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 one.

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

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

1. **Candidate node selection**: to construct a 1Tree, a **candidate Node** must be chosen. The algorithm tries 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**: the algorithms needs an initial feasible solution to prune the search space using the bounding step. In the classical solver, this is accomplished by executing the nearest neighbor algorithm with each Node as the starting Node and then selecting the lowest tour found as the initial tour. The hybrid solver also uses a prob-nearest-neighbor algorithm. Starting from each node, it selects 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, 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 from lowest to highest. In the Hybrid Solver when two subproblems have the same value, the one with the highest probability is selected first. This procedure is extremely flexible, as it provides meta-parameters that allow for the modification of the subproblems sorting criterion, enabling any desired trade-off between the probability and the value of 1Trees.

1.5 Code Documentation

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

1.6 Results

Below are the mean values obtained from 100 instances for each graph size. The best value in each comparison is highlighted in bold:

	Classic Solver	Hybrid Solver
20 nodes 100 instances max 10 minutes		
Total time (s)	**0.028**	1.494
B-&-B time (s)	0.025	**0.020**
B-&-B tree depth	4.72	**3.94**
Generated B-&-B nodes	228.69	**147.95**
Explored B-&-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-&-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		

	Classic Solver	Hybrid Solver
Total time (s)	24.931	**16.512**
B-&-B time (s)	24.922	**14.633**
B-&-B tree depth	13.57	**12.44**
Generated B-&-B nodes	18384.37	**10519.63**
Explored B-&-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-&-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-&-B time (s)	188.870	**98.586**
B-&-B tree depth	14.37	**12.49**
Generated B-&-B nodes	37802.4	**13214.54**
Explored B-&-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-&-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

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

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

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

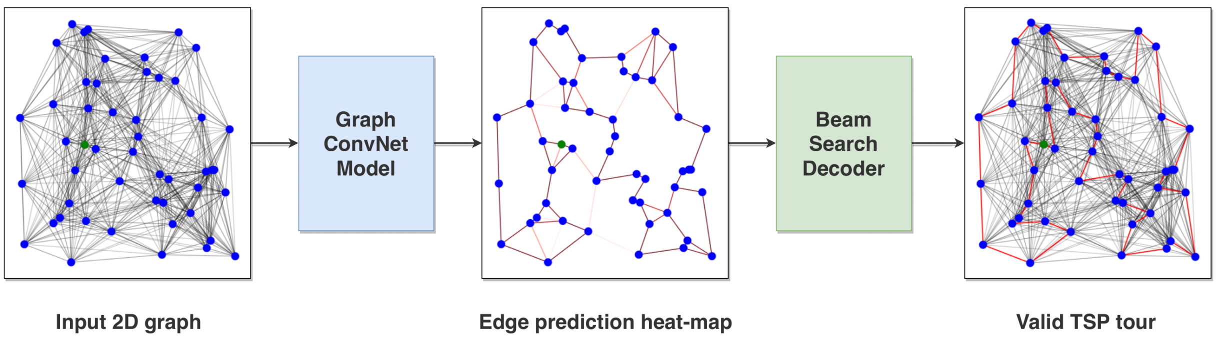


Figure 2.1 model-blocks

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 hybric
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 ir
@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 k
    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
```

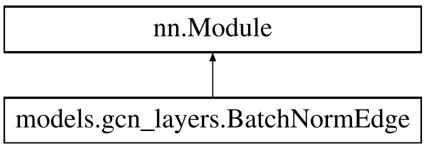
Definition at line 50 of file plot_utils.py.

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

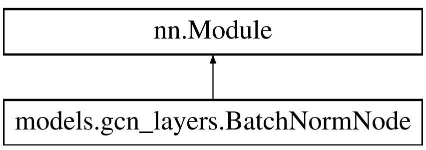
9.1.4.1 batch_norm

```
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

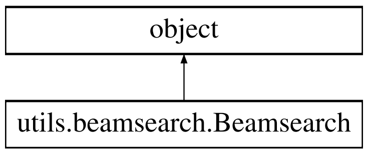
9.2.4.1 batch_norm

`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

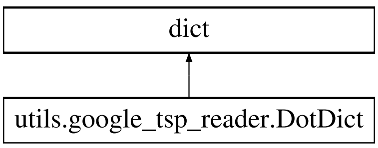
```
void* DllElem::value
```

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

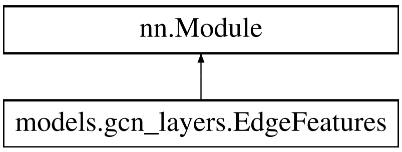
```
float Edge::weight
```

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

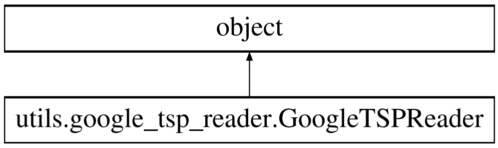
```
Set Forest::sets[MAX_VERTEX_NUM]
```

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

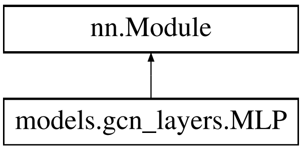
```
List* ListIterator::list
```

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

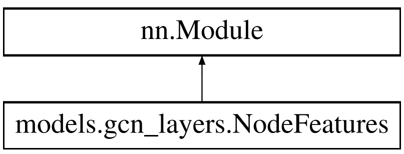
```
float Node::y
```

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

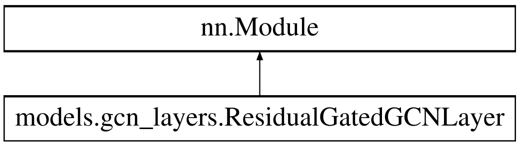
```
unsigned short Problem::totTreeLevels
```

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.

