00031     while (current) {
00032         next = current->next;
00033         free(current);
00034         current = next;
00035     }
00036 }
00037
00038
00039 bool is_SubProblemList_empty(SubProblemsList *list){
00040     return (list == NULL || list->size == 0);
00041 }
00042
00043
00044 SubProblemElem *build_list_elem(SubProblem *value, SubProblemElem *next, SubProblemElem *prev) {
00045     SubProblemElem *e = malloc(sizeof(SubProblemElem));
00046     e->subProblem = *value;
00047     e->next = next;
00048     e->prev = prev;
00049
00050     return e;
00051 }
00052
00053
00054 size_t get_SubProblemList_size(SubProblemsList *list){
00055     return (list != NULL) ? list->size : 0;
00056 }
00057
00058
00059 void add_elem_SubProblemList_bottom(SubProblemsList *list, SubProblem *element){
00060     if (list == NULL) {
00061         return;
00062     }
00063
00064     SubProblemElem *e = build_list_elem(element, NULL, list->tail);
00065
00066     if (is_SubProblemList_empty(list))
00067         list->head = e;
00068     else
00069         list->tail->next = e;
00070     list->tail = e;
00071     list->size++;
00072 }
00073
00074
00075 void add_elem_SubProblemList_index(SubProblemsList *list, SubProblem *element, size_t index){
00077     if (!list || index > get_SubProblemList_size(list)) {
00078         return;
00079     }
00080
00081 // support element is a temporary pointer which avoids losing data
00082     SubProblemElem *e;
00083     SubProblemElem *supp = list->head;
00084
00085     for (size_t i = 0; i < index; ++i)
00086         supp = supp->next;
00087
00088     if (supp == list->head) {
00089         e = build_list_elem(element, supp, NULL);
00090
00091         if (supp == NULL) {
00092             list->head = list->tail = e;
00093         } else {
00094 //          e->next->prev = e;
00095             list->head->prev = e;
00096             list->head = e;
00097         }
00098     } else {
00099         if (supp == NULL) {
00100             e = build_list_elem(element, NULL, list->tail);
00101             list->tail->next = e;
00102         } else {
00103             e = build_list_elem(element, supp, supp->prev);
00104             e->next->prev = e;
00105             e->prev->next = e;
00106         }
```

```
00107     }
00108
00109     list->size++;
00110 }
00111
00112
00113 void delete_SubProblemList_elem_index(SubProblemsList *list, size_t index){
00114     if (list == NULL || is_SubProblemList_empty(list) || index >= get_SubProblemList_size(list)) {
00115         return;
00116     }
00117
00118     SubProblemElem *oldElem;
00119     oldElem = list->head;
00120
00121     for (size_t i = 0; i < index; ++i)
00122         oldElem = oldElem->next;
00123
00124     // Found index  to remove!!
00125     if (oldElem != list->head) {
00126         oldElem->prev->next = oldElem->next;
00127         if (oldElem->next != NULL) {
00128             oldElem->next->prev = oldElem->prev;
00129         } else {
00130             list->tail = oldElem->prev;
00131         }
00132     } else {
00133         if (list->head == list->tail) {
00134             list->head = list->tail = NULL;
00135         } else {
00136             list->head = list->head->next;
00137             list->head->prev = NULL;
00138         }
00139     }
00140
00141     free(oldElem);
00142     list->size--;
00143 }
00144
00145
00146 SubProblem *get_SubProblemList_elem_index(SubProblemsList *list, size_t index){
00147     if (list == NULL  || index >= get_SubProblemList_size(list)) {
00148         return NULL;
00149     }
00150
00151     SubProblemElem *supp; // iteration support element
00152     supp = list->head;
00153
00154     for (size_t i = 0; i < index; ++i)
00155         supp = supp->next;
00156     return &supp->subProblem;
00157 }
00158
00159 SubProblemsListIterator *create_SubProblemList_iterator(SubProblemsList *list) {
00160     if (!list)
00161         return NULL;
00162
00163     SubProblemsListIterator *new_iterator = malloc(sizeof(SubProblemsListIterator));
00164     new_iterator->list = list;
00165     new_iterator->curr = new_iterator->list->head;
00166     new_iterator->index = 0;
00167     return new_iterator;
00168 }
00169
00170 bool is_SubProblemList_iterator_valid(SubProblemsListIterator *iterator){
00171     return (iterator) ? iterator->index < get_SubProblemList_size(iterator->list) : 0;
00172 }
00173
00174 SubProblem *get_current_openSubProblemList_iterator_element(SubProblemsListIterator *iterator) {
00175     return (iterator && iterator->curr) ? &iterator->curr->subProblem : NULL;
00176 }
00177
00178 void list_openSubProblemList_next(SubProblemsListIterator *iterator) {
00179     if (is_SubProblemList_iterator_valid(iterator)) {
00180         iterator->index++;
00181
00182         if (is_SubProblemList_iterator_valid(iterator)) {
00183             iterator->curr = iterator->curr->next;
00184         }
00185     }
00186 }
00187
00188 SubProblem *SubProblemList_iterator_get_next(SubProblemsListIterator *iterator){
00189     if (!is_SubProblemList_iterator_valid(iterator)) {
00190         return NULL;
00191     }
00192
00193     SubProblem *element = get_current_openSubProblemList_iterator_element(iterator);
```

```
00194     list_openSubProblemList_next(iterator);
00195     return element;
00196 }
00197
00198 void delete_SubProblemList_iterator(SubProblemsListIterator *iterator){
00199     free(iterator);
00200 }
```

10.11 HybridTSPSolver/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_BOUND , CLOSED_HAMILTONIAN , CLOSED_UNFEASIBLE ,
  CLOSED_NEW_BEST }

  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.

10.11.1 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

0.1.0

Date

2023-04-18

Copyright

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

Definition in file b_and_b_data.h.

10.11.2 Typedef Documentation

10.11.2.1 BBNodeType

`typedef enum BBNodeType BBNodeType`

The labels used to identify the type of a SubProblem.

10.11.2.2 ConstraintType

`typedef enum ConstraintType ConstraintType`

The enum used to identify the type of Edge constraints.

10.11.2.3 Problem

`typedef struct Problem Problem`

The struct used to represent the overall problem.

10.11.2.4 SubProblem

`typedef struct SubProblem SubProblem`

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

10.11.2.5 SubProblemElem

`typedef struct SubProblemElem SubProblemElem`

The element of the list of SubProblems.

10.11.2.6 SubProblemsList

`typedef struct SubProblemsList SubProblemsList`

The list of open SubProblems.

10.11.3 Enumeration Type Documentation

10.11.3.1 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_BOUND	The SubProblem is a feasible 1Tree, with a value greater than the best solution found so far.
CLOSED_HAMILTONIAN	The SubProblem is a feasible Hamiltonian cycle, with a value grater than the best solution found so far, and so discarded.
CLOSED_UNFEASIBLE	The SubProblem is not a feasible 1Tree, and so discarded.
CLOSED_NEW_BEST	The SubProblem is a feasible tour, with a value lower than the best solution found so far, new best solution found.

Definition at line 22 of file b_and_b_data.h.

10.11.3.2 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 32 of file b_and_b_data.h.

10.11.4 Function Documentation

10.11.4.1 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.

10.11.4.2 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.

10.11.4.3 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.

10.11.4.4 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.

10.11.4.5 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.

10.11.4.6 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.

10.11.4.7 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.

10.11.4.8 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.

10.11.4.9 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.

10.11.4.10 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.

10.11.4.11 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.

10.11.4.12 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.

10.12 b_and_b_data.h

Go to the documentation of this file.

```
00001
00016 #ifndef BRANCHANDBOUND1TREE_B_AND_B_DATA_H
00017 #define BRANCHANDBOUND1TREE_B_AND_B_DATA_H
00018
00019 #include "mst.h"
00020
00022 typedef enum BBNodeType{
00023     OPEN,
00024     CLOSED_BOUND,
00025     CLOSED_HAMILTONIAN,
00026     CLOSED_UNFEASIBLE,
00027     CLOSED_NEW_BEST
00028 }BBNodeType;
00029
00030
00032 typedef enum ConstraintType{
00033     NOTHING,
00034     MANDATORY,
00035     FORBIDDEN
00036 }ConstraintType;
00037
00038
00040 typedef struct SubProblem{
00041     BBNodeType type;
00042     unsigned int id;
00043     float value;
00044     unsigned short treeLevel;
00045     float timeToReach;
00046     MST oneTree;
00047     unsigned short num_edges_in_cycle;
00048     float prob;
00049     ConstrainedEdge cycleEdges [MAX_VERTEX_NUM];
00050     unsigned short num_forbidden_edges;
00051     ConstrainedEdge forbiddenEdges [MAX_EDGES_NUM];
00052     unsigned short num_mandatory_edges;
00053     ConstrainedEdge mandatoryEdges [MAX_EDGES_NUM];
00054     ConstraintType constraints [MAX_VERTEX_NUM][MAX_VERTEX_NUM];
00055 }SubProblem;
00056
00057
00059 typedef struct Problem{
00060     Graph graph;
00061     Graph reformulationGraph;
00062     unsigned short candidateNodeId;
00063     unsigned short totTreeLevels;
00064     SubProblem bestSolution;
00065     float bestValue;
00066     unsigned int generatedBBNodes;
00067     unsigned int exploredBBNodes;
00068     bool interrupted;
00069     clock_t start;
00070     clock_t end;
00071 }Problem;
00072
00073
00075 typedef struct SubProblemElem{
00076     SubProblem subProblem;
00077     struct SubProblemElem * next;
00078     struct SubProblemElem * prev;
00079 }SubProblemElem;
00080
00081
00083 typedef struct SubProblemsList{
00084     SubProblemElem * head;
00085     SubProblemElem * tail;
00086     size_t size;
00087 }SubProblemsList;
00088
00089
00091 typedef struct {
```

```
00092     SubProblemsList * list;
00093     SubProblemElem * curr;
00094     size_t index;
00095 } SubProblemsListIterator;
00096
00097
00102 void new_SubProblemList(SubProblemsList * list);
00103
00104
00109 void delete_SubProblemList(SubProblemsList * list);
00110
00111
00117 bool is_SubProblemList_empty(SubProblemsList *list);
00118
00119
00125 size_t get_SubProblemList_size(SubProblemsList *list);
00126
00127
00133 void add_elem_SubProblemList_bottom(SubProblemsList *list, SubProblem *element);
00134
00135
00142 void add_elem_SubProblemList_index(SubProblemsList *list, SubProblem *element, size_t index);
00143
00144
00150 void delete_SubProblemList_elem_index(SubProblemsList *list, size_t index);
00151
00152
00159 SubProblem *get_SubProblemList_elem_index(SubProblemsList *list, size_t index);
00160
00161
00167 SubProblemsListIterator  *create_SubProblemList_iterator(SubProblemsList *list);
00168
00169
00171
00176 bool is_SubProblemList_iterator_valid(SubProblemsListIterator *iterator);
00177
00178
00184 SubProblem *SubProblemList_iterator_get_next(SubProblemsListIterator *iterator);
00185
00186
00191 void delete_SubProblemList_iterator(SubProblemsListIterator *iterator);
00192
00193 #endif //BRANCHANDBOUND1TREE_B_AND_B_DATA_H
```

10.13 HybridTSPSolver/src/HybridSolver/main/data_structures/graph.c File Reference

The implementation of the graph data structure.

```
#include "graph.h"
#include "linked_list/list_functions.h"
#include "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.

10.13.1 Detailed Description

The implementation of the graph data structure.

Author

Lorenzo Sciandra

Version

0.1.0

Date

2023-04-18

Copyright

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

Definition in file graph.c.

10.13.2 Function Documentation

10.13.2.1 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.

10.13.2.2 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 19 of file graph.c.

10.13.2.3 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.

10.14 graph.c

Go to the documentation of this file.

```
00001
00014 #include "graph.h"
00015 #include "linked_list/list_functions.h"
00016 #include "linked_list/list_iterator.h"
00017
00018
00019 void create_graph(Graph * graph, List *nodes_list, List *edges_list, GraphKind kind) {
00020     graph->kind = kind;
00021     graph->num_edges = 0;
00022     graph->num_nodes = 0;
00023     graph->orderedEdges = false;
00024     graph->cost = 0;
00025
00026     ListIterator *nodes_iterator = create_list_iterator(nodes_list);
00027     unsigned short numNodes = 0;
00028     for (size_t j = 0; j < nodes_list->size; j++) {
00029         Node *curr = list_iterator_get_next(nodes_iterator);
00030         graph->nodes[numNodes].positionInGraph = numNodes;
00031         graph->nodes[numNodes].num_neighbours = 0;
00032         graph->nodes[numNodes].y = curr->y;
00033         graph->nodes[numNodes].x = curr->x;
00034         graph->num_nodes++;
00035         numNodes++;
00036     }
00037     delete_list_iterator(nodes_iterator);
00038
```

```
00039     unsigned short numEdges = 0;
00040     ListIterator *edges_iterator = create_list_iterator(edges_list);
00041     for (size_t i = 0; i < edges_list->size ; i++) {
00042         //add the source vertex to the data_structures
00043         Edge * current_edge = list_iterator_get_next(edges_iterator);
00044         unsigned short src = current_edge->src;
00045         unsigned short dest = current_edge->dest;
00046 
00047         graph->edges[numEdges].dest = dest;
00048         graph->edges[numEdges].src = src;
00049         graph->edges[numEdges].prob = current_edge->prob;
00050         graph->edges[numEdges].weight = current_edge->weight;
00051         graph->edges[numEdges].symbol = current_edge->symbol;
00052         graph->edges[numEdges].positionInGraph =  numEdges;
00053         graph->nodes[src].neighbours[graph->nodes[src].num_neighbours] = dest;
00054         graph->nodes[src].num_neighbours++;
00055         graph->edges_matrix[src][dest] = graph->edges[numEdges];
00056         graph->cost += current_edge->weight;
00057 
00058         graph->edges_matrix[dest][src] = graph->edges_matrix[src][dest];
00059         graph->nodes[dest].neighbours[graph->nodes[dest].num_neighbours] = src;
00060         graph->nodes[dest].num_neighbours++;
00061 
00062         numEdges++;
00063         graph->num_edges++;
00064     }
00065     delete_list_iterator(edges_iterator);
00066     del_list(edges_list);
00067     del_list(nodes_list);
00068 }
00069 
00070 
00071 void create_euclidean_graph(Graph * graph, List *nodes) {
00072     List *edges_list = new_list();
00073 
00074     unsigned short z = 0;
00075     Edge edges [MAX_EDGES_NUM];
00076     ListIterator *i_nodes_iterator = create_list_iterator(nodes);
00077     for (size_t i = 0; i < nodes->size; i++) {
00078         Node *node_src = list_iterator_get_next(i_nodes_iterator);
00079         for (size_t j = i + 1; j < nodes->size; j++) {
00080             Node *node_dest = get_list_elem_index(nodes,j);
00081 
00082             edges[z].src = node_src->positionInGraph;
00083             edges[z].dest = node_dest->positionInGraph;
00084             edges[z].symbol = z + 1;
00085             edges[z].positionInGraph = z;
00086             edges[z].prob = 0;
00087             edges[z].weight = (float) sqrt(pow(fabsf(node_src->x - node_dest->x), 2) +
00088                                            pow(fabsf(node_src->y - node_dest->y), 2));
00089             add_elem_list_bottom(edges_list, &edges[z]);
00090             z++;
00091 
00092         }
00093 
00094     }
00095     delete_list_iterator(i_nodes_iterator);
00096 
00097     create_graph(graph,nodes, edges_list, WEIGHTED_GRAPH);
00098 }
00099 
00100 
00101 void print_graph(const Graph *G) {
00102     printf("Nodes: %i\n", G->num_nodes);
00103     for (int i = 0; i < G->num_nodes; i++) {
00104         Node curr = G->nodes[i];
00105         printf("Node%i:\t(%.3f, %.3f)\t%i neighbours: ", curr.positionInGraph, curr.x, curr.y,
      curr.num_neighbours);
00106 
00107         for (int z = 0; z < curr.num_neighbours; z++) {
00108             printf("%i ", G->nodes[curr.neighbours[z]].positionInGraph);
00109         }
00110         printf("\n");
00111     }
00112 
00113     printf("\nCost: %lf\n", G->cost);
00114     printf("\nEdges: %i\n", G->num_edges);
00115 
00116     double dim = (log(G->num_nodes) / log(10) + 1) * 2 + 7;
00117     for (unsigned short j = 0; j < G->num_edges; j++) {
00118         char edge_print [(int) dim] ;
00119         char edge_print_dest [(int) (dim-7)/2] ;
00120         Edge curr = G->edges[j];
00121         sprintf(edge_print, "%i", curr.src);
00122         strcat(edge_print, " <--> ");
00123         sprintf(edge_print_dest, "%i", curr.dest);
00124         strcat(edge_print, edge_print_dest);
```

```
00125
00126         printf("Edge%i:\t%s\tweight = %lf\tprob = %lf\n",
00127                curr.symbol,
00128                edge_print,
00129                curr.weight,
00130                curr.prob);
00131     }
00132
00133 }
```