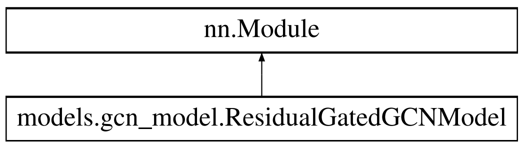
9.19.4.4 node_feat

`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

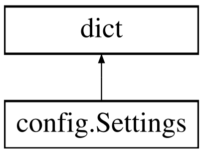
```
unsigned short Set::rango
```

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.

```

#include "branch_and_bound.h"


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

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

    add_elem_list_bottom(stack, &start);

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

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

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

    if (pathLength[toNode] > 2) {
        while (fromNode != start.positionInGraph) {
            fromNode = parentId[toNode];
```

```
            Edge current_in_cycle = problem->graph.edges_matrix[fromNode][toNode];
            subProblem->cycleEdges[subProblem->num_edges_in_cycle].src = current_in_cycle.src;
            subProblem->cycleEdges[subProblem->num_edges_in_cycle].dest = current_in_cycle.dest;
            subProblem->num_edges_in_cycle++;
            toNode = fromNode;
        }
    }

}


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


BBNodeType mst_to_one_tree(SubProblem *currentSubproblem, Graph *graph) {
    Node candidate = graph->nodes[problem->candidateNodeId];
    int bestEdgePos[candidate.num_neighbours];
    float bestNewEdgeWeight[candidate.num_neighbours];
    unsigned short src = candidate.positionInGraph;
    unsigned short others[candidate.num_neighbours];
    unsigned short num_others = 0;
    unsigned short add = 0;
    for (unsigned short i = 0; i < candidate.num_neighbours; i++) {
        unsigned short dest = candidate.neighbours[i];

        if (currentSubproblem->constraints[src][dest] == MANDATORY) {
            Edge mandatoryEdge = graph->edges_matrix[src][dest];
            bestEdgePos[add] = mandatoryEdge.positionInGraph;
            bestNewEdgeWeight[add] = mandatoryEdge.weight;
            add++;
        } else if (currentSubproblem->constraints[src][dest] == NOTHING) {
            others[num_others] = dest;
            num_others++;
        }
    }
    if (add > 2) {
        return CLOSED_UNFEASIBLE;
    } else {
        if (add < 2) {
            if (add == 0) {
                bestNewEdgeWeight[0] = INFINITE;
                bestEdgePos[0] = -1;
            }
            bestNewEdgeWeight[1] = INFINITE;
            bestEdgePos[1] = -1;

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

                if (add == 0) {
                    if ((bestNewEdgeWeight[0] - candidateEdge.weight) > APPROXIMATION) {
                        unsigned short temp_pos = bestEdgePos[0];
                        float temp_weight = bestNewEdgeWeight[0];
                        bestEdgePos[0] = candidateEdge.positionInGraph;
                        bestNewEdgeWeight[0] = candidateEdge.weight;
                        bestEdgePos[1] = temp_pos;
                        bestNewEdgeWeight[1] = temp_weight;

                    } else if ((bestNewEdgeWeight[1] - candidateEdge.weight) > APPROXIMATION) {
                        bestEdgePos[1] = candidateEdge.positionInGraph;
                        bestNewEdgeWeight[1] = candidateEdge.weight;
                    }
                } else if (add == 1) {
                    if ((bestNewEdgeWeight[1] - candidateEdge.weight) > APPROXIMATION) {
                        bestEdgePos[1] = candidateEdge.positionInGraph;
                        bestNewEdgeWeight[1] = candidateEdge.weight;
                    }
                }
            }
            if (bestNewEdgeWeight[0] == INFINITE || bestNewEdgeWeight[1] == INFINITE) {
                return CLOSED_UNFEASIBLE;
            }
        }

        Edge best_first = graph->edges[bestEdgePos[0]];
        add_edge(&currentSubproblem->oneTree, &best_first);

        Edge best_second = graph->edges[bestEdgePos[1]];
        add_edge(&currentSubproblem->oneTree, &best_second);

        return OPEN;
    }
}
```

```


void clean_matrix(SubProblem *subProblem) {
    for (short i = 0; i < MAX_VERTEX_NUM; i++) {
        for (short j = i; j < MAX_VERTEX_NUM; j++) {
            subProblem->constraints[i][j] = NOTHING;
            subProblem->constraints[j][i] = NOTHING;
        }
    }
}


void copy_constraints(SubProblem *subProblem, const SubProblem *otherSubProblem) {
    for (short i = 0; i < MAX_VERTEX_NUM; i++) {
        for (short j = i; j < MAX_VERTEX_NUM; j++) {
            subProblem->constraints[i][j] = otherSubProblem->constraints[i][j];
            subProblem->constraints[j][i] = otherSubProblem->constraints[j][i];
        }
    }

    for (short j = 0; j < otherSubProblem->num_forbidden_edges; j++) {
        subProblem->forbiddenEdges[j] = otherSubProblem->forbiddenEdges[j];
    }

    for (short k = 0; k < otherSubProblem->num_mandatory_edges; k++) {
        subProblem->mandatoryEdges[k] = otherSubProblem->mandatoryEdges[k];
    }
}


// a better than b?
bool compare_subproblems(const SubProblem *a, const SubProblem *b) {
    if (HYBRID) {
        return ((b->value - a->value) > EPSILON) ||
               (((b->value - a->value) > EPSILON2) && ((a->prob - b->prob) >= BETTER_PROB));
    } else {
        return (b->value - a->value) > EPSILON;
    }
}


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

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

    for (unsigned short i = 0; i < currentSubProblem->num_edges_in_cycle; i++) {

        ConstrainedEdge current_cycle_edge = currentSubProblem->cycleEdges[i];

        if (currentSubProblem->constraints[current_cycle_edge.src][current_cycle_edge.dest] ==
NOTHING) {
            problem->generatedBBNodes++;
            SubProblem child;
            child.num_edges_in_cycle = 0;
            child.type = OPEN;
            child.prob = currentSubProblem->prob;
            child.id = problem->generatedBBNodes;
            child.value = currentSubProblem->value;
            child.treeLevel = currentSubProblem->treeLevel + 1;
            child.num_forbidden_edges = currentSubProblem->num_forbidden_edges;
            child.num_mandatory_edges = currentSubProblem->num_mandatory_edges;
            copy_constraints(&child, currentSubProblem);

            child.forbiddenEdges[currentSubProblem->num_forbidden_edges].src = current_cycle_edge.src;
            child.forbiddenEdges[currentSubProblem->num_forbidden_edges].dest =
current_cycle_edge.dest;
            child.constraints[current_cycle_edge.src][current_cycle_edge.dest] = FORBIDDEN;
            child.constraints[current_cycle_edge.dest][current_cycle_edge.src] = FORBIDDEN;

            child.num_forbidden_edges++;

            for (unsigned short z = 0; z < i; z++) {
                ConstrainedEdge mandatory = currentSubProblem->cycleEdges[z];

                if (currentSubProblem->constraints[mandatory.src][mandatory.dest] == NOTHING) {
                    child.mandatoryEdges[child.num_mandatory_edges].src = mandatory.src;
                    child.mandatoryEdges[child.num_mandatory_edges].dest = mandatory.dest;
                    child.num_mandatory_edges++;
                    child.constraints[mandatory.src][mandatory.dest] = MANDATORY;
                    child.constraints[mandatory.dest][mandatory.src] = MANDATORY;
                }
            }

            long position = -1;

```

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

        }
    }

}


void held_karp_bound(SubProblem *currentSubProb) {

    if (compare_subproblems(currentSubProb, &problem->bestSolution) || currentSubProb->treeLevel == 0)
{
        problem->exploredBBNodes++;
        float pi[MAX_VERTEX_NUM] = {0};
        float v[MAX_VERTEX_NUM] = {0};
        float v_old[MAX_VERTEX_NUM] = {0};
        float total_pi = 0;
        int max_iter = currentSubProb->treeLevel == 0 ? (int) NUM_HK_INITIAL_ITERATIONS : (int)
NUM_HK_ITERATIONS;
        float best_lower_bound = currentSubProb->value;
        BBNodeType type = currentSubProb->type;
        float t_0;
        SubProblemsList generatedSubProblems;
        new_SubProblemList(&generatedSubProblems);
        Graph *used_graph = &problem->graph;
        bool first_iter = true;
        currentSubProb->timeToReach = ((float) (clock() - problem->start)) / CLOCKS_PER_SEC;

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

            SubProblem analyzedSubProblem = *currentSubProb;

            for (unsigned short j = 0; j < problem->graph.num_edges; j++) {
                if ((pi[used_graph->edges[j].src] +
                     pi[used_graph->edges[j].dest]) != 0) {
                    used_graph->edges[j].weight += (pi[used_graph->edges[j].src] +
                                                    pi[used_graph->edges[j].dest]);

used_graph->edges_matrix[used_graph->edges[j].src][used_graph->edges[j].dest].weight =
used_graph->edges[j].weight;

used_graph->edges_matrix[used_graph->edges[j].dest][used_graph->edges[j].src].weight =
used_graph->edges[j].weight;
                    used_graph->orderedEdges = false;
                }
            }

            kruskal_constrained(used_graph, &analyzedSubProblem.oneTree, problem->candidateNodeId,
                                analyzedSubProblem.forbiddenEdges,
analyzedSubProblem.num_forbidden_edges,
                                analyzedSubProblem.mandatoryEdges,
analyzedSubProblem.num_mandatory_edges);

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

                if (type == OPEN) {

                    analyzedSubProblem.value = 0;
                    analyzedSubProblem.prob = analyzedSubProblem.oneTree.prob;
                    analyzedSubProblem.type = type;

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

                    bool better_value = compare_subproblems(&analyzedSubProblem,
&problem->bestSolution);
                    if (!better_value) {
                        analyzedSubProblem.type = CLOSED_BOUND;
                    } else {
                        analyzedSubProblem.num_edges_in_cycle = 0;
```