10.15 HybridTSPSolver/src/HybridSolver/main/data_structures/graph.h File Reference

The data structures to model the Graph.

```
#include "./linked_list/linked_list.h"
#include "./linked_list/list_iterator.h"
#include "./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.

10.15.1 Detailed Description

The data structures to model the Graph.

Author

Lorenzo Sciandra

Version

0.1.0

Date

2023-04-18

Copyright

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

Definition in file graph.h.

10.15.2 Typedef Documentation

10.15.2.1 Edge

```
typedef struct Edge Edge
```

Structure of an Edge.

10.15.2.2 Graph

```
typedef struct Graph Graph
```

Structure of a Graph.

10.15.2.3 GraphKind

```
typedef enum GraphKind GraphKind
```

Enum to specify the kind of the Graph.

10.15.2.4 Node

```
typedef struct Node Node
```

Structure of a Node.

10.15.3 Enumeration Type Documentation

10.15.3.1 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.

10.15.4 Function Documentation

10.15.4.1 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.

10.15.4.2 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 19 of file graph.c.

10.15.4.3 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.

10.16 graph.h

Go to the documentation of this file.

```
00001
00014 #ifndef BRANCHANDBOUND_1TREE_GRAPH_H
00015 #define BRANCHANDBOUND_1TREE_GRAPH_H
00016 #include "./linked_list/linked_list.h"
00017 #include "./linked_list/list_iterator.h"
00018 #include "./linked_list/list_functions.h"
00019 #include "../problem_settings.h"
00020
00021
00023 typedef enum GraphKind{
00024     WEIGHTED_GRAPH,
00025     UNWEIGHTED_GRAPH
00026 } GraphKind;
00027
00028
00030 typedef struct Node {
00031     float x;
00032     float y;
00033     unsigned short positionInGraph;
00034     unsigned short num_neighbours;
00035     unsigned short neighbours [MAX_VERTEX_NUM - 1];
00036 }Node;
00037
00038
00040 typedef struct Edge {
00041     unsigned short src;
00042     unsigned short dest;
00043     unsigned short symbol;
00044     float weight;
00045     float prob;
00046     unsigned short positionInGraph;
00047 }Edge;
00048
00049
00051 typedef struct Graph {
00052     GraphKind kind;
00053     float cost;
00054     unsigned short num_nodes;
00055     unsigned short num_edges;
00056     bool orderedEdges;
00057     Node nodes [MAX_VERTEX_NUM];
00058     Edge edges [MAX_EDGES_NUM];
00059     Edge edges_matrix [MAX_VERTEX_NUM] [MAX_VERTEX_NUM];
00060 }Graph;
00061
00062
00070 void create_graph(Graph* graph, List *  nodes, List *  edges, GraphKind kind);
00071
00072
00078 void create_euclidean_graph(Graph * graph, List * nodes);
00079
00080
00085 void print_graph(const Graph * graph);
00086
00087
00088 #endif //BRANCHANDBOUND_1TREE_GRAPH_H
```

10.17 HybridTSPSolver/src/HybridSolver/main/data_structures/linked_list/linked_list.h File Reference

A double linked list implementation.

```
#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.

10.17.1 Detailed Description

A double linked list implementation.

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

0.1.0

Date

2019-07-9

Copyright

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

Definition in file linked_list.h.

10.17.2 Macro Definition Documentation

10.17.2.1 BRANCHANDBOUND1TREE_LINKED_LIST_H

```
#define BRANCHANDBOUND1TREE_LINKED_LIST_H
```

Definition at line 23 of file linked_list.h.

10.17.3 Typedef Documentation

10.17.3.1 DllElem

```
typedef struct DllElem DllElem
```

The double linked List element.

10.18 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
```

10.19 HybridTSPSolver/src/HybridSolver/main/data_structures/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.

10.19.1 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

0.1.0

Date

2019-07-9

Copyright

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

Definition in file list_functions.c.

10.19.2 Function Documentation

10.19.2.1 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.

10.19.2.2 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.

10.19.2.3 build_dll_elem()

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

Definition at line 46 of file list_functions.c.

10.19.2.4 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.

10.19.2.5 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.

10.19.2.6 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.

10.19.2.7 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.

10.19.2.8 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.

10.19.2.9 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.

10.19.2.10 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.

10.20 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));
00058 }
00059
00060
00061 size_t get_list_size(List *list) {
00062     return (list != NULL) ? list->size : 0;
00063 }
00064
00065
00066 void add_elem_list_bottom(List *list, void *element) {
00067     if (list == NULL) {
00068         return;
00069     }
00070
00071     DllElem *e = build_dll_elem(element, NULL, list->tail);
```

```
00072
00073     if (is_list_empty(list))
00074         list->head = e;
00075     else
00076         list->tail->next = e;
00077     list->tail = e;
00078     list->size++;
00079 }
00080
00081
00082 /*
00083  * This method deletes the element at the indicated index.
00084  * If the index is greater than the size of the List, no element is removed.
00085  */
00086 void add_elem_list_index(List *list, void *element, size_t index) {
00088     if (!list || index > get_list_size(list)) {
00089         return;
00090     }
00091
00092 // support element is a temporary pointer which avoids losing data
00093     DllElem *e;
00094     DllElem *supp = list->head;
00095
00096     for (size_t i = 0; i < index; ++i)
00097         supp = supp->next;
00098
00099     if (supp == list->head) {
00100         e = build_dll_elem(element, supp, NULL);
00101
00102         if (supp == NULL) {
00103             list->head = list->tail = e;
00104         } else {
00105 //          e->next->prev = e;
00106             list->head->prev = e;
00107             list->head = e;
00108         }
00109     } else {
00110         if (supp == NULL) {
00111             e = build_dll_elem(element, NULL, list->tail);
00112             list->tail->next = e;
00113         } else {
00114             e = build_dll_elem(element, supp, supp->prev);
00115             e->next->prev = e;
00116             e->prev->next = e;
00117         }
00118     }
00119
00120     list->size++;
00121 }
00122
00123
00124 /*
00125  * This method deletes the element at the bottom of the List.
00126  */
00127 void delete_list_elem_bottom(List *list) {
00128
00129     if (list == NULL || is_list_empty(list)) {
00130         return;
00131     }
00132
00133     DllElem *oldTail = list->tail;
00134
00135     list->tail = oldTail->prev;
00136     list->tail->next = NULL;
00137
00138     free(oldTail);
00139     list->size--;
00140 }
00141
00142
00143 /*
00144  * This method iteratively finds and deletes the element at the specified index, but only if it
      doesn't exceed
00145  * the size of the List. In this case, instead, no reference gets deleted.
00146  */
00147 void delete_list_elem_index(List *list, size_t index) {
00149     if (list == NULL || is_list_empty(list) || index >= get_list_size(list)) {
00150         return;
00151     }
00152
00153     DllElem *oldElem;
00154     oldElem = list->head;
00155
00156     for (size_t i = 0; i < index; ++i)
00157         oldElem = oldElem->next;
00158
00159     // Found index  to remove!!
```

```
00160     if (oldElem != list->head) {
00161         oldElem->prev->next = oldElem->next;
00162         if (oldElem->next != NULL) {
00163             oldElem->next->prev = oldElem->prev;
00164         } else {
00165             list->tail = oldElem->prev;
00166         }
00167     } else {
00168         if (list->head == list->tail) {
00169             list->head = list->tail = NULL;
00170         } else {
00171             list->head = list->head->next;
00172             list->head->prev = NULL;
00173         }
00174     }
00175
00176     free(oldElem);
00177     list->size--;
00178 }
00179
00180
00181 /*
00182  * This method iteratively runs through the dllist elements and returns the one at the requested
    index.
00183  * If the index exceeds the size of the List, we instead return no element.
00184  */
00185 void *get_list_elem_index(List *list, size_t index) {
00186     if (list == NULL  || index >= get_list_size(list)) {
00187         return NULL;
00188     }
00189
00190     DllElem *supp; // iteration support element
00191     supp = list->head;
00192
00193     for (size_t i = 0; i < index; ++i)
00194         supp = supp->next;
00195     return supp->value;
00196 }
```

10.21 HybridTSPSolver/src/HybridSolver/main/data_structures/linked_↩ list/list_functions.h File Reference

The declaration of the functions to manipulate the List.

```
#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.

10.21.1 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

0.1.0

Date

2019-07-9

Copyright

Copyright (c) 2023, license MIT

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

Definition in file list_functions.h.

10.21.2 Function Documentation

10.21.2.1 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.

10.21.2.2 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.

10.21.2.3 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.

10.21.2.4 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.

10.21.2.5 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.

10.21.2.6 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.

10.21.2.7 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.

10.21.2.8 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.

10.21.2.9 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.

10.22 list_functions.h

Go to the documentation of this file.

```
00001
00014 #ifndef BRANCHANDBOUND1TREE_LIST_FUNCTIONS_H
00015 #define BRANCHANDBOUND1TREE_LIST_FUNCTIONS_H
00016
00017 #include "linked_list.h"
00018
00019
00024 List *new_list(void);
00025
00026
00032 void del_list(List *list);
00033
00034
00040 bool is_list_empty(List *list);
00041
00042
00048 size_t get_list_size(List *list);
00049
00050
00056 void add_elem_list_bottom(List *list, void *element);
00057
00058
00065 void add_elem_list_index(List *array, void *element, size_t index);
00066
00067
00072 void delete_list_elem_bottom(List *list);
00073
00074
00080 void delete_list_elem_index(List *list, size_t index);
00081
00082
00089 void *get_list_elem_index(List *list, size_t index);
00090
00091
00092 #endif //BRANCHANDBOUND1TREE_LIST_FUNCTIONS_H
```

10.23 HybridTSPSolver/src/HybridSolver/main/data_structures/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.

10.23.1 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

0.1.0

Date

2019-07-9

Copyright

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

Definition in file list_iterator.c.

10.23.2 Function Documentation

10.23.2.1 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.

10.23.2.2 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.

10.23.2.3 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.

10.23.2.4 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.

10.23.2.5 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.

10.23.2.6 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.

10.24 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 }
```

```
00028
00029
00030 void *get_current_list_iterator_element(ListIterator *iterator) {
00031     return (iterator && iterator->curr && iterator->curr->value) ? iterator->curr->value : NULL;
00032 }
00033
00034
00035 bool is_list_iterator_valid(ListIterator *iterator) {
00036     return (iterator) ? iterator->index < get_list_size(iterator->list) : 0;
00037 }
00038
00039
00040 void list_iterator_next(ListIterator *iterator) {
00041     if (is_list_iterator_valid(iterator)) {
00042         iterator->index++;
00043
00044         if (is_list_iterator_valid(iterator)) {
00045             iterator->curr = iterator->curr->next;
00046         }
00047     }
00048 }
00049
00050
00051 void delete_list_iterator(ListIterator *iterator) {
00052     free(iterator);
00053 }
00054
00055
00056 void *list_iterator_get_next(ListIterator *iterator) {
00057     if (!is_list_iterator_valid(iterator)) {
00058         return NULL;
00059     }
00060
00061     void *element = get_current_list_iterator_element(iterator);
00062     list_iterator_next(iterator);
00063     return element;
00064 }
```

10.25 HybridTSPSolver/src/HybridSolver/main/data_structures/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.

10.25.1 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

0.1.0

Date

2019-07-9

Copyright

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

Definition in file list_iterator.h.

10.25.2 Function Documentation

10.25.2.1 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.

10.25.2.2 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.

10.25.2.3 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.

10.25.2.4 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.

10.25.2.5 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.

10.25.2.6 list_iterator_next()

```
void list_iterator_next (
            ListIterator * iterator )
```

Used to move the ListIterator to the next value of the object.

Parameters

l	The ListIterator considered.

Definition at line 40 of file list_iterator.c.

10.26 list_iterator.h

Go to the documentation of this file.

```
00001
00015 #ifndef BRANCHANDBOUND1TREE_LIST_ITERATOR_H
00016 #define BRANCHANDBOUND1TREE_LIST_ITERATOR_H
00017 #include "linked_list.h"
00018
00019
00025 ListIterator *create_list_iterator(List *list);
00026
00027
00033 bool is_list_iterator_valid(ListIterator *iterator);
00034
00035
00041 void *get_current_list_iterator_element(ListIterator *iterator);
00042
```

```
00043
00048 void list_iterator_next(ListIterator *iterator);
00049
00050
00056 void *list_iterator_get_next(ListIterator *iterator);
00057
00058
00063 void delete_list_iterator(ListIterator *iterator);
00064
00065
00066 #endif //BRANCHANDBOUND1TREE_LIST_ITERATOR_H
```

10.27 HybridTSPSolver/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.

10.27.1 Detailed Description

This file contains the implementation of the Merge-Find Set datastructure for the Minimum Spanning Tree problem.

Author

Lorenzo Sciandra

Version

0.1.0

Date

2023-04-18

Copyright

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

Definition in file mfset.c.