```
                        if (check_hamiltonian(&analyzedSubProblem)) {
                            problem->bestValue = analyzedSubProblem.value;
                            analyzedSubProblem.type = CLOSED_NEW_BEST;
                            problem->bestSolution = analyzedSubProblem;
                        } else {
                            analyzedSubProblem.type = OPEN;
                        }
                    }

                    float current_value = analyzedSubProblem.oneTree.cost - (2 * total_pi);

                    if (current_value > best_lower_bound || first_iter) {
                        best_lower_bound = current_value;
                        t_0 = best_lower_bound / (2 * MAX_VERTEX_NUM);
                        if (first_iter) {
                            first_iter = false;
                            used_graph = &problem->reformulationGraph;
                        }

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

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

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

                    total_pi = 0;

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

        }

        float best_value = -1;
        SubProblem *best_found = NULL;
        SubProblemsListIterator *subProblem_iterators = create_SubProblemList_iterator(&generatedSubProblems);
        for (size_t j = 0; j < generatedSubProblems.size; j++) {
            SubProblem *generatedSubProblem = SubProblemList_iterator_get_next(subProblem_iterators);
            if (generatedSubProblem->value > best_value &&
                generatedSubProblem->value <= best_lower_bound &&
                generatedSubProblem->type != CLOSED_UNFEASIBLE) {
                best_value = generatedSubProblem->value;
                best_found = generatedSubProblem;
            }
        }
        *currentSubProb = best_found == NULL ? *currentSubProb : *best_found;
        delete_SubProblemList_iterator(subProblem_iterators);
        delete_SubProblemList(&generatedSubProblems);
    } else {
        currentSubProb->type = CLOSED_BOUND;
    }
}


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


void nearest_prob_neighbour(unsigned short start_node) {
    SubProblem nn_subProblem;
    nn_subProblem.num_forbidden_edges = 0;
    nn_subProblem.num_mandatory_edges = 0;
    nn_subProblem.num_edges_in_cycle = 0;
    nn_subProblem.timeToReach = ((float) (clock() - problem->start)) / CLOCKS_PER_SEC;
    create_mst(&nn_subProblem.oneTree, problem->graph.nodes, problem->graph.num_nodes);
```

```
    unsigned short current_node = start_node;
    bool visited[MAX_VERTEX_NUM] = {false};
    ConstrainedEdge cycleEdge;

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

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

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

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

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

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

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

```c
        prob_nn_subProblem.oneTree.isValid = true;
        prob_nn_subProblem.type = CLOSED_HAMILTONIAN;
        prob_nn_subProblem.prob = prob_nn_subProblem.oneTree.prob;

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

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

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

}


unsigned short find_candidate_node(void) {
    SubProblemsList findCandidateSubProblems;
    new_SubProblemList(&findCandidateSubProblems);
    ConstrainedEdge *forbiddenEdges = NULL;
    ConstrainedEdge *mandatoryEdges = NULL;

    for (unsigned short i = 0; i < problem->graph.num_nodes; i++) {
        SubProblem currentCandidate;
        clean_matrix(&currentCandidate);
        nearest_prob_neighbour(i);
        kruskal_constrained(&problem->graph, &currentCandidate.oneTree, i, forbiddenEdges, 0, mandatoryEdges, 0);
        mst_to_one_tree(&currentCandidate, &problem->graph);
        currentCandidate.value = currentCandidate.oneTree.cost;
        currentCandidate.prob = currentCandidate.oneTree.prob;
        add_elem_SubProblemList_bottom(&findCandidateSubProblems, &currentCandidate);
    }

    unsigned short best_candidate = 0;
    SubProblem *best_subProblem = NULL;
    SubProblemsListIterator *subProblems_iterators = create_SubProblemList_iterator(&findCandidateSubProblems);
    for (unsigned short j = 0; j < problem->graph.num_nodes; j++) {
        SubProblem *current_subProblem = SubProblemList_iterator_get_next(subProblems_iterators);
        if (best_subProblem == NULL || compare_subproblems(best_subProblem, current_subProblem)) {
            best_candidate = j;
            best_subProblem = current_subProblem;
        }
    }
    delete_SubProblemList_iterator(subProblems_iterators);
    delete_SubProblemList(&findCandidateSubProblems);
    return best_candidate;
}


bool check_feasibility(Graph *graph) {

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


void branch_and_bound(Problem *current_problem) {

    problem = current_problem;

    if (check_feasibility(&problem->graph)) {
        problem->start = clock();
        problem->bestValue = INFINITE;
        problem->candidateNodeId = find_candidate_node();
        problem->exploredBBNodes = 0;
        problem->generatedBBNodes = 0;
        problem->totTreeLevels = 0;
        problem->interrupted = false;
        problem->reformulationGraph = problem->graph;

```

```
        SubProblem subProblem;
        subProblem.treeLevel = 0;
        subProblem.id = problem->generatedBBNodes;
        subProblem.type = OPEN;
        subProblem.prob = 0;
        subProblem.value = INFINITE;
        subProblem.num_edges_in_cycle = 0;
        subProblem.num_forbidden_edges = 0;
        subProblem.num_mandatory_edges = 0;
        problem->generatedBBNodes++;
        clean_matrix(&subProblem);

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

        while (subProblems.size != 0 && !time_limit_reached()) {
            SubProblem current_sub_problem = *get_SubProblemList_elem_index(&subProblems, 0);
            delete_SubProblemList_elem_index(&subProblems, 0);
            held_karp_bound(&current_sub_problem);
            if (current_sub_problem.type == OPEN) {
                branch(&subProblems, &current_sub_problem);
            }
        }

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

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


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


void print_subProblem(const SubProblem *subProblem) {

    char *type;
    if (subProblem->type == OPEN) {
        type = "OPEN";
    } else if (subProblem->type == CLOSED_UNFEASIBLE) {
        type = "CLOSED_UNFEASIBLE";
    } else if (subProblem->type == CLOSED_BOUND) {
        type = "CLOSED_BOUND";
    } else if (subProblem->type == CLOSED_HAMILTONIAN) {
        type = "CLOSED_HAMILTONIAN";
    } else {
        type = "CLOSED_NEW_BEST";
    }
    printf("\nSUBPROBLEM with cost = %lf, type = %s, level of the BB tree = %i, prob = %lf, BBNode number = %u and time to obtain = %lfs",
           subProblem->value, type, subProblem->treeLevel, subProblem->oneTree.prob, subProblem->id,
           subProblem->timeToReach);

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

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

    printf("\n%i Mandatory edges:", subProblem->num_mandatory_edges);
    for (unsigned short j = 0; j < subProblem->num_mandatory_edges; j++) {
        ConstrainedEdge mandatory = subProblem->mandatoryEdges[j];
        printf("  %i <-> %i ", mandatory.src, mandatory.dest);
    }

    printf("\n%i Forbidden edges:", subProblem->num_forbidden_edges);
    for (unsigned short z = 0; z < subProblem->num_forbidden_edges; z++) {
        ConstrainedEdge forbidden = subProblem->forbiddenEdges[z];
        printf("  %i <-> %i ", forbidden.src, forbidden.dest);
    }
    printf("\n");
}
```

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);
```

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

#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.

```

#include "kruskal.h"


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

    Edge * edges = graph->edges;

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

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


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

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


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

    if ((last_edge.weight - first_edge.weight) > APPROXIMATION) {
        if ((last_edge.weight - middle_edge.weight) > APPROXIMATION) {
            if ((middle_edge.weight - first_edge.weight) > APPROXIMATION) {
                pivot = middle_edge.weight;
                swap(graph, first, middle);
            }
        } else {
            pivot = last_edge.weight;
            swap(graph, first, last);
        }
    } else {
        if ((last_edge.weight - middle_edge.weight ) > APPROXIMATION) {
            pivot = last_edge.weight;
            swap(graph, first, last);

        } else if ((first_edge.weight - middle_edge.weight) > APPROXIMATION) {
            pivot = middle_edge.weight;
            swap(graph, first, middle);
        }
    }
    unsigned short j = last;
    unsigned short i = first + 1;
    bool condition = true;
    while (condition) {
        Edge i_edge = edges[i];
        while (i <= j && (pivot - i_edge.weight) >= -APPROXIMATION) {
            i += 1;
            i_edge = edges[i];
        }
        Edge j_edge = edges[j];
        while (i <= j && (j_edge.weight - pivot) > APPROXIMATION) {
            j -= 1;
            j_edge = edges[j];
        }
        if (i <= j) {
            swap(graph, i,  j);
        } else {
            condition = false;
        }

    }

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

    return j;

```

```c
}


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


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


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

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

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

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


void kruskal_constrained(Graph * graph, MST * oneTree, unsigned short candidateId, const
ConstrainedEdge * forbiddenEdges,
                         unsigned short numForbidden, const ConstrainedEdge * mandatoryEdges, unsigned
short numMandatory) {
    create_mst(oneTree,graph->nodes, graph->num_nodes);
    Forest forest;
    create_forest_constrained(&forest, graph->nodes, graph->num_nodes, candidateId);
    wrap_quick_sort(graph);

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

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

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

    bool isForbidden = false;
    unsigned short num_edges_inG = 0;

```

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

        Edge current_edge = graph->edges[num_edges_inG];

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

        if (src_id != candidateId && dest_id != candidateId) {

            for (unsigned short j = 0; !isForbidden && j < numForbidden; j++) {
                ConstrainedEdge current_mandatory = forbiddenEdges[j];
                Edge forbidden_edge =
graph->edges_matrix[current_mandatory.src][current_mandatory.dest];

                if (forbidden_edge.src != candidateId && forbidden_edge.dest != candidateId) {
                    if (current_edge.symbol == forbidden_edge.symbol) {
                        isForbidden = true;
                    }
                }
            }

            if (!isForbidden) {
                // get the edge with the minimum weight
                unsigned short src = current_edge.src;
                unsigned short dest = current_edge.dest;

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

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

        num_edges_inG++;
    }

    if (num_edges_inMST == graph->num_nodes - 2) {
        oneTree->isValid = true;
    }
}
```

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.