10.27.2 Function Documentation

10.27.2.1 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.

10.27.2.2 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.
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.

10.27.2.3 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.

10.27.2.4 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.

10.27.2.5 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.

10.28 mfset.c

Go to the documentation of this file.

```
00001
00014 #include "mfset.h"
00015
00016
00017 void create_forest_constrained(Forest *forest, const Node *nodes, unsigned short num_nodes, unsigned
      short candidateId) {
00018     forest->num_sets = num_nodes - 1;
00019
00020     for (unsigned short i = 0; i < num_nodes; i++) {
00021         if (i != candidateId) {
00022             forest->sets[i].parentSet = NULL;
00023             forest->sets[i].rango = 0;
00024             forest->sets[i].curr = nodes[i];
00025             forest->sets[i].num_in_forest = i;
00026         }
00027     }
00028 }
00029
00030
00031 void create_forest(Forest *forest, const Node *nodes, unsigned short num_nodes) {
00032
00033     forest->num_sets = num_nodes;
00034     for (unsigned short i = 0; i < num_nodes; i++) {
00035         forest->sets[i].parentSet = NULL;
00036         forest->sets[i].rango = 0;
00037         forest->sets[i].curr = nodes[i];
00038         forest->sets[i].num_in_forest = i;
00039     }
00040
00041 }
00042
00043
00044 Set *find(Set *set) {
00045     if (set->parentSet != NULL) {
00046         set->parentSet = find(set->parentSet);
00047         return set->parentSet;
00048     }
00049     return set;
00050 }
00051
00052
00053 void merge(Set *set1, Set *set2) {
00054
00055     Set *set1_root = find(set1);
00056     Set *set2_root = find(set2);
00057
00058     //printf("\nThe root are %.2fd ,%d\n", set1_root->num_in_forest, set2_root->num_in_forest);
00059     if (set1_root->num_in_forest != set2_root->num_in_forest) {
00060         if (set1_root->rango > set2_root->rango) {
00061             set2_root->parentSet = set1_root;
00062         } else if (set1_root->rango < set2_root->rango) {
00063             set1_root->parentSet = set2_root;
00064         } else {
00065             set2_root->parentSet = set1_root;
00066             set1_root->rango++;
00067         }
00068     }
00069 }
00070
00071
00072 void print_forest(const Forest *forest) {
00073     for (unsigned short i = 0; i < forest->num_sets; i++) {
00074         Set set = forest->sets[i];
00075
00076         printf("Set %i: ", set.curr.positionInGraph);
00077         if (set.parentSet != NULL) {
00078             printf("Parent: %i, ", set.parentSet->curr.positionInGraph);
00079         } else {
00080             printf("Parent: NULL, ");
00081         }
00082         printf("Rango: %d, ", set.rango);
00083         printf("Num in forest: %d\n", set.num_in_forest);
00084
00085     }
00086 }
```

10.29 HybridTSPSolver/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.

10.29.1 Detailed Description

This file contains the declaration of the Merge-Find Set datastructure for the Minimum Spanning Tree problem.

Author

Lorenzo Sciandra

Version

0.1.0

Date

2023-04-18

Copyright

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

Definition in file mfset.h.

10.29.2 Typedef Documentation

10.29.2.1 Forest

```
typedef struct Forest Forest
```

A Forest is a list of Sets.

10.29.2.2 Set

```
typedef struct Set Set
```

A Set is a node in the Forest.

10.29.3 Function Documentation

10.29.3.1 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.

10.29.3.2 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.

10.29.3.3 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.

10.29.3.4 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.

10.29.3.5 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.

10.30 mfset.h

Go to the documentation of this file.

```
00001
00013 #ifndef BRANCHANDBOUND1TREE_MFSET_H
00014 #define BRANCHANDBOUND1TREE_MFSET_H
00015 #include "graph.h"
00016
00017
00019 typedef struct Set {
00020     struct Set * parentSet;
00021     unsigned short rango;
00022     Node curr;
00023     unsigned short num_in_forest;
00024 }Set;
00025
00026
00028 typedef struct Forest {
00029     unsigned short num_sets;
00030     Set sets [MAX_VERTEX_NUM];
00031 }Forest;
00032
00033
00040 void create_forest(Forest * forest, const Node * nodes, unsigned short num_nodes);
00041
00042
00051 void create_forest_constrained(Forest * forest, const Node * nodes, unsigned short num_nodes, unsigned
      short candidateId);
00052
00053
00060 void merge(Set * set1, Set * set2);
00061
00062
00069 Set* find(Set * set);
00070
00071
00076 void print_forest(const Forest * forest);
00077
00078
00079 #endif //BRANCHANDBOUND1TREE_MFSET_H
```

10.31 HybridTSPSolver/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.

10.31.1 Detailed Description

This file contains the definition of the Minimum Spanning Tree operations.

Author

Lorenzo Sciandra

Version

0.1.0

Date

2023-04-18

Copyright

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

Definition in file mst.c.

10.31.2 Function Documentation

10.31.2.1 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 33 of file mst.c.

10.31.2.2 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.

10.31.2.3 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 63 of file mst.c.

10.31.2.4 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 85 of file mst.c.

10.32 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 }
00031
00032
00033 void add_edge(MST * tree, const Edge * edge){
00034
00035     unsigned short src = edge->src;
00036     unsigned short dest = edge->dest;
00037
00038     tree->edges[tree->num_edges].src = src;
00039     tree->edges[tree->num_edges].dest = dest;
00040     tree->edges[tree->num_edges].weight = edge->weight;
00041     tree->edges[tree->num_edges].symbol = edge->symbol;
00042     tree->edges[tree->num_edges].prob = edge->prob;
00043     tree->edges[tree->num_edges].positionInGraph = tree->num_edges;
00044     tree->nodes[src].neighbours[tree->nodes[src].num_neighbours] = dest;
00045     tree->nodes[src].num_neighbours++;
00046     tree->nodes[dest].neighbours[tree->nodes[dest].num_neighbours] = src;
00047     tree->nodes[dest].num_neighbours++;
00048
00049     tree->num_edges++;
00050     tree->cost += edge->weight;
00051
00052     if(HYBRID){
00053         if(tree->num_edges == 1){
00054             tree->prob = edge->prob;
00055         }
00056         else{
00057             tree->prob  = ((tree->prob * ((float) tree->num_edges -1)) + edge->prob) / ((float)
      tree->num_edges);
00058         }
00059     }
00060 }
00061
00062
00063 void print_mst(const MST * tree){
00064     printf("\nMST or 1-Tree with cost: %.2lf and validity = %s\n", tree->cost, tree->isValid ? "TRUE"
      : "FALSE");
00065
00066     double dim = (log(tree->num_nodes) / log(10) + 1) * 2 + 7;
00067     for (unsigned short i = 0; i < tree->num_edges; i++) {
00068         char edge_print [(int) dim] ;
00069         char edge_print_dest [(int) (dim-7)/2] ;
```

```
00070         const Edge * curr = &tree->edges[i];
00071         sprintf(edge_print, "%i", curr->src);
00072         strcat(edge_print, " <--> ");
00073         sprintf(edge_print_dest, "%i", curr->dest);
00074         strcat(edge_print, edge_print_dest);
00075         printf("Edge%i:\t%s\tweight = %.2lf\tprob = %lf\n",
00076                curr->symbol,
00077                edge_print,
00078                curr->weight,
00079                curr->prob);
00080     }
00081
00082 }
00083
00084
00085 void print_mst_original_weight(const MST * tree, const Graph * graph){
00086     printf("\nMST or 1-Tree with cost: %f and validity = %s\n", tree->cost, tree->isValid ? "TRUE" :
      "FALSE");
00087
00088     double dim = (log(tree->num_nodes) / log(10) + 1) * 2 + 7;
00089     for (unsigned short i = 0; i < tree->num_edges; i++) {
00090         char edge_print [(int) dim] ;
00091         char edge_print_dest [(int) (dim-7)/2] ;
00092         const Edge * curr = &tree->edges[i];
00093         sprintf(edge_print, "%i", curr->src);
00094         strcat(edge_print, " <--> ");
00095         sprintf(edge_print_dest, "%i", curr->dest);
00096         strcat(edge_print, edge_print_dest);
00097         printf("Edge%i: %s weight = %f prob = %f\n",
00098                curr->symbol,
00099                edge_print,
00100                graph->edges_matrix[curr->src][curr->dest].weight,
00101                curr->prob);
00102     }
00103 }
```

10.33 HybridTSPSolver/src/HybridSolver/main/data_structures/mst.h File Reference

This file contains the declaration of the Minimum Spanning Tree datastructure.

```
#include "mfset.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.

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.

10.33.1 Detailed Description

This file contains the declaration of the Minimum Spanning Tree datastructure.

Author

Lorenzo Sciandra

Version

0.1.0

Date

2023-04-18

Copyright

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

Definition in file mst.h.

10.33.2 Typedef Documentation

10.33.2.1 ConstrainedEdge

```
typedef struct ConstrainedEdge ConstrainedEdge
```

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

10.33.2.2 MST

```
typedef struct MST MST
```

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

10.33.3 Function Documentation

10.33.3.1 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 33 of file mst.c.

10.33.3.2 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.

10.33.3.3 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 63 of file mst.c.

10.33.3.4 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 85 of file mst.c.

10.34 mst.h

Go to the documentation of this file.

```
00001
00014 #ifndef BRANCHANDBOUND1TREE_MST_H
00015 #define BRANCHANDBOUND1TREE_MST_H
00016 #include "mfset.h"
00017
00018
00020 typedef struct ConstrainedEdge{
00021     unsigned short src;
00022     unsigned short dest;
00023 }ConstrainedEdge;
00024
00025
00027 typedef struct MST{
00028     bool isValid;
00029     float cost;
00030     float prob;
00031     unsigned short num_nodes;
00032     unsigned short num_edges;
00033     Node nodes [MAX_VERTEX_NUM];
00034     Edge edges [MAX_VERTEX_NUM];
00035 }MST;
```

```
00036
00037
00044 void create_mst(MST* mst, const Node * nodes, unsigned short num_nodes);
00045
00046
00052 void add_edge(MST * tree, const Edge * edge);
00053
00054
00059 void print_mst(const MST * mst);
00060
00061
00068 void print_mst_original_weight(const MST * mst, const Graph * graph);
00069
00070
00071 #endif //BRANCHANDBOUND1TREE_MST_H
```

10.35 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/config.py File Reference

Classes

- class config.Settings

Namespaces

- namespace config

Functions

- def config.get_default_config ()
- def config.get_config (filepath)

10.36 config.py

Go to the documentation of this file.

```
00001 import json
00002
00003
00004 class Settings(dict):
00005     """Experiment configuration options.
00006
00007     Wrapper around in-built dict class to access members through the dot operation.
00008
00009     Experiment parameters:
00010         "expt_name": Name/description of experiment, used for logging.
00011         "gpu_id": Available GPU ID(s)
00012
00013         "train_filepath": Training set path
00014         "val_filepath": Validation set path
00015         "test_filepath": Test set path
00016
00017         "num_nodes": Number of nodes in TSP tours
00018         "num_neighbors": Number of neighbors in k-nearest neighbor input graph (-1 for fully
      connected)
00019
00020         "node_dim": Number of dimensions for each node
00021         "voc_nodes_in": Input node signal vocabulary size
00022         "voc_nodes_out": Output node prediction vocabulary size
00023         "voc_edges_in": Input edge signal vocabulary size
00024         "voc_edges_out": Output edge prediction vocabulary size
00025
00026         "beam_size": Beam size for beamsearch procedure (-1 for disabling beamsearch)
00027
00028         "hidden_dim": Dimension of model's hidden state
```

```
00029         "num_layers": Number of GCN layers
00030         "mlp_layers": Number of MLP layers
00031         "aggregation": Node aggregation scheme in GCN (`mean` or `sum`)
00032 
00033         "max_epochs": Maximum training epochs
00034         "val_every": Interval (in epochs) at which validation is performed
00035         "test_every": Interval (in epochs) at which testing is performed
00036 
00037         "batch_size": Batch size
00038         "batches_per_epoch": Batches per epoch (-1 for using full training set)
00039         "accumulation_steps": Number of steps for gradient accumulation (DO NOT USE: BUGGY)
00040 
00041         "learning_rate": Initial learning rate
00042         "decay_rate": Learning rate decay parameter
00043     """
00044 
00045     def __init__(self, config_dict):
00046         super().__init__()
00047         for key in config_dict:
00048             self[key] = config_dict[key]
00049 
00050     def __getattr__(self, attr):
00051         return self[attr]
00052 
00053     def __setitem__(self, key, value):
00054         return super().__setitem__(key, value)
00055 
00056     def __setattr__(self, key, value):
00057         return self.__setitem__(key, value)
00058 
00059     __delattr__ = dict.__delitem__
00060 
00061 
00062 def get_default_config():
00063     """Returns default settings object.
00064     """
00065     return Settings(json.load(open("./configs/default.json")))
00066 
00067 
00068 def get_config(filepath):
00069     """Returns settings from json file.
00070     """
00071     config = get_default_config()
00072     config.update(Settings(json.load(open(filepath))))
00073     return config
```

10.37 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/main.py File Reference

Namespaces

- namespace main

Functions

- def main.compute_prob (net, config, dtypeLong, dtypeFloat, instance_number)
- def main.write_adjacency_matrix (y_probs, x_edges_values, filepath)
- def main.main (filepath, num_nodes, instance_number)