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


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



static int pivot_quicksort(Graph * graph, unsigned short first, unsigned short last);



static void quick_sort(Graph * graph, unsigned short first, unsigned short last);


void wrap_quick_sort(Graph * graph);



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



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


#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;
```

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


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

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

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


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


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

    return e;
}


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


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

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

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


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

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

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

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

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

```
    }

    list->size++;
}


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

    SubProblemElem *oldElem;
    oldElem = list->head;

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

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

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


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

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

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

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

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

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

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

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

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

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

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

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

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

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.

```
#ifndef BRANCHANDBOUND1TREE_B_AND_B_DATA_H
#define BRANCHANDBOUND1TREE_B_AND_B_DATA_H

#include "mst.h"

typedef enum BBNodeType{
    OPEN,
    CLOSED_BOUND,
    CLOSED_HAMILTONIAN,
    CLOSED_UNFEASIBLE,
    CLOSED_NEW_BEST
}BBNodeType;


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


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


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


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


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


typedef struct {
```

```
    SubProblemsList * list;
    SubProblemElem * curr;
    size_t index;
} SubProblemsListIterator;
void new_SubProblemList(SubProblemsList * list);
void delete_SubProblemList(SubProblemsList * list);
bool is_SubProblemList_empty(SubProblemsList *list);
size_t get_SubProblemList_size(SubProblemsList *list);
void add_elem_SubProblemList_bottom(SubProblemsList *list, SubProblem *element);
void add_elem_SubProblemList_index(SubProblemsList *list, SubProblem *element, size_t index);
void delete_SubProblemList_elem_index(SubProblemsList *list, size_t index);
SubProblem *get_SubProblemList_elem_index(SubProblemsList *list, size_t index);
SubProblemsListIterator  *create_SubProblemList_iterator(SubProblemsList *list);
bool is_SubProblemList_iterator_valid(SubProblemsListIterator *iterator);
SubProblem *SubProblemList_iterator_get_next(SubProblemsListIterator *iterator);
void delete_SubProblemList_iterator(SubProblemsListIterator *iterator);
#endif //BRANCHANDBOUND1TREE_B_AND_B_DATA_H
```

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
```

```
    unsigned short numEdges = 0;
    ListIterator *edges_iterator = create_list_iterator(edges_list);
    for (size_t i = 0; i < edges_list->size ; i++) {
        //add the source vertex to the data_structures
        Edge * current_edge = list_iterator_get_next(edges_iterator);
        unsigned short src = current_edge->src;
        unsigned short dest = current_edge->dest;

        graph->edges[numEdges].dest = dest;
        graph->edges[numEdges].src = src;
        graph->edges[numEdges].prob = current_edge->prob;
        graph->edges[numEdges].weight = current_edge->weight;
        graph->edges[numEdges].symbol = current_edge->symbol;
        graph->edges[numEdges].positionInGraph =  numEdges;
        graph->nodes[src].neighbours[graph->nodes[src].num_neighbours] = dest;
        graph->nodes[src].num_neighbours++;
        graph->edges_matrix[src][dest] = graph->edges[numEdges];
        graph->cost += current_edge->weight;

        graph->edges_matrix[dest][src] = graph->edges_matrix[src][dest];
        graph->nodes[dest].neighbours[graph->nodes[dest].num_neighbours] = src;
        graph->nodes[dest].num_neighbours++;

        numEdges++;
        graph->num_edges++;
    }
    delete_list_iterator(edges_iterator);
    del_list(edges_list);
    del_list(nodes_list);
}


void create_euclidean_graph(Graph * graph, List *nodes) {
    List *edges_list = new_list();

    unsigned short z = 0;
    Edge edges [MAX_EDGES_NUM];
    ListIterator *i_nodes_iterator = create_list_iterator(nodes);
    for (size_t i = 0; i < nodes->size; i++) {
        Node *node_src = list_iterator_get_next(i_nodes_iterator);
        for (size_t j = i + 1; j < nodes->size; j++) {
            Node *node_dest = get_list_elem_index(nodes,j);

            edges[z].src = node_src->positionInGraph;
            edges[z].dest = node_dest->positionInGraph;
            edges[z].symbol = z + 1;
            edges[z].positionInGraph = z;
            edges[z].prob = 0;
            edges[z].weight = (float) sqrt(pow(fabsf(node_src->x - node_dest->x), 2) +
                                           pow(fabsf(node_src->y - node_dest->y), 2));
            add_elem_list_bottom(edges_list, &edges[z]);
            z++;

        }

    }
    delete_list_iterator(i_nodes_iterator);

    create_graph(graph,nodes, edges_list, WEIGHTED_GRAPH);
}


void print_graph(const Graph *G) {
    printf("Nodes: %i\n", G->num_nodes);
    for (int i = 0; i < G->num_nodes; i++) {
        Node curr = G->nodes[i];
        printf("Node%i:\t(%.3f, %.3f)\t%i neighbours: ", curr.positionInGraph, curr.x, curr.y, curr.num_neighbours);

        for (int z = 0; z < curr.num_neighbours; z++) {
            printf("%i ", G->nodes[curr.neighbours[z]].positionInGraph);
        }
        printf("\n");
    }

    printf("\nCost: %lf\n", G->cost);
    printf("\nEdges: %i\n", G->num_edges);

    double dim = (log(G->num_nodes) / log(10) + 1) * 2 + 7;
    for (unsigned short j = 0; j < G->num_edges; j++) {
        char edge_print [(int) dim] ;
        char edge_print_dest [(int) (dim-7)/2] ;
        Edge curr = G->edges[j];
        sprintf(edge_print, "%i", curr.src);
        strcat(edge_print, " <--> ");
        sprintf(edge_print_dest, "%i", curr.dest);
        strcat(edge_print, edge_print_dest);
```

```

        printf("Edge%i:\t%s\tweight = %lf\tprob = %lf\n",
               curr.symbol,
               edge_print,
               curr.weight,
               curr.prob);
    }

}
```

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.