Variables

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

10.38 main.py

Go to the documentation of this file.

```
00001 """
00002     @file main.py
00003     @author Lorenzo Sciandra, by Chaitanya K. Joshi, Thomas Laurent and Xavier Bresson.
00004     @brief A recombination of code take from: https://github.com/chaitjo/graph-convnet-tsp.
00005     Some functions were created for the purpose of this project.
00006     @version 0.1.0
00007     @date 2023-04-18
00008     @copyright Copyright (c) 2023, license MIT
00009     Repo: https://github.com/LorenzoSciandra/HybridTSPSolver
00010 """
00011
00012
00013 import os
00014 import sys
00015 import time
00016 import numpy as np
00017 import torch
00018 from torch.autograd import Variable
00019 import torch.nn.functional as F
00020 import torch.nn as nn
00021 from sklearn.utils.class_weight import compute_class_weight
00022 # Remove warning
00023 import warnings
00024 warnings.filterwarnings("ignore", category=UserWarning)
00025 from scipy.sparse import SparseEfficiencyWarning
00026 warnings.simplefilter('ignore', SparseEfficiencyWarning)
00027 from config import *
00028 from utils.graph_utils import *
00029 from utils.google_tsp_reader import GoogleTSPReader
00030 from utils.plot_utils import *
00031 from models.gcn_model import ResidualGatedGCNModel
00032 from utils.model_utils import *
00033
00034
00035 def compute_prob(net, config, dtypeLong, dtypeFloat, instance_number):
00036     """
00037     This function computes the probability of the edges being in the optimal tour, by running the GCN.
00038     Args:
00039         net: The Graph Convolutional Network.
00040         config: The configuration file, from which the parameters are taken.
00041         dtypeLong: The data type for the long tensors.
00042         dtypeFloat: The data type for the float tensors.
00043         instance_number: The number of the instance to be computed.
00044     Returns:
00045         y_probs: The probability of the edges being in the optimal tour.
00046         x_edges_values: The distance between the nodes.
00047     """
00048     # Set evaluation mode
00049     net.eval()
00050
00051     # Assign parameters
00052     num_nodes = config.num_nodes
00053     num_neighbors = config.num_neighbors
00054     batch_size = config.batch_size
00055     test_filepath = config.test_filepath
00056
00057     # Load TSP data
00058     dataset = GoogleTSPReader(num_nodes, num_neighbors, batch_size=batch_size, filepath=test_filepath)
00059
00060     # Convert dataset to iterable
00061     dataset = iter(dataset)
00062
00063     # Initially set loss class weights as None
00064     edge_cw = None
00065
00066     y_probs = []
00067
00068     # read the instance number line from the test_filepath
00069     instance = None
00070     with open(test_filepath, 'r') as f:
00071         for i, line in enumerate(f):
00072             if i == instance_number:
00073                 instance = line
00074                 break
00075
00076     # split the instance before the "output" part
00077     instance = instance.split(" output")[0]
00078     # create a list of the nodes spliting by spaces and convert to double
00079     instance = [float(x) for x in instance.split(" ")]
00080
00081     with torch.no_grad():
00082
```

```
00083         batch = next(dataset)
00084
00085         while batch.nodes_coord.flatten().tolist() != instance:
00086             batch = next(dataset)
00087
00088         x_edges = Variable(torch.LongTensor(batch.edges).type(dtypeLong), requires_grad=False)
00089         x_edges_values = Variable(torch.FloatTensor(batch.edges_values).type(dtypeFloat),
      requires_grad=False)
00090         x_nodes = Variable(torch.LongTensor(batch.nodes).type(dtypeLong), requires_grad=False)
00091         x_nodes_coord = Variable(torch.FloatTensor(batch.nodes_coord).type(dtypeFloat),
      requires_grad=False)
00092         y_edges = Variable(torch.LongTensor(batch.edges_target).type(dtypeLong), requires_grad=False)
00093
00094         # Compute class weights (if uncomputed)
00095         if type(edge_cw) != torch.Tensor:
00096             edge_labels = y_edges.cpu().numpy().flatten()
00097             edge_cw = compute_class_weight("balanced", classes=np.unique(edge_labels), y=edge_labels)
00098
00099         y_preds, _ = net.forward(x_edges, x_edges_values, x_nodes, x_nodes_coord, y_edges, edge_cw)
00100         y = F.softmax(y_preds, dim=3)
00101         # y_bins = y.argmax(dim=3)
00102         y_probs = y[:, :, :, 1]
00103
00104     return y_probs, x_edges_values
00105
00106
00107 def write_adjacency_matrix(y_probs, x_edges_values, filepath):
00108     """
00109     This function simply writes the probabilistic adjacency matrix in a file, where each cell
00110     is a tuple (distance, probability).
00111     Args:
00112         y_probs: The probability of the edges being in the optimal tour.
00113         x_edges_values: The distance between the nodes.
00114         filepath: The path to the file where the adjacency matrix will be written.
00115     """
00116     # Convert to numpy
00117     num_nodes = y_probs.shape[1]
00118     y_probs = y_probs.flatten().numpy()
00119     x_edges_values = x_edges_values.flatten().numpy()
00120
00121     # stack the arrays horizontally and convert to string data type
00122     arr_combined = np.stack((x_edges_values, y_probs), axis=1).astype('U')
00123
00124     # format the strings using a list comprehension
00125     arr_strings = np.array(['({}, {});'.format(x[0], x[1]) for x in arr_combined])
00126
00127     filepath = filepath.replace(".csv", "_temp.csv")
00128     # write arr_strings to file
00129     with open(filepath, 'w') as f:
00130         edge = 0
00131         for item in arr_strings:
00132             if (edge + 1) % num_nodes == 0:
00133                 f.write("%s\n" % item)
00134             else:
00135                 f.write("%s" % item)
00136             edge += 1
00137
00138
00139 def main(filepath, num_nodes, instance_number):
00140     """
00141     The function that calls the previous functions and first sets the parameters for the calculation.
00142     Args:
00143         filepath: The path to the file where the adjacency matrix will be written.
00144         num_nodes: The number of nodes in the TSP instance.
00145         instance_number: The number of the instance to be computed.
00146     """
00147     config_path = "./logs/tsp" + num_nodes + "/config.json"
00148     config = get_config(config_path)
00149
00150     config.gpu_id = "0"
00151     config.accumulation_steps = 1
00152     config.val_filepath = "./data/hyb_tsp_" + num_nodes + "/test_100_instances.txt"
00153     config.test_filepath = "./data/hyb_tsp_" + num_nodes + "/test_100_instances.txt"
00154
00155     os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
00156     os.environ["CUDA_VISIBLE_DEVICES"] = str(config.gpu_id)
00157
00158     if torch.cuda.is_available():
00159         # print("CUDA available, using GPU ID {}".format(config.gpu_id))
00160         dtypeFloat = torch.cuda.FloatTensor
00161         dtypeLong = torch.cuda.LongTensor
00162         torch.cuda.manual_seed(1)
00163     else:
00164         # print("CUDA not available")
00165         dtypeFloat = torch.FloatTensor
00166         dtypeLong = torch.LongTensor
00167         torch.manual_seed(1)
```

```
00168
00169     net = nn.DataParallel(ResidualGatedGCNModel(config, dtypeFloat, dtypeLong))
00170     if torch.cuda.is_available():
00171         net.cuda()
00172
00173     log_dir = f"./logs/{config.expt_name}/"
00174     if torch.cuda.is_available():
00175         checkpoint = torch.load(log_dir + "best_val_checkpoint.tar")
00176     else:
00177         checkpoint = torch.load(log_dir + "best_val_checkpoint.tar", map_location='cpu')
00178     # Load network state
00179     net.load_state_dict(checkpoint['model_state_dict'])
00180     config.batch_size = 1
00181     probs, edges_value = compute_prob(net, config, dtypeLong, dtypeFloat, instance_number)
00182     write_adjacency_matrix(probs, edges_value, filepath)
00183
00184
00185 if __name__ == "__main__":
00186     """
00187     Args:
00188         sys.argv[1]: The path to the file where the adjacency matrix will be written.
00189         sys.argv[2]: The number of nodes in the TSP instance.
00190         sys.argv[3]: The number of the instance to be computed.
00191     """
00192     if len(sys.argv) != 4:
00193         print("\nPlease provide the path to the output file to write in, the number of nodes in the
      tsp and the "
00194               "instance number to analyze. The format is: "
00195               "<filepath> <number of nodes> <instance number>\n")
00196         sys.exit(1)
00197
00198     if not isinstance(sys.argv[1], str) or not isinstance(sys.argv[2], str) or not
      isinstance(sys.argv[3], str):
00199         print("Error: The arguments must be strings.")
00200         sys.exit(1)
00201
00202     filepath = sys.argv[1]
00203     num_nodes = sys.argv[2]
00204     instance_number = int(sys.argv[3]) - 1
00205
00206     main(filepath, num_nodes, instance_number)
```

10.39 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/models/gcn_layers.py File Reference

Classes

- class models.gcn_layers.BatchNormNode
- class models.gcn_layers.BatchNormEdge
- class models.gcn_layers.NodeFeatures
- class models.gcn_layers.EdgeFeatures
- class models.gcn_layers.ResidualGatedGCNLayer
- class models.gcn_layers.MLP

Namespaces

- namespace models
- namespace models.gcn_layers

10.40 gcn_layers.py

Go to the documentation of this file.

```
00001 import torch
00002 import torch.nn.functional as F
00003 import torch.nn as nn
```

```
00004
00005 import numpy as np
00006
00007
00008 class BatchNormNode(nn.Module):
00009     """Batch normalization for node features.
00010     """
00011
00012     def __init__(self, hidden_dim):
00013         super(BatchNormNode, self).__init__()
00014         self.batch_norm = nn.BatchNorm1d(hidden_dim, track_running_stats=False)
00015
00016     def forward(self, x):
00017         """
00018         Args:
00019             x: Node features (batch_size, num_nodes, hidden_dim)
00020
00021         Returns:
00022             x_bn: Node features after batch normalization (batch_size, num_nodes, hidden_dim)
00023         """
00024         x_trans = x.transpose(1, 2).contiguous()  # Reshape input: (batch_size, hidden_dim, num_nodes)
00025         x_trans_bn = self.batch_norm(x_trans)
00026         x_bn = x_trans_bn.transpose(1, 2).contiguous()  # Reshape to original shape
00027         return x_bn
00028
00029
00030 class BatchNormEdge(nn.Module):
00031     """Batch normalization for edge features.
00032     """
00033
00034     def __init__(self, hidden_dim):
00035         super(BatchNormEdge, self).__init__()
00036         self.batch_norm = nn.BatchNorm2d(hidden_dim, track_running_stats=False)
00037
00038     def forward(self, e):
00039         """
00040         Args:
00041             e: Edge features (batch_size, num_nodes, num_nodes, hidden_dim)
00042
00043         Returns:
00044             e_bn: Edge features after batch normalization (batch_size, num_nodes, num_nodes,
      hidden_dim)
00045         """
00046         e_trans = e.transpose(1, 3).contiguous()  # Reshape input: (batch_size, num_nodes, num_nodes,
      hidden_dim)
00047         e_trans_bn = self.batch_norm(e_trans)
00048         e_bn = e_trans_bn.transpose(1, 3).contiguous()  # Reshape to original
00049         return e_bn
00050
00051
00052 class NodeFeatures(nn.Module):
00053     """Convnet features for nodes.
00054
00055     Using `sum` aggregation:
00056         x_i = U*x_i +  sum_j [ gate_ij * (V*x_j) ]
00057
00058     Using `mean` aggregation:
00059         x_i = U*x_i + ( sum_j [ gate_ij * (V*x_j) ] / sum_j [ gate_ij] )
00060     """
00061
00062     def __init__(self, hidden_dim, aggregation="mean"):
00063         super(NodeFeatures, self).__init__()
00064         self.aggregation = aggregation
00065         self.U = nn.Linear(hidden_dim, hidden_dim, True)
00066         self.V = nn.Linear(hidden_dim, hidden_dim, True)
00067
00068     def forward(self, x, edge_gate):
00069         """
00070         Args:
00071             x: Node features (batch_size, num_nodes, hidden_dim)
00072             edge_gate: Edge gate values (batch_size, num_nodes, num_nodes, hidden_dim)
00073
00074         Returns:
00075             x_new: Convolved node features (batch_size, num_nodes, hidden_dim)
00076         """
00077         Ux = self.U(x)  # B x V x H
00078         Vx = self.V(x)  # B x V x H
00079         Vx = Vx.unsqueeze(1)  # extend Vx from "B x V x H" to "B x 1 x V x H"
00080         gateVx = edge_gate * Vx  # B x V x V x H
00081         if self.aggregation=="mean":
00082             x_new = Ux + torch.sum(gateVx, dim=2) / (1e-20 + torch.sum(edge_gate, dim=2))  # B x V x H
00083         elif self.aggregation=="sum":
00084             x_new = Ux + torch.sum(gateVx, dim=2)  # B x V x H
00085         return x_new
00086
00087
00088 class EdgeFeatures(nn.Module):
```

```
00089     """Convnet features for edges.
00090
00091     e_ij = U*e_ij + V*(x_i + x_j)
00092     """
00093
00094     def __init__(self, hidden_dim):
00095         super(EdgeFeatures, self).__init__()
00096         self.U = nn.Linear(hidden_dim, hidden_dim, True)
00097         self.V = nn.Linear(hidden_dim, hidden_dim, True)
00098
00099     def forward(self, x, e):
00100         """
00101         Args:
00102             x: Node features (batch_size, num_nodes, hidden_dim)
00103             e: Edge features (batch_size, num_nodes, num_nodes, hidden_dim)
00104
00105         Returns:
00106             e_new: Convolved edge features (batch_size, num_nodes, num_nodes, hidden_dim)
00107         """
00108         Ue = self.U(e)
00109         Vx = self.V(x)
00110         Wx = Vx.unsqueeze(1)  # Extend Vx from "B x V x H" to "B x V x 1 x H"
00111         Vx = Vx.unsqueeze(2)  # extend Vx from "B x V x H" to "B x 1 x V x H"
00112         e_new = Ue + Vx + Wx
00113         return e_new
00114
00115
00116 class ResidualGatedGCNLayer(nn.Module):
00117     """Convnet layer with gating and residual connection.
00118     """
00119
00120     def __init__(self, hidden_dim, aggregation="sum"):
00121         super(ResidualGatedGCNLayer, self).__init__()
00122         self.node_feat = NodeFeatures(hidden_dim, aggregation)
00123         self.edge_feat = EdgeFeatures(hidden_dim)
00124         self.bn_node = BatchNormNode(hidden_dim)
00125         self.bn_edge = BatchNormEdge(hidden_dim)
00126
00127     def forward(self, x, e):
00128         """
00129         Args:
00130             x: Node features (batch_size, num_nodes, hidden_dim)
00131             e: Edge features (batch_size, num_nodes, num_nodes, hidden_dim)
00132
00133         Returns:
00134             x_new: Convolved node features (batch_size, num_nodes, hidden_dim)
00135             e_new: Convolved edge features (batch_size, num_nodes, num_nodes, hidden_dim)
00136         """
00137         e_in = e
00138         x_in = x
00139         # Edge convolution
00140         e_tmp = self.edge_feat(x_in, e_in)  # B x V x V x H
00141         # Compute edge gates
00142         edge_gate = F.sigmoid(e_tmp)
00143         # Node convolution
00144         x_tmp = self.node_feat(x_in, edge_gate)
00145         # Batch normalization
00146         e_tmp = self.bn_edge(e_tmp)
00147         x_tmp = self.bn_node(x_tmp)
00148         # ReLU Activation
00149         e = F.relu(e_tmp)
00150         x = F.relu(x_tmp)
00151         # Residual connection
00152         x_new = x_in + x
00153         e_new = e_in + e
00154         return x_new, e_new
00155
00156
00157 class MLP(nn.Module):
00158     """Multi-layer Perceptron for output prediction.
00159     """
00160
00161     def __init__(self, hidden_dim, output_dim, L=2):
00162         super(MLP, self).__init__()
00163         self.L = L
00164         U = []
00165         for layer in range(self.L - 1):
00166             U.append(nn.Linear(hidden_dim, hidden_dim, True))
00167         self.U = nn.ModuleList(U)
00168         self.V = nn.Linear(hidden_dim, output_dim, True)
00169
00170     def forward(self, x):
00171         """
00172         Args:
00173             x: Input features (batch_size, hidden_dim)
00174
00175         Returns:
```

```
00176             y: Output predictions (batch_size, output_dim)
00177         """
00178         Ux = x
00179         for U_i in self.U:
00180             Ux = U_i(Ux)  # B x H
00181             Ux = F.relu(Ux)  # B x H
00182         y = self.V(Ux)  # B x O
00183         return y
```

10.41 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/models/gcn_model.py File Reference

Classes

- class models.gcn_model.ResidualGatedGCNModel

Namespaces

- namespace models
- namespace models.gcn_model

10.42 gcn_model.py

Go to the documentation of this file.

```
00001 import torch
00002 import torch.nn.functional as F
00003 import torch.nn as nn
00004
00005 from models.gcn_layers import ResidualGatedGCNLayer, MLP
00006 from utils.model_utils import *
00007
00008
00009 class ResidualGatedGCNModel(nn.Module):
00010     """Residual Gated GCN Model for outputting predictions as edge adjacency matrices.
00011
00012     References:
00013         Paper: https://arxiv.org/pdf/1711.07553v2.pdf
00014         Code: https://github.com/xbresson/spatial_graph_convnets
00015     """
00016
00017     def __init__(self, config, dtypeFloat, dtypeLong):
00018         super(ResidualGatedGCNModel, self).__init__()
00019         self.dtypeFloat = dtypeFloat
00020         self.dtypeLong = dtypeLong
00021         # Define net parameters
00022         self.num_nodes = config.num_nodes
00023         self.node_dim = config.node_dim
00024         self.voc_nodes_in = config['voc_nodes_in']
00025         self.voc_nodes_out = config['num_nodes']  # config['voc_nodes_out']
00026         self.voc_edges_in = config['voc_edges_in']
00027         self.voc_edges_out = config['voc_edges_out']
00028         self.hidden_dim = config['hidden_dim']
00029         self.num_layers = config['num_layers']
00030         self.mlp_layers = config['mlp_layers']
00031         self.aggregation = config['aggregation']
00032         # Node and edge embedding layers/lookups
00033         self.nodes_coord_embedding = nn.Linear(self.node_dim, self.hidden_dim, bias=False)
00034         self.edges_values_embedding = nn.Linear(1, self.hidden_dim//2, bias=False)
00035         self.edges_embedding = nn.Embedding(self.voc_edges_in, self.hidden_dim//2)
00036         # Define GCN Layers
00037         gcn_layers = []
00038         for layer in range(self.num_layers):
00039             gcn_layers.append(ResidualGatedGCNLayer(self.hidden_dim, self.aggregation))
00040         self.gcn_layers = nn.ModuleList(gcn_layers)
00041         # Define MLP classifiers
00042         self.mlp_edges = MLP(self.hidden_dim, self.voc_edges_out, self.mlp_layers)
00043         # self.mlp_nodes = MLP(self.hidden_dim, self.voc_nodes_out, self.mlp_layers)
00044
```

```
00045     def forward(self, x_edges, x_edges_values, x_nodes, x_nodes_coord, y_edges, edge_cw):
00046         """
00047         Args:
00048             x_edges: Input edge adjacency matrix (batch_size, num_nodes, num_nodes)
00049             x_edges_values: Input edge distance matrix (batch_size, num_nodes, num_nodes)
00050             x_nodes: Input nodes (batch_size, num_nodes)
00051             x_nodes_coord: Input node coordinates (batch_size, num_nodes, node_dim)
00052             y_edges: Targets for edges (batch_size, num_nodes, num_nodes)
00053             edge_cw: Class weights for edges loss
00054             # y_nodes: Targets for nodes (batch_size, num_nodes, num_nodes)
00055             # node_cw: Class weights for nodes loss
00056 
00057         Returns:
00058             y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
00059             # y_pred_nodes: Predictions for nodes (batch_size, num_nodes)
00060             loss: Value of loss function
00061         """
00062         # Node and edge embedding
00063         x = self.nodes_coord_embedding(x_nodes_coord)  # B x V x H
00064         e_vals = self.edges_values_embedding(x_edges_values.unsqueeze(3))  # B x V x V x H
00065         e_tags = self.edges_embedding(x_edges)  # B x V x V x H
00066         e = torch.cat((e_vals, e_tags), dim=3)
00067         # GCN layers
00068         for layer in range(self.num_layers):
00069             x, e = self.gcn_layers[layer](x, e)  # B x V x H, B x V x V x H
00070         # MLP classifier
00071         y_pred_edges = self.mlp_edges(e)  # B x V x V x voc_edges_out
00072         # y_pred_nodes = self.mlp_nodes(x)  # B x V x voc_nodes_out
00073 
00074         # Compute loss
00075         edge_cw = torch.Tensor(edge_cw).type(self.dtypeFloat)  # Convert to tensors
00076         loss = loss_edges(y_pred_edges, y_edges, edge_cw)
00077 
00078         return y_pred_edges, loss
```

10.43 HybridTSPSolver/README.md File Reference

10.44 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/↩ README.md File Reference

10.45 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/models/__init__.py File Reference

10.46 __init__.py

Go to the documentation of this file.

10.47 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/utils/↩ __init__.py File Reference

10.48 __init__.py

Go to the documentation of this file.

10.49 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/utils/beamsearch.py File Reference

Classes

- class utils.beamsearch.Beamsearch

Namespaces

- namespace utils
- namespace utils.beamsearch

10.50 beamsearch.py

Go to the documentation of this file.

```
00001 import numpy as np
00002 import torch
00003
00004
00005 class Beamsearch(object):
00006     """Class for managing internals of beamsearch procedure.
00007
00008     References:
00009         General: https://github.com/OpenNMT/OpenNMT-py/blob/master/onmt/translate/beam.py
00010         For TSP: https://github.com/alexnowakvila/QAP_pt/blob/master/src/tsp/beam_search.py
00011     """
00012
00013     def __init__(self, beam_size, batch_size, num_nodes,
00014                  dtypeFloat=torch.FloatTensor, dtypeLong=torch.LongTensor,
00015                  probs_type='raw', random_start=False):
00016         """
00017         Args:
00018             beam_size: Beam size
00019             batch_size: Batch size
00020             num_nodes: Number of nodes in TSP tours
00021             dtypeFloat: Float data type (for GPU/CPU compatibility)
00022             dtypeLong: Long data type (for GPU/CPU compatibility)
00023             probs_type: Type of probability values being handled by beamsearch (either
      'raw''logits''argmax'(TODO))
00024             random_start: Flag for using fixed (at node 0) vs. random starting points for beamsearch
00025         """
00026         # Beamsearch parameters
00027         self.batch_size = batch_size
00028         self.beam_size = beam_size
00029         self.num_nodes = num_nodes
00030         self.probs_type = probs_type
00031         # Set data types
00032         self.dtypeFloat = dtypeFloat
00033         self.dtypeLong = dtypeLong
00034         # Set beamsearch starting nodes
00035         self.start_nodes = torch.zeros(batch_size, beam_size).type(self.dtypeLong)
00036         if random_start == True:
00037             # Random starting nodes
00038             self.start_nodes = torch.randint(0, num_nodes, (batch_size,
      beam_size)).type(self.dtypeLong)
00039         # Mask for constructing valid hypothesis
00040         self.mask = torch.ones(batch_size, beam_size, num_nodes).type(self.dtypeFloat)
00041         self.update_mask(self.start_nodes)  # Mask the starting node of the beam search
00042         # Score for each translation on the beam
00043         self.scores = torch.zeros(batch_size, beam_size).type(self.dtypeFloat)
00044         self.all_scores = []
00045         # Backpointers at each time-step
00046         self.prev_Ks = []
00047         # Outputs at each time-step
00048         self.next_nodes = [self.start_nodes]
00049
00050     def get_current_state(self):
00051         """Get the output of the beam at the current timestep.
00052         """
00053         current_state = (self.next_nodes[-1].unsqueeze(2)
00054                          .expand(self.batch_size, self.beam_size, self.num_nodes))
00055         return current_state
```

```
00056
00057     def get_current_origin(self):
00058         """Get the backpointers for the current timestep.
00059         """
00060         return self.prev_Ks[-1]
00061
00062     def advance(self, trans_probs):
00063         """Advances the beam based on transition probabilities.
00064
00065         Args:
00066             trans_probs: Probabilities of advancing from the previous step (batch_size, beam_size,
      num_nodes)
00067         """
00068         # Compound the previous scores (summing logits == multiplying probabilities)
00069         if len(self.prev_Ks) > 0:
00070             if self.probs_type == 'raw':
00071                 beam_lk = trans_probs * self.scores.unsqueeze(2).expand_as(trans_probs)
00072             elif self.probs_type == 'logits':
00073                 beam_lk = trans_probs + self.scores.unsqueeze(2).expand_as(trans_probs)
00074         else:
00075             beam_lk = trans_probs
00076             # Only use the starting nodes from the beam
00077             if self.probs_type == 'raw':
00078                 beam_lk[:, 1:] = torch.zeros(beam_lk[:, 1:].size()).type(self.dtypeFloat)
00079             elif self.probs_type == 'logits':
00080                 beam_lk[:, 1:] = -1e20 * torch.ones(beam_lk[:, 1:].size()).type(self.dtypeFloat)
00081         # Multiply by mask
00082         beam_lk = beam_lk * self.mask
00083         beam_lk = beam_lk.view(self.batch_size, -1)  # (batch_size, beam_size * num_nodes)
00084         # Get top k scores and indexes (k = beam_size)
00085         bestScores, bestScoresId = beam_lk.topk(self.beam_size, 1, True, True)
00086         # Update scores
00087         self.scores = bestScores
00088         # Update backpointers
00089         prev_k = bestScoresId / self.num_nodes
00090         self.prev_Ks.append(prev_k)
00091         # Update outputs
00092         new_nodes = bestScoresId - prev_k * self.num_nodes
00093         self.next_nodes.append(new_nodes)
00094         # Re-index mask
00095         perm_mask = prev_k.unsqueeze(2).expand_as(self.mask)  # (batch_size, beam_size, num_nodes)
00096         self.mask = self.mask.gather(1, perm_mask)
00097         # Mask newly added nodes
00098         self.update_mask(new_nodes)
00099
00100     def update_mask(self, new_nodes):
00101         """Sets new_nodes to zero in mask.
00102         """
00103         arr = (torch.arange(0, self.num_nodes).unsqueeze(0).unsqueeze(1)
00104                .expand_as(self.mask).type(self.dtypeLong))
00105         new_nodes = new_nodes.unsqueeze(2).expand_as(self.mask)
00106         update_mask = 1 - torch.eq(arr, new_nodes).type(self.dtypeFloat)
00107         self.mask = self.mask * update_mask
00108         if self.probs_type == 'logits':
00109             # Convert 0s in mask to inf
00110             self.mask[self.mask == 0] = 1e20
00111
00112     def sort_best(self):
00113         """Sort the beam.
00114         """
00115         return torch.sort(self.scores, 0, True)
00116
00117     def get_best(self):
00118         """Get the score and index of the best hypothesis in the beam.
00119         """
00120         scores, ids = self.sort_best()
00121         return scores[1], ids[1]
00122
00123     def get_hypothesis(self, k):
00124         """Walk back to construct the full hypothesis.
00125
00126         Args:
00127             k: Position in the beam to construct (usually 0s for most probable hypothesis)
00128         """
00129         assert self.num_nodes == len(self.prev_Ks) + 1
00130
00131         hyp = -1 * torch.ones(self.batch_size, self.num_nodes).type(self.dtypeLong)
00132         for j in range(len(self.prev_Ks) - 1, -2, -1):
00133             hyp[:, j + 1] = self.next_nodes[j + 1].gather(1, k).view(1, self.batch_size)
00134             k = self.prev_Ks[j].gather(1, k)
00135         return hyp
```

10.51 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/utils/google_tsp_reader.py File Reference

Classes

- class utils.google_tsp_reader.DotDict
- class utils.google_tsp_reader.GoogleTSPReader

Namespaces

- namespace utils
- namespace utils.google_tsp_reader

10.52 google_tsp_reader.py

Go to the documentation of this file.

```
00001 import time
00002 import numpy as np
00003 from scipy.spatial.distance import pdist, squareform
00004 from sklearn.utils import shuffle
00005
00006
00007 class DotDict(dict):
00008     """Wrapper around in-built dict class to access members through the dot operation.
00009     """
00010
00011     def __init__(self, **kwds):
00012         self.update(kwds)
00013         self.__dict__ = self
00014
00015
00016 class GoogleTSPReader(object):
00017     """Iterator that reads TSP dataset files and yields mini-batches.
00018
00019     Format expected as in Vinyals et al., 2015: https://arxiv.org/abs/1506.03134, http://goo.gl/NDcOIG
00020     """
00021
00022     def __init__(self, num_nodes, num_neighbors, batch_size, filepath):
00023         """
00024         Args:
00025             num_nodes: Number of nodes in TSP tours
00026             num_neighbors: Number of neighbors to consider for each node in graph
00027             batch_size: Batch size
00028             filepath: Path to dataset file (.txt file)
00029         """
00030         self.num_nodes = num_nodes
00031         self.num_neighbors = num_neighbors
00032         self.batch_size = batch_size
00033         self.filepath = filepath
00034         self.filedata = shuffle(open(filepath, "r").readlines())  # Always shuffle upon reading data
00035         self.max_iter = (len(self.filedata) // batch_size)
00036
00037     def __iter__(self):
00038         for batch in range(self.max_iter):
00039             start_idx = batch * self.batch_size
00040             end_idx = (batch + 1) * self.batch_size
00041             yield self.process_batch(self.filedata[start_idx:end_idx])
00042
00043     def process_batch(self, lines):
00044         """Helper function to convert raw lines into a mini-batch as a DotDict.
00045         """
00046         batch_edges = []
00047         batch_edges_values = []
00048         batch_edges_target = []  # Binary classification targets (0/1)
00049         batch_nodes = []
00050         batch_nodes_target = []  # Multi-class classification targets (`num_nodes` classes)
00051         batch_nodes_coord = []
00052         batch_tour_nodes = []
00053         batch_tour_len = []
00054
00055         for line_num, line in enumerate(lines):
```

```
00056             line = line.split(" ")  # Split into list
00057
00058             # Compute signal on nodes
00059             nodes = np.ones(self.num_nodes)  # All 1s for TSP...
00060
00061             # Convert node coordinates to required format
00062             nodes_coord = []
00063             for idx in range(0, 2 * self.num_nodes, 2):
00064                 nodes_coord.append([float(line[idx]), float(line[idx + 1])])
00065
00066             # Compute distance matrix
00067             W_val = squareform(pdist(nodes_coord, metric='euclidean'))
00068
00069             # Compute adjacency matrix
00070             if self.num_neighbors == -1:
00071                 W = np.ones((self.num_nodes, self.num_nodes))  # Graph is fully connected
00072             else:
00073                 W = np.zeros((self.num_nodes, self.num_nodes))
00074                 # Determine k-nearest neighbors for each node
00075                 knns = np.argpartition(W_val, kth=self.num_neighbors, axis=-1)[:,
      self.num_neighbors::-1]
00076                 # Make connections
00077                 for idx in range(self.num_nodes):
00078                     W[idx][knns[idx]] = 1
00079             np.fill_diagonal(W, 2)  # Special token for self-connections
00080
00081             # Convert tour nodes to required format
00082             # Don't add final connection for tour/cycle
00083             tour_nodes = [int(node) - 1 for node in line[line.index('output') + 1:-1]][:-1]
00084
00085             # Compute node and edge representation of tour + tour_len
00086             tour_len = 0
00087             nodes_target = np.zeros(self.num_nodes)
00088             edges_target = np.zeros((self.num_nodes, self.num_nodes))
00089             for idx in range(len(tour_nodes) - 1):
00090                 i = tour_nodes[idx]
00091                 j = tour_nodes[idx + 1]
00092                 nodes_target[i] = idx  # node targets: ordering of nodes in tour
00093                 edges_target[i][j] = 1
00094                 edges_target[j][i] = 1
00095                 tour_len += W_val[i][j]
00096
00097             # Add final connection of tour in edge target
00098             nodes_target[j] = len(tour_nodes) - 1
00099             edges_target[j][tour_nodes[0]] = 1
00100             edges_target[tour_nodes[0]][j] = 1
00101             tour_len += W_val[j][tour_nodes[0]]
00102
00103             # Concatenate the data
00104             batch_edges.append(W)
00105             batch_edges_values.append(W_val)
00106             batch_edges_target.append(edges_target)
00107             batch_nodes.append(nodes)
00108             batch_nodes_target.append(nodes_target)
00109             batch_nodes_coord.append(nodes_coord)
00110             batch_tour_nodes.append(tour_nodes)
00111             batch_tour_len.append(tour_len)
00112
00113         # From list to tensors as a DotDict
00114         batch = DotDict()
00115         batch.edges = np.stack(batch_edges, axis=0)
00116         batch.edges_values = np.stack(batch_edges_values, axis=0)
00117         batch.edges_target = np.stack(batch_edges_target, axis=0)
00118         batch.nodes = np.stack(batch_nodes, axis=0)
00119         batch.nodes_target = np.stack(batch_nodes_target, axis=0)
00120         batch.nodes_coord = np.stack(batch_nodes_coord, axis=0)
00121         batch.tour_nodes = np.stack(batch_tour_nodes, axis=0)
00122         batch.tour_len = np.stack(batch_tour_len, axis=0)
00123         return batch
```