```c

#ifndef BRANCHANDBOUND_1TREE_GRAPH_H
#define BRANCHANDBOUND_1TREE_GRAPH_H
#include "./linked_list/linked_list.h"
#include "./linked_list/list_iterator.h"
#include "./linked_list/list_functions.h"
#include "../problem_settings.h"


typedef enum GraphKind{
    WEIGHTED_GRAPH,
    UNWEIGHTED_GRAPH
} GraphKind;


typedef struct Node {
    float x;
    float y;
    unsigned short positionInGraph;
    unsigned short num_neighbours;
    unsigned short neighbours [MAX_VERTEX_NUM - 1];
}Node;


typedef struct Edge {
    unsigned short src;
    unsigned short dest;
    unsigned short symbol;
    float weight;
    float prob;
    unsigned short positionInGraph;
}Edge;


typedef struct Graph {
    GraphKind kind;
    float cost;
    unsigned short num_nodes;
    unsigned short num_edges;
    bool orderedEdges;
    Node nodes [MAX_VERTEX_NUM];
    Edge edges [MAX_EDGES_NUM];
    Edge edges_matrix [MAX_VERTEX_NUM] [MAX_VERTEX_NUM];
}Graph;


void create_graph(Graph* graph, List *  nodes, List *  edges, GraphKind kind);


void create_euclidean_graph(Graph * graph, List * nodes);


void print_graph(const Graph * graph);


#endif //BRANCHANDBOUND_1TREE_GRAPH_H
```

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

A double linked list implementation.

```c
#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);
```

```

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


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

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

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

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

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

    list->size++;
}


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

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

    DllElem *oldTail = list->tail;

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

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


/*
 * This method iteratively finds and deletes the element at the specified index, but only if it doesn't exceed
 * the size of the List. In this case, instead, no reference gets deleted.
 */
void delete_list_elem_index(List *list, size_t index) {
    if (list == NULL || is_list_empty(list) || index >= get_list_size(list)) {
        return;
    }

    DllElem *oldElem;
    oldElem = list->head;

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

    // Found index  to remove!!
```

```
    if (oldElem != list->head) {
        oldElem->prev->next = oldElem->next;
        if (oldElem->next != NULL) {
            oldElem->next->prev = oldElem->prev;
        } else {
            list->tail = oldElem->prev;
        }
    } else {
        if (list->head == list->tail) {
            list->head = list->tail = NULL;
        } else {
            list->head = list->head->next;
            list->head->prev = NULL;
        }
    }

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


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

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

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

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

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.

```
#ifndef BRANCHANDBOUND1TREE_LIST_FUNCTIONS_H
#define BRANCHANDBOUND1TREE_LIST_FUNCTIONS_H

#include "linked_list.h"


List *new_list(void);


void del_list(List *list);


bool is_list_empty(List *list);


size_t get_list_size(List *list);


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


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


void delete_list_elem_bottom(List *list);


void delete_list_elem_index(List *list, size_t index);


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


#endif //BRANCHANDBOUND1TREE_LIST_FUNCTIONS_H
```

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

I	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 }
```

```


void *get_current_list_iterator_element(ListIterator *iterator) {
    return (iterator && iterator->curr && iterator->curr->value) ? iterator->curr->value : NULL;
}


bool is_list_iterator_valid(ListIterator *iterator) {
    return (iterator) ? iterator->index < get_list_size(iterator->list) : 0;
}


void list_iterator_next(ListIterator *iterator) {
    if (is_list_iterator_valid(iterator)) {
        iterator->index++;

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


void delete_list_iterator(ListIterator *iterator) {
    free(iterator);
}


void *list_iterator_get_next(ListIterator *iterator) {
    if (!is_list_iterator_valid(iterator)) {
        return NULL;
    }

    void *element = get_current_list_iterator_element(iterator);
    list_iterator_next(iterator);
    return element;
}
```

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

I	The ListIterator considered.

Definition at line 40 of file list_iterator.c.

10.26 list_iterator.h

Go to the documentation of this file.

```
#ifndef BRANCHANDBOUND1TREE_LIST_ITERATOR_H
#define BRANCHANDBOUND1TREE_LIST_ITERATOR_H
#include "linked_list.h"
ListIterator *create_list_iterator(List *list);
bool is_list_iterator_valid(ListIterator *iterator);
void *get_current_list_iterator_element(ListIterator *iterator);
```

```

void list_iterator_next(ListIterator *iterator);


void *list_iterator_get_next(ListIterator *iterator);


void delete_list_iterator(ListIterator *iterator);


#endif //BRANCHANDBOUND1TREE_LIST_ITERATOR_H
```

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.

```c
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.
candidateId	Id of the Node in the List to be excluded from the Forest.
forest	Pointer to the Forest to be initialized.

Definition at line 17 of file mfset.c.