10.53 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/utils/graph_utils.py File Reference

Namespaces

- namespace utils
- namespace utils.graph_utils

Functions

- def utils.graph_utils.tour_nodes_to_W (nodes)
- def utils.graph_utils.tour_nodes_to_tour_len (nodes, W_values)
- def utils.graph_utils.W_to_tour_len (W, W_values)
- def utils.graph_utils.is_valid_tour (nodes, num_nodes)
- def utils.graph_utils.mean_tour_len_edges (x_edges_values, y_pred_edges)
- def utils.graph_utils.mean_tour_len_nodes (x_edges_values, bs_nodes)
- def utils.graph_utils.get_max_k (dataset, max_iter=1000)

10.54 graph_utils.py

Go to the documentation of this file.

```
00001 import torch
00002 import torch.nn.functional as F
00003
00004 import numpy as np
00005
00006
00007 def tour_nodes_to_W(nodes):
00008     """Helper function to convert ordered list of tour nodes to edge adjacency matrix.
00009     """
00010     W = np.zeros((len(nodes), len(nodes)))
00011     for idx in range(len(nodes) - 1):
00012         i = int(nodes[idx])
00013         j = int(nodes[idx + 1])
00014         W[i][j] = 1
00015         W[j][i] = 1
00016     # Add final connection of tour in edge target
00017     W[j][int(nodes[0])] = 1
00018     W[int(nodes[0])][j] = 1
00019     return W
00020
00021
00022 def tour_nodes_to_tour_len(nodes, W_values):
00023     """Helper function to calculate tour length from ordered list of tour nodes.
00024     """
00025     tour_len = 0
00026     for idx in range(len(nodes) - 1):
00027         i = nodes[idx]
00028         j = nodes[idx + 1]
00029         tour_len += W_values[i][j]
00030     # Add final connection of tour in edge target
00031     tour_len += W_values[j][nodes[0]]
00032     return tour_len
00033
00034
00035 def W_to_tour_len(W, W_values):
00036     """Helper function to calculate tour length from edge adjacency matrix.
00037     """
00038     tour_len = 0
00039     for i in range(W.shape[0]):
00040         for j in range(W.shape[1]):
00041             if W[i][j] == 1:
00042                 tour_len += W_values[i][j]
00043     tour_len /= 2  # Divide by 2 because adjacency matrices are symmetric
00044     return tour_len
00045
00046
00047 def is_valid_tour(nodes, num_nodes):
00048     """Sanity check: tour visits all nodes given.
00049     """
00050     return sorted(nodes) == [i for i in range(num_nodes)]
00051
00052
00053 def mean_tour_len_edges(x_edges_values, y_pred_edges):
00054     """
00055     Computes mean tour length for given batch prediction as edge adjacency matrices (for PyTorch
      tensors).
00056
00057     Args:
00058         x_edges_values: Edge values (distance) matrix (batch_size, num_nodes, num_nodes)
00059         y_pred_edges: Edge predictions (batch_size, num_nodes, num_nodes, voc_edges)
00060
00061     Returns:
00062         mean_tour_len: Mean tour length over batch
00063     """
```

```
00064     y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
00065     y = y.argmax(dim=3)  # B x V x V
00066     # Divide by 2 because edges_values is symmetric
00067     tour_lens = (y.float() * x_edges_values.float()).sum(dim=1).sum(dim=1) / 2
00068     mean_tour_len = tour_lens.sum().to(dtype=torch.float).item() / tour_lens.numel()
00069     return mean_tour_len
00070
00071
00072 def mean_tour_len_nodes(x_edges_values, bs_nodes):
00073     """
00074     Computes mean tour length for given batch prediction as node ordering after beamsearch (for
      Pytorch tensors).
00075
00076     Args:
00077         x_edges_values: Edge values (distance) matrix (batch_size, num_nodes, num_nodes)
00078         bs_nodes: Node orderings (batch_size, num_nodes)
00079
00080     Returns:
00081         mean_tour_len: Mean tour length over batch
00082     """
00083     y = bs_nodes.cpu().numpy()
00084     W_val = x_edges_values.cpu().numpy()
00085     running_tour_len = 0
00086     for batch_idx in range(y.shape[0]):
00087         for y_idx in range(y[batch_idx].shape[0] - 1):
00088             i = y[batch_idx][y_idx]
00089             j = y[batch_idx][y_idx + 1]
00090             running_tour_len += W_val[batch_idx][i][j]
00091         running_tour_len += W_val[batch_idx][j][0]  # Add final connection to tour/cycle
00092     return running_tour_len / y.shape[0]
00093
00094
00095 def get_max_k(dataset, max_iter=1000):
00096     """
00097     Given a TSP dataset, compute the maximum value of k for which the k'th nearest neighbor
00098     of a node is connected to it in the groundtruth TSP tour.
00099
00100     For each node in all instances, compute the value of k for the next node in the tour,
00101     and take the max of all ks.
00102     """
00103     ks = []
00104     for _ in range(max_iter):
00105         batch = next(iter(dataset))
00106         for idx in range(batch.edges.shape[0]):
00107             for row in range(dataset.num_nodes):
00108                 # Compute indices of current node's neighbors in the TSP solution
00109                 connections = np.where(batch.edges_target[idx][row]==1)[0]
00110                 # Compute sorted list of indices of nearest neighbors (ascending order)
00111                 sorted_neighbors = np.argsort(batch.edges_values[idx][row], axis=-1)
00112                 for conn_idx in connections:
00113                     ks.append(np.where(sorted_neighbors==conn_idx)[0][0])
00114     # print("Ks array counts: ", np.unique(ks, return_counts=True))
00115     # print(f"Mean: {np.mean(ks)}, StdDev: {np.std(ks)}")
00116     return int(np.max(ks))
```

10.55 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/utils/model_utils.py File Reference

Namespaces

- namespace utils
- namespace utils.model_utils

Functions

- def utils.model_utils.loss_nodes (y_pred_nodes, y_nodes, node_cw)
- def utils.model_utils.loss_edges (y_pred_edges, y_edges, edge_cw)
- def utils.model_utils.beamsearch_tour_nodes (y_pred_edges, beam_size, batch_size, num_nodes, dtype↩ Float, dtypeLong, probs_type='raw', random_start=False)
- def utils.model_utils.beamsearch_tour_nodes_shortest (y_pred_edges, x_edges_values, beam_size, batch_size, num_nodes, dtypeFloat, dtypeLong, probs_type='raw', random_start=False)
- def utils.model_utils.update_learning_rate (optimizer, lr)
- def utils.model_utils.edge_error (y_pred, y_target, x_edges)
- def utils.model_utils._edge_error (y, y_target, mask)

10.56 model_utils.py

Go to the documentation of this file.

```
00001 import torch
00002 import torch.nn.functional as F
00003 import torch.nn as nn
00004
00005 from utils.beamsearch import *
00006 from utils.graph_utils import *
00007
00008
00009 def loss_nodes(y_pred_nodes, y_nodes, node_cw):
00010     """
00011     Loss function for node predictions.
00012
00013     Args:
00014         y_pred_nodes: Predictions for nodes (batch_size, num_nodes)
00015         y_nodes: Targets for nodes (batch_size, num_nodes)
00016         node_cw: Class weights for nodes loss
00017
00018     Returns:
00019         loss_nodes: Value of loss function
00020
00021     """
00022     # Node loss
00023     y = F.log_softmax(y_pred_nodes, dim=2)  # B x V x voc_nodes_out
00024     y = y.permute(0, 2, 1)  # B x voc_nodes x V
00025     loss_nodes = nn.NLLLoss(node_cw)(y, y_nodes)
00026     return loss_nodes
00027
00028
00029 def loss_edges(y_pred_edges, y_edges, edge_cw):
00030     """
00031     Loss function for edge predictions.
00032
00033     Args:
00034         y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
00035         y_edges: Targets for edges (batch_size, num_nodes, num_nodes)
00036         edge_cw: Class weights for edges loss
00037
00038     Returns:
00039         loss_edges: Value of loss function
00040
00041     """
00042     # Edge loss
00043     y = F.log_softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
00044     y = y.permute(0, 3, 1, 2)  # B x voc_edges x V x V
00045     loss_edges = nn.NLLLoss(edge_cw)(y, y_edges)
00046     return loss_edges
00047
00048
00049 def beamsearch_tour_nodes(y_pred_edges, beam_size, batch_size, num_nodes, dtypeFloat, dtypeLong,
      probs_type='raw', random_start=False):
00050     """
00051     Performs beamsearch procedure on edge prediction matrices and returns possible TSP tours.
00052
00053     Args:
00054         y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
00055         beam_size: Beam size
00056         batch_size: Batch size
00057         num_nodes: Number of nodes in TSP tours
00058         dtypeFloat: Float data type (for GPU/CPU compatibility)
00059         dtypeLong: Long data type (for GPU/CPU compatibility)
00060         random_start: Flag for using fixed (at node 0) vs. random starting points for beamsearch
00061
00062     Returns: TSP tours in terms of node ordering (batch_size, num_nodes)
00063
00064     """
00065     if probs_type == 'raw':
00066         # Compute softmax over edge prediction matrix
00067         y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
00068         # Consider the second dimension only
00069         y = y[:, :, :, 1]  # B x V x V
00070     elif probs_type == 'logits':
00071         # Compute logits over edge prediction matrix
00072         y = F.log_softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
00073         # Consider the second dimension only
00074         y = y[:, :, :, 1]  # B x V x V
00075         y[y == 0] = -1e-20  # Set 0s (i.e. log(1)s) to very small negative number
00076     # Perform beamsearch
00077     beamsearch = Beamsearch(beam_size, batch_size, num_nodes, dtypeFloat, dtypeLong, probs_type,
      random_start)
00078     trans_probs = y.gather(1, beamsearch.get_current_state())
00079     for step in range(num_nodes - 1):
00080         beamsearch.advance(trans_probs)
```

```
00081         trans_probs = y.gather(1, beamsearch.get_current_state())
00082     # Find TSP tour with highest probability among beam_size candidates
00083     ends = torch.zeros(batch_size, 1).type(dtypeLong)
00084     return beamsearch.get_hypothesis(ends)
00085
00086
00087 def beamsearch_tour_nodes_shortest(y_pred_edges, x_edges_values, beam_size, batch_size, num_nodes,
00088                                    dtypeFloat, dtypeLong, probs_type='raw', random_start=False):
00089     """
00090     Performs beamsearch procedure on edge prediction matrices and returns possible TSP tours.
00091
00092     Final predicted tour is the one with the shortest tour length.
00093     (Standard beamsearch returns the one with the highest probability and does not take length into
      account.)
00094
00095     Args:
00096         y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
00097         x_edges_values: Input edge distance matrix (batch_size, num_nodes, num_nodes)
00098         beam_size: Beam size
00099         batch_size: Batch size
00100         num_nodes: Number of nodes in TSP tours
00101         dtypeFloat: Float data type (for GPU/CPU compatibility)
00102         dtypeLong: Long data type (for GPU/CPU compatibility)
00103         probs_type: Type of probability values being handled by beamsearch (either
      'raw'/'logits'/'argmax'(TODO))
00104         random_start: Flag for using fixed (at node 0) vs. random starting points for beamsearch
00105
00106     Returns:
00107         shortest_tours: TSP tours in terms of node ordering (batch_size, num_nodes)
00108
00109     """
00110     if probs_type == 'raw':
00111         # Compute softmax over edge prediction matrix
00112         y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
00113         # Consider the second dimension only
00114         y = y[:, :, :, 1]  # B x V x V
00115     elif probs_type == 'logits':
00116         # Compute logits over edge prediction matrix
00117         y = F.log_softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
00118         # Consider the second dimension only
00119         y = y[:, :, :, 1]  # B x V x V
00120         y[y == 0] = -1e-20  # Set 0s (i.e. log(1)s) to very small negative number
00121     # Perform beamsearch
00122     beamsearch = Beamsearch(beam_size, batch_size, num_nodes, dtypeFloat, dtypeLong, probs_type,
      random_start)
00123     trans_probs = y.gather(1, beamsearch.get_current_state())
00124     for step in range(num_nodes - 1):
00125         beamsearch.advance(trans_probs)
00126         trans_probs = y.gather(1, beamsearch.get_current_state())
00127     # Initially assign shortest_tours as most probable tours i.e. standard beamsearch
00128     ends = torch.zeros(batch_size, 1).type(dtypeLong)
00129     shortest_tours = beamsearch.get_hypothesis(ends)
00130     # Compute current tour lengths
00131     shortest_lens = [1e6] * len(shortest_tours)
00132     for idx in range(len(shortest_tours)):
00133         shortest_lens[idx] = tour_nodes_to_tour_len(shortest_tours[idx].cpu().numpy(),
00134                                                     x_edges_values[idx].cpu().numpy())
00135     # Iterate over all positions in beam (except position 0 --> highest probability)
00136     for pos in range(1, beam_size):
00137         ends = pos * torch.ones(batch_size, 1).type(dtypeLong)  # New positions
00138         hyp_tours = beamsearch.get_hypothesis(ends)
00139         for idx in range(len(hyp_tours)):
00140             hyp_nodes = hyp_tours[idx].cpu().numpy()
00141             hyp_len = tour_nodes_to_tour_len(hyp_nodes, x_edges_values[idx].cpu().numpy())
00142             # Replace tour in shortest_tours if new length is shorter than current best
00143             if hyp_len < shortest_lens[idx] and is_valid_tour(hyp_nodes, num_nodes):
00144                 shortest_tours[idx] = hyp_tours[idx]
00145                 shortest_lens[idx] = hyp_len
00146     return shortest_tours
00147
00148
00149 def update_learning_rate(optimizer, lr):
00150     """
00151     Updates learning rate for given optimizer.
00152
00153     Args:
00154         optimizer: Optimizer object
00155         lr: New learning rate
00156
00157     Returns:
00158         optimizer: Updated optimizer object
00159         s
00160     """
00161     for param_group in optimizer.param_groups:
00162         param_group['lr'] = lr
00163     return optimizer
00164
```

```
00165
00166 def edge_error(y_pred, y_target, x_edges):
00167     """
00168     Computes edge error metrics for given batch prediction and targets.
00169
00170     Args:
00171         y_pred: Edge predictions (batch_size, num_nodes, num_nodes, voc_edges)
00172         y_target: Edge targets (batch_size, num_nodes, num_nodes)
00173         x_edges: Adjacency matrix (batch_size, num_nodes, num_nodes)
00174
00175     Returns:
00176         err_edges, err_tour, err_tsp, edge_err_idx, err_idx_tour, err_idx_tsp
00177
00178     """
00179     y = F.softmax(y_pred, dim=3)  # B x V x V x voc_edges
00180     y = y.argmax(dim=3)  # B x V x V
00181
00182     # Edge error: Mask out edges which are not connected
00183     mask_no_edges = x_edges.long()
00184     err_edges, _ = _edge_error(y, y_target, mask_no_edges)
00185
00186     # TSP tour edges error: Mask out edges which are not on true TSP tours
00187     mask_no_tour = y_target
00188     err_tour, err_idx_tour = _edge_error(y, y_target, mask_no_tour)
00189
00190     # TSP tour edges + positively predicted edges error:
00191     # Mask out edges which are not on true TSP tours or are not predicted positively by model
00192     mask_no_tsp = ((y_target + y) > 0).long()
00193     err_tsp, err_idx_tsp = _edge_error(y, y_target, mask_no_tsp)
00194
00195     return 100 * err_edges, 100 * err_tour, 100 * err_tsp, err_idx_tour, err_idx_tsp
00196
00197
00198 def _edge_error(y, y_target, mask):
00199     """
00200     Helper method to compute edge errors.
00201
00202     Args:
00203         y: Edge predictions (batch_size, num_nodes, num_nodes)
00204         y_target: Edge targets (batch_size, num_nodes, num_nodes)
00205         mask: Edges which are not counted in error computation (batch_size, num_nodes, num_nodes)
00206
00207     Returns:
00208         err: Mean error over batch
00209         err_idx: One-hot array of shape (batch_size)- 1s correspond to indices which are not perfectly
      predicted
00210
00211     """
00212     # Compute equalities between pred and target
00213     acc = (y == y_target).long()
00214     # Multipy by mask => set equality to 0 on disconnected edges
00215     acc = (acc * mask)
00216     #  Get accuracy of each y in the batch (sum of 1s in acc_edges divided by sum of 1s in edges mask)
00217     acc = acc.sum(dim=1).sum(dim=1).to(dtype=torch.float) /
     mask.sum(dim=1).sum(dim=1).to(dtype=torch.float)
00218     # Compute indices which are not perfect
00219     err_idx = (acc < 1.0)
00220     # Take mean over batch
00221     acc = acc.sum().to(dtype=torch.float).item() / acc.numel()
00222     # Compute error
00223     err = 1.0 - acc
00224     return err, err_idx
```

10.57 HybridTSPSolver/src/HybridSolver/main/graph-convnet-tsp/utils/plot_utils.py File Reference

Namespaces

- namespace utils
- namespace utils.plot_utils

Functions

- def utils.plot_utils.plot_tsp (p, x_coord, W, W_val, W_target, title="default")

- def utils.plot_utils.plot_tsp_heatmap (p, x_coord, W_val, W_pred, title="default")
- def utils.plot_utils.plot_predictions (x_nodes_coord, x_edges, x_edges_values, y_edges, y_pred_edges, num_plots=3)
- def utils.plot_utils.plot_predictions_beamsearch (x_nodes_coord, x_edges, x_edges_values, y_edges, y_↩ pred_edges, bs_nodes, num_plots=3)

10.58 plot_utils.py

Go to the documentation of this file.

```
00001 import torch
00002 import torch.nn.functional as F
00003
00004 import matplotlib
00005 import matplotlib.pyplot as plt
00006 import networkx as nx
00007
00008 from utils.graph_utils import *
00009
00010
00011 def plot_tsp(p, x_coord, W, W_val, W_target, title="default"):
00012     """
00013     Helper function to plot TSP tours.
00014
00015     Args:
00016         p: Matplotlib figure/subplot
00017         x_coord: Coordinates of nodes
00018         W: Edge adjacency matrix
00019         W_val: Edge values (distance) matrix
00020         W_target: One-hot matrix with 1s on groundtruth/predicted edges
00021         title: Title of figure/subplot
00022
00023     Returns:
00024         p: Updated figure/subplot
00025
00026     """
00027
00028     def _edges_to_node_pairs(W):
00029         """Helper function to convert edge matrix into pairs of adjacent nodes.
00030         """
00031         pairs = []
00032         for r in range(len(W)):
00033             for c in range(len(W)):
00034                 if W[r][c] == 1:
00035                     pairs.append((r, c))
00036         return pairs
00037
00038     G = nx.from_numpy_matrix(W_val)
00039     pos = dict(zip(range(len(x_coord)), x_coord.tolist()))
00040     adj_pairs = _edges_to_node_pairs(W)
00041     target_pairs = _edges_to_node_pairs(W_target)
00042     colors = ['g'] + ['b'] * (len(x_coord) - 1)  # Green for 0th node, blue for others
00043     nx.draw_networkx_nodes(G, pos, node_color=colors, node_size=50)
00044     nx.draw_networkx_edges(G, pos, edgelist=adj_pairs, alpha=0.3, width=0.5)
00045     nx.draw_networkx_edges(G, pos, edgelist=target_pairs, alpha=1, width=1, edge_color='r')
00046     p.set_title(title)
00047     return p
00048
00049
00050 def plot_tsp_heatmap(p, x_coord, W_val, W_pred, title="default"):
00051     """
00052     Helper function to plot predicted TSP tours with edge strength denoting confidence of prediction.
00053
00054     Args:
00055         p: Matplotlib figure/subplot
00056         x_coord: Coordinates of nodes
00057         W_val: Edge values (distance) matrix
00058         W_pred: Edge predictions matrix
00059         title: Title of figure/subplot
00060
00061     Returns:
00062         p: Updated figure/subplot
00063
00064     """
00065
00066     def _edges_to_node_pairs(W):
00067         """Helper function to convert edge matrix into pairs of adjacent nodes.
00068         """
00069         pairs = []
00070         edge_preds = []
```

```
00071         for r in range(len(W)):
00072             for c in range(len(W)):
00073                 if W[r][c] > 0.25:
00074                     pairs.append((r, c))
00075                     edge_preds.append(W[r][c])
00076         return pairs, edge_preds
00077 
00078     G = nx.from_numpy_matrix(W_val)
00079     pos = dict(zip(range(len(x_coord)), x_coord.tolist()))
00080     node_pairs, edge_color = _edges_to_node_pairs(W_pred)
00081     node_color = ['g'] + ['b'] * (len(x_coord) - 1)  # Green for 0th node, blue for others
00082     nx.draw_networkx_nodes(G, pos, node_color=node_color, node_size=50)
00083     nx.draw_networkx_edges(G, pos, edgelist=node_pairs, edge_color=edge_color, edge_cmap=plt.cm.Reds,
      width=0.75)
00084     p.set_title(title)
00085     return p
00086 
00087 
00088 def plot_predictions(x_nodes_coord, x_edges, x_edges_values, y_edges, y_pred_edges, num_plots=3):
00089     """
00090     Plots groundtruth TSP tour vs. predicted tours (without beamsearch).
00091 
00092     Args:
00093         x_nodes_coord: Input node coordinates (batch_size, num_nodes, node_dim)
00094         x_edges: Input edge adjacency matrix (batch_size, num_nodes, num_nodes)
00095         x_edges_values: Input edge distance matrix (batch_size, num_nodes, num_nodes)
00096         y_edges: Groundtruth labels for edges (batch_size, num_nodes, num_nodes)
00097         y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
00098         num_plots: Number of figures to plot
00099 
00100     """
00101     y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
00102     y_bins = y.argmax(dim=3)  # Binary predictions: B x V x V
00103     y_probs = y[:,:,:,1]  # Prediction probabilities: B x V x V
00104     for f_idx, idx in enumerate(np.random.choice(len(y), num_plots, replace=False)):
00105         f = plt.figure(f_idx, figsize=(10, 5))
00106         x_coord = x_nodes_coord[idx].cpu().numpy()
00107         W = x_edges[idx].cpu().numpy()
00108         W_val = x_edges_values[idx].cpu().numpy()
00109         W_target = y_edges[idx].cpu().numpy()
00110         W_sol_bins = y_bins[idx].cpu().numpy()
00111         W_sol_probs = y_probs[idx].cpu().numpy()
00112         plt1 = f.add_subplot(121)
00113         plot_tsp(plt1, x_coord, W, W_val, W_target, 'Groundtruth:
      {:.3f}'.format(W_to_tour_len(W_target, W_val)))
00114         plt2 = f.add_subplot(122)
00115         plot_tsp_heatmap(plt2, x_coord, W_val, W_sol_probs, 'Prediction Heatmap')
00116         plt.show()
00117 
00118 
00119 def plot_predictions_beamsearch(x_nodes_coord, x_edges, x_edges_values, y_edges, y_pred_edges,
      bs_nodes, num_plots=3):
00120     """
00121     Plots groundtruth TSP tour vs. predicted tours (with beamsearch).
00122 
00123     Args:
00124         x_nodes_coord: Input node coordinates (batch_size, num_nodes, node_dim)
00125         x_edges: Input edge adjacency matrix (batch_size, num_nodes, num_nodes)
00126         x_edges_values: Input edge distance matrix (batch_size, num_nodes, num_nodes)
00127         y_edges: Groundtruth labels for edges (batch_size, num_nodes, num_nodes)
00128         y_pred_edges: Predictions for edges (batch_size, num_nodes, num_nodes)
00129         bs_nodes: Predicted node ordering in TSP tours after beamsearch (batch_size, num_nodes)
00130         num_plots: Number of figures to plot
00131 
00132     """
00133     y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
00134     y_bins = y.argmax(dim=3)  # Binary predictions: B x V x V
00135     y_probs = y[:,:,:,1]  # Prediction probabilities: B x V x V
00136     #print(y_probs)
00137     for f_idx, idx in enumerate(np.random.choice(len(y), num_plots, replace=False)):
00138         f = plt.figure(f_idx, figsize=(15, 5))
00139         x_coord = x_nodes_coord[idx].cpu().numpy()
00140         W = x_edges[idx].cpu().numpy()
00141         W_val = x_edges_values[idx].cpu().numpy()
00142         W_target = y_edges[idx].cpu().numpy()
00143         W_sol_bins = y_bins[idx].cpu().numpy()
00144         W_sol_probs = y_probs[idx].cpu().numpy()
00145         W_bs = tour_nodes_to_W(bs_nodes[idx].cpu().numpy())
00146         plt1 = f.add_subplot(131)
00147         plot_tsp(plt1, x_coord, W, W_val, W_target, 'Groundtruth:
      {:.3f}'.format(W_to_tour_len(W_target, W_val)))
00148         plt2 = f.add_subplot(132)
00149         plot_tsp_heatmap(plt2, x_coord, W_val, W_sol_probs, 'Prediction Heatmap')
00150         plt3 = f.add_subplot(133)
00151         plot_tsp(plt3, x_coord, W, W_val, W_bs, 'Beamsearch: {:.3f}'.format(W_to_tour_len(W_bs,
      W_val)))
00152         plt.show()
```

10.59 HybridTSPSolver/src/HybridSolver/main/HybridSolver.py File Reference

Namespaces

- namespace HybridSolver

Functions

- def HybridSolver.build_c_program (build_directory, num_nodes, hyb_mode)
- def HybridSolver.hybrid_solver (num_instances, num_nodes, hyb_mode)

Variables

- sys HybridSolver.num_instances = sys.argv[1]
- int HybridSolver.num_nodes = int(sys.argv[2])
- tuple HybridSolver.hyb_mode = (sys.argv[3] == "y" or sys.argv[3] == "Y" or sys.argv[3] == "yes" or sys.argv[3] == "Yes")

10.60 HybridSolver.py

Go to the documentation of this file.

```
00001 """
00002     @file: HybridSolver.py
00003     @author Lorenzo Sciandra
00004     @brief First it builds the program in C, specifying the number of nodes to use and whether it is
      in hybrid mode or not.
00005     Then it runs the graph conv net on the instance, and finally it runs the Branch and Bound.
00006     It can be run on a single instance or a range of instances.
00007     The input matrix is generated by the neural network and stored in the data folder. The output is
      stored in the results folder.
00008     @version 0.1.0
00009     @date 2023-04-18
00010     @copyright Copyright (c) 2023, license MIT
00011
00012     Repo: https://github.com/LorenzoSciandra/HybridTSPSolver
00013 """
00014
00015
00016 import subprocess
00017 import sys
00018 import os
00019 import time
00020
00021
00022 def build_c_program(build_directory, num_nodes, hyb_mode):
00023     """
00024     Args:
00025         build_directory: The directory where the CMakeLists.txt file is located and where the
      executable will be built.
00026         num_nodes: The number of nodes to use in the C program.
00027         hyb_mode: 1 if the program is in hybrid mode, 0 otherwise.
00028     """
00029     source_directory = "../"
00030     cmake_command = [
00031         "cmake",
00032         "-S" + source_directory,
00033         "-B" + build_directory,
00034         "-DCMAKE_BUILD_TYPE=Release",
00035         "-DMAX_VERTEX_NUM=" + str(num_nodes),
00036         "-DHYBRID=" + str(hyb_mode)
00037     ]
00038     print(cmake_command)
00039     make_command = [
00040         "make",
00041         "-C" + build_directory,
```

```
00042         "-j"
00043     ]
00044     try:
00045         subprocess.check_call(cmake_command)
00046         subprocess.check_call(make_command)
00047     except subprocess.CalledProcessError as e:
00048         print("Build failed:")
00049         print(e.output)
00050         raise Exception("Build failed")
00051
00052
00053 def hybrid_solver(num_instances, num_nodes, hyb_mode):
00054     """
00055     Args:
00056         num_instances: The range of instances to run on the Solver.
00057         num_nodes: The number of nodes to use in the C program.
00058         hyb_mode: True if the program is in hybrid mode, False otherwise.
00059     """
00060     build_directory = "../cmake-build/CMakeFiles/BranchAndBound1Tree.dir"
00061     hybrid = 1 if hyb_mode else 0
00062     build_c_program(build_directory, num_nodes, hybrid)
00063
00064     if "-" in num_instances:
00065         instances = num_instances.split("-")
00066         start_instance = 1 if int(instances[0]) == 0 else int(instances[0])
00067         end_instance = int(instances[1])
00068     else:
00069         start_instance = 1
00070         end_instance = int(num_instances)
00071
00072     print("Starting instance: " + str(start_instance))
00073     print("Ending instance: " + str(end_instance))
00074
00075     for i in range(start_instance, end_instance + 1):
00076         start_time = time.time()
00077         input_file = "../data/AdjacencyMatrix/tsp_" + str(num_nodes) + "_nodes/tsp_test_" + str(i) +
      ".csv"
00078         absolute_input_path = os.path.abspath(input_file)
00079         result_mode = "hybrid" if hyb_mode else "classic"
00080         output_file = "../results/AdjacencyMatrix/tsp_" + str(num_nodes) + "_nodes_" + result_mode \
00081                       + "/tsp_result_" + str(i) + ".txt"
00082         if hyb_mode:
00083             absolute_python_path = os.path.abspath("./graph-convnet-tsp/main.py")
00084             result = subprocess.run(['python3', absolute_python_path, absolute_input_path,
      str(num_nodes), str(i)], cwd="./graph-convnet-tsp",
00085                                     check=True)
00086             if result.returncode == 0:
00087                 print('Neural Network completed successfully on instance ' + str(i) + ' / ' +
      str(end_instance))
00088             else:
00089                 print('Neural Network failed on instance ' + str(i) + ' / ' + str(end_instance))
00090             os.rename(absolute_input_path.replace(".csv", "_temp.csv"), absolute_input_path)
00091
00092         absolute_output_path = os.path.abspath(output_file)
00093         cmd = [build_directory + "/BranchAndBound1Tree", absolute_input_path, absolute_output_path]
00094         result = subprocess.run(cmd)
00095         if result.returncode == 0:
00096             print('Branch-and-Bound completed successfully on instance ' + str(i) + ' / ' +
      str(end_instance))
00097         else:
00098             print('Branch-and-Bound failed on instance ' + str(i) + ' / ' + str(end_instance))
00099         end_time = time.time()
00100         # append to the output file the time taken to solve the instance
00101         with open(output_file, "a") as f:
00102             f.write("Time taken: " + str(end_time - start_time) + "s\n")
00103
00104
00105 if __name__ == "__main__":
00106     """
00107     Args:
00108         sys.argv[1]: The range of instances to run on the Solver.
00109         sys.argv[2]: The number of nodes to use in the C program.
00110         sys.argv[3]: "y" if the program is in hybrid mode, "n" otherwise.
00111     """
00112
00113     if len(sys.argv) != 4:
00114         print("\nERROR: Please provide the number of instances to run on the Solver, the number of
      nodes to select the "
00115               "correct Neural Network and yes or no to run on hybrid mode or not.\nUsage: "
00116               "python3 HybridSolver.py <num instances> <num nodes> <y/n>  or\n"
00117               "python3 HybridSolver.py <num start instance>-<num end instance> <num nodes> <y/n>\n")
00118         sys.exit(1)
00119
00120     if not isinstance(sys.argv[1], str) or not isinstance(sys.argv[2], str) or not
      isinstance(sys.argv[3], str):
00121         print("ERROR: The arguments must be strings.")
00122         sys.exit(1)
```

```
00123
00124     num_instances = sys.argv[1]
00125     num_nodes = int(sys.argv[2])
00126     hyb_mode = (sys.argv[3] == "y" or sys.argv[3] == "Y" or sys.argv[3] == "yes" or sys.argv[3] ==
      "Yes")
00127
00128     hybrid_solver(num_instances, num_nodes, hyb_mode)
```

10.61 HybridTSPSolver/src/HybridSolver/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.

10.61.1 Detailed Description

Project main file, where you start the program, read the input file and print/write the results.

Author

Lorenzo Sciandra

Version

0.1.0

Date

2023-04-18

Copyright

Copyright (c) 2023, license MIT

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

Definition in file main.c.

10.61.2 Function Documentation

10.61.2.1 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.

10.62 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     char *input_file = argv[1];
00032     char *output_file = argv[2];
00033
00034     //printf("\nNodes :%d \t\tMode: %s\n", MAX_VERTEX_NUM, HYBRID ? "Hybrid" : "Classic");
00035
00036     freopen(output_file, "w+", stdout);
00037
00038     printf("\nReading from file '%s'\n", input_file);
00039     printf("\nWriting to file '%s'\n", output_file);
00040
00041     //run_all_tests();
00042
00043     static Problem new_problem;
00044
00045     //read_tsp_lib_file(&new_problem.graph, input_file);
00046     read_tsp_csv_file(&new_problem.graph, input_file);
00047
00048     //print_graph(&new_problem.graph);
00049
00050     branch_and_bound(&new_problem);
00051
00052     printf("\nOptimal tour found with candidate node = %i, elapsed time = %lfs and interrupted =
      %s\n",
00053             new_problem.candidateNodeId,  ((double) (new_problem.end - new_problem.start)) /
      CLOCKS_PER_SEC,
00054            new_problem.interrupted ? "TRUE" : "FALSE");
00055
00056     printf("\nB-&-B tree with generated BBNodes = %u,  explored BBNodes = %u and max tree level =
      %u\n",
00057            new_problem.generatedBBNodes, new_problem.exploredBBNodes, new_problem.totTreeLevels);
00058
00059     print_subProblem(&new_problem.bestSolution);
00060
00061     fclose(stdout);
00062
00063     return 0;
00064 }
```

10.63 HybridTSPSolver/src/HybridSolver/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 FLT_MAX

  The maximum number to set the initial value of Problem and SubProblem.
- #define MAX_EDGES_NUM (MAX_VERTEX_NUM ∗ (MAX_VERTEX_NUM - 1) / 2)

  The maximum number of edges in the Graph.
- #define INIT_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 APPROXIMATION 0.00001f

  The minimum value to consider two floats equal.
- #define EPSILON (INIT_UB / 1000)

  The first constant used to compare two SubProblem in the branch and bound algorithm.
- #define EPSILON2 (0.33f ∗ EPSILON)

  The second constant used to compare two SubProblem in the branch and bound algorithm.
- #define BETTER_PROB 0.1f

  The third constant used to compare two SubProblem in the branch and bound algorithm.
- #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.

10.63.1 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

0.1.0

Date

2023-04-18

Copyright

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

Definition in file problem_settings.h.

10.63.2 Macro Definition Documentation

10.63.2.1 APPROXIMATION

```
#define APPROXIMATION 0.00001f
```

The minimum value to consider two floats equal.

Definition at line 53 of file problem_settings.h.

10.63.2.2 BETTER_PROB

```
#define BETTER_PROB 0.1f
```

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 78 of file problem_settings.h.

10.63.2.3 EPSILON

```
#define EPSILON (INIT_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 61 of file problem_settings.h.

10.63.2.4 EPSILON2

```
#define EPSILON2 (0.33f * 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 69 of file problem_settings.h.

10.63.2.5 INFINITE

```
#define INFINITE FLT_MAX
```

The maximum number to set the initial value of Problem and SubProblem.

Definition at line 33 of file problem_settings.h.

10.63.2.6 INIT_UB

```
#define INIT_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 45 of file problem_settings.h.

10.63.2.7 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 41 of file problem_settings.h.

10.63.2.8 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 86 of file problem_settings.h.

10.63.2.9 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 90 of file problem_settings.h.

10.63.2.10 TIME_LIMIT_SECONDS

```
#define TIME_LIMIT_SECONDS 600
```

The maximum time to run the algorithm. Default: 10 minutes.

Definition at line 82 of file problem_settings.h.

10.63.2.11 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 49 of file problem_settings.h.

10.64 problem_settings.h

Go to the documentation of this file.

```
00001
00017 #ifndef BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
00018 #define BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
00019 #include <stdio.h>
00020 #include <limits.h>
00021 #include <float.h>
00022 #include <string.h>
00023 #include <stdarg.h>
00024 #include <stdbool.h>
00025 #include <stdlib.h>
00026 #include <math.h>
00027 #include <time.h>
00028 #include <errno.h>
00029 #include <pthread.h>
00030
00031
00033 #define INFINITE FLT_MAX
00034
00035
00036 // #define MAX_VERTEX_NUM 50 -- no longer in this file, but in the CMakeLists.txt to be able to change
      it from the command line.
00037 // #define HYBRID 0 -- no longer in this file, but in the CMakeLists.txt to be able to change it from
      the command line.
00038
00039
00041 #define MAX_EDGES_NUM (MAX_VERTEX_NUM * (MAX_VERTEX_NUM - 1) / 2)
00042
00043
00045 #define INIT_UB (sqrt(MAX_VERTEX_NUM) * 1.27f)
00046
00047
00049 #define TRACE() fprintf(stderr, "%s (%d): %s\n", __FILE__, __LINE__, __func__)
00050
00051
00053 #define APPROXIMATION 0.00001f
00054
00055
00057
00061 #define EPSILON (INIT_UB / 1000)
00062
00063
00065
00069 #define EPSILON2 (0.33f * EPSILON)
00070
00071
00073
00078 #define BETTER_PROB 0.1f
00079
00080
00082 #define TIME_LIMIT_SECONDS 600
00083
00084
00086 #define NUM_HK_INITIAL_ITERATIONS (((((float) MAX_VERTEX_NUM * MAX_VERTEX_NUM)/50) + 0.5f) +
      MAX_VERTEX_NUM + 15)
00087
00088
00090 #define NUM_HK_ITERATIONS (((float) MAX_VERTEX_NUM / 4) + 5)
00091
00092
00093 #endif //BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
```

10.65 HybridTSPSolver/src/HybridSolver/main/ReadME.md File Reference

10.66 HybridTSPSolver/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.

10.66.1 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

0.1.0

Date

2023-04-18

Copyright

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

Definition in file tsp_instance_reader.c.

10.66.2 Function Documentation

10.66.2.1 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.

10.66.2.2 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.

10.67 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             }
```

```
00059             nodes[i].positionInGraph = i;
00060             nodes[i].x = x;
00061             nodes[i].y = y;
00062             nodes[i].num_neighbours = 0;
00063             add_elem_list_bottom(nodes_list, &nodes[i]);
00064             i++;
00065         } else {
00066             end_of_file = true;
00067         }
00068     }
00069     free(line);
00070     if (fclose(fp) == EOF) {
00071         perror("Error while closing the file.\n");
00072         printf("\nFile: %s\n", filename);
00073         exit(1);
00074     }
00075     create_euclidean_graph(graph, nodes_list);
00076 }
00077
00078
00079 void read_tsp_csv_file(Graph *graph, char *filename) {
00080     FILE *fp = fopen(filename, "r");
00081     if (fp == NULL) {
00082         perror("Error while opening the file.\n");
00083         printf("\nFile: %s\n", filename);
00084         exit(1);
00085     }
00086     graph->cost = 0;
00087     graph->num_edges = 0;
00088     graph->num_nodes = 0;
00089     graph->kind = WEIGHTED_GRAPH;
00090     graph->orderedEdges = false;
00091     unsigned short i = 0;
00092     unsigned short z = 0;
00093     char *line = NULL;
00094     size_t len = 0;
00095     while (getline(&line, &len, fp) != -1) {
00096         graph->nodes[i].positionInGraph = i;
00097         graph->nodes[i].x = 0;
00098         graph->nodes[i].y = 0;
00099         graph->nodes[i].num_neighbours = 0;
00100         char *token = strtok(line, ";");
00101         unsigned short j = 0;
00102         while (token != NULL && strcmp(token, "\n") != 0) {
00103             if (j != i) {
00104                 double weight = 0, prob = 0;
00105
00106                 int result = sscanf(token, "(%lf, %lf)", &weight, &prob);
00107                 if (result != 2) {
00108                     perror("Error while reading the file.\n");
00109                     printf("\nFile: %s\n", filename);
00110                     exit(1);
00111                 }
00112
00113                 if (weight > 0) {
00114                     if (j > i) {
00115                         graph->nodes[i].neighbours[graph->nodes[i].num_neighbours] = j;
00116                         graph->nodes[i].num_neighbours++;
00117                         graph->num_edges++;
00118                         graph->edges[z].src = i;
00119                         graph->edges[z].dest = j;
00120                         graph->edges[z].prob = HYBRID ? prob : 0;
00121                         graph->edges[z].symbol = z + 1;
00122                         graph->edges[z].positionInGraph = z;
00123                         graph->edges[z].weight = weight;
00124                         graph->cost += graph->edges[z].weight;
00125                         graph->nodes[j].positionInGraph = j;
00126                         graph->edges_matrix[i][j] = graph->edges[z];
00127                         graph->edges_matrix[j][i] = graph->edges[z];
00128                         z++;
00129                     } else {
00130                         graph->nodes[i].neighbours[graph->nodes[i].num_neighbours] = j;
00131                         graph->nodes[i].num_neighbours++;
00132                     }
00133                 }
00134             }
00135             token = strtok(NULL, ";");
00136             j++;
00137         }
00138         graph->num_nodes++;
00139         i++;
00140     }
00141     free(line);
00142     if (fclose(fp) == EOF) {
00143         perror("Error while closing the file.\n");
00144         printf("\nFile: %s\n", filename);
00145         exit(1);
```

```
00146     }
00147 }
```

10.68 HybridTSPSolver/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.

10.68.1 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

0.1.0

Date

2023-04-18

Copyright

Copyright (c) 2023, license MIT

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

Definition in file tsp_instance_reader.h.

10.68.2 Function Documentation

10.68.2.1 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.

10.68.2.2 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.

10.69 tsp_instance_reader.h

Go to the documentation of this file.

```
00001
00015 #ifndef BRANCHANDBOUND1TREE_TSP_INSTANCE_READER_H
00016 #define BRANCHANDBOUND1TREE_TSP_INSTANCE_READER_H
00017 #include "data_structures/graph.h"
00018
00019
00025 void read_tsp_lib_file(Graph * graph, char * filename);
00026
00027
00033 void read_tsp_csv_file(Graph * graph, char * filename);
00034
00035
00036 #endif //BRANCHANDBOUND1TREE_TSP_INSTANCE_READER_H
```

Index