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.

```

#ifndef BRANCHANDBOUND1TREE_MFSET_H
#define BRANCHANDBOUND1TREE_MFSET_H
#include "graph.h"


typedef struct Set {
    struct Set * parentSet;
    unsigned short rango;
    Node curr;
    unsigned short num_in_forest;
}Set;


typedef struct Forest {
    unsigned short num_sets;
    Set sets [MAX_VERTEX_NUM];
}Forest;


void create_forest(Forest * forest, const Node * nodes, unsigned short num_nodes);


void create_forest_constrained(Forest * forest, const Node * nodes, unsigned short num_nodes, unsigned
short candidateId);


void merge(Set * set1, Set * set2);


Set* find(Set * set);


void print_forest(const Forest * forest);


#endif //BRANCHANDBOUND1TREE_MFSET_H
```

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] ;
```

```
        const Edge * curr = &tree->edges[i];
        sprintf(edge_print, "%i", curr->src);
        strcat(edge_print, " <--> ");
        sprintf(edge_print_dest, "%i", curr->dest);
        strcat(edge_print, edge_print_dest);
        printf("Edge%i:\t%s\tweight = %.2lf\tprob = %lf\n",
               curr->symbol,
               edge_print,
               curr->weight,
               curr->prob);
    }

}


void print_mst_original_weight(const MST * tree, const Graph * graph){
    printf("\nMST or 1-Tree with cost: %f and validity = %s\n", tree->cost, tree->isValid ? "TRUE" :
"FALSE");

    double dim = (log(tree->num_nodes) / log(10) + 1) * 2 + 7;
    for (unsigned short i = 0; i < tree->num_edges; i++) {
        char edge_print [(int) dim] ;
        char edge_print_dest [(int) (dim-7)/2] ;
        const Edge * curr = &tree->edges[i];
        sprintf(edge_print, "%i", curr->src);
        strcat(edge_print, " <--> ");
        sprintf(edge_print_dest, "%i", curr->dest);
        strcat(edge_print, edge_print_dest);
        printf("Edge%i: %s weight = %f prob = %f\n",
               curr->symbol,
               edge_print,
               graph->edges_matrix[curr->src][curr->dest].weight,
               curr->prob);
    }
}
```

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.

```

#ifndef BRANCHANDBOUND1TREE_MST_H
#define BRANCHANDBOUND1TREE_MST_H
#include "mfset.h"


typedef struct ConstrainedEdge{
    unsigned short src;
    unsigned short dest;
}ConstrainedEdge;


typedef struct MST{
    bool isValid;
    float cost;
    float prob;
    unsigned short num_nodes;
    unsigned short num_edges;
    Node nodes [MAX_VERTEX_NUM];
    Edge edges [MAX_VERTEX_NUM];
}MST;
```

```
void create_mst(MST* mst, const Node * nodes, unsigned short num_nodes);


void add_edge(MST * tree, const Edge * edge);


void print_mst(const MST * mst);


void print_mst_original_weight(const MST * mst, const Graph * graph);


#endif //BRANCHANDBOUND1TREE_MST_H
```

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.

```
import json


class Settings(dict):
    """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
    """

    def __init__(self, config_dict):
        super().__init__()
        for key in config_dict:
            self[key] = config_dict[key]

    def __getattr__(self, attr):
        return self[attr]

    def __setitem__(self, key, value):
        return super().__setitem__(key, value)

    def __setattr__(self, key, value):
        return self.__setitem__(key, value)

    __delattr__ = dict.__delitem__


def get_default_config():
    """Returns default settings object.
    """
    return Settings(json.load(open("./configs/default.json")))


def get_config(filepath):
    """Returns settings from json file.
    """
    config = get_default_config()
    config.update(Settings(json.load(open(filepath))))
    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.

```
"""
    @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
"""


import os
import sys
import time
import numpy as np
import torch
from torch.autograd import Variable
import torch.nn.functional as F
import torch.nn as nn
from sklearn.utils.class_weight import compute_class_weight
# Remove warning
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
from scipy.sparse import SparseEfficiencyWarning
warnings.simplefilter('ignore', SparseEfficiencyWarning)
from config import *
from utils.graph_utils import *
from utils.google_tsp_reader import GoogleTSPReader
from utils.plot_utils import *
from models.gcn_model import ResidualGatedGCNModel
from utils.model_utils import *


def 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.
    """
    # Set evaluation mode
    net.eval()

    # Assign parameters
    num_nodes = config.num_nodes
    num_neighbors = config.num_neighbors
    batch_size = config.batch_size
    test_filepath = config.test_filepath

    # Load TSP data
    dataset = GoogleTSPReader(num_nodes, num_neighbors, batch_size=batch_size, filepath=test_filepath)

    # Convert dataset to iterable
    dataset = iter(dataset)

    # Initially set loss class weights as None
    edge_cw = None

    y_probs = []

    # read the instance number line from the test_filepath
    instance = None
    with open(test_filepath, 'r') as f:
        for i, line in enumerate(f):
            if i == instance_number:
                instance = line
                break

    # split the instance before the "output" part
    instance = instance.split(" output")[0]
    # create a list of the nodes spliting by spaces and convert to double
    instance = [float(x) for x in instance.split(" ")]

    with torch.no_grad():

```

```
        batch = next(dataset)

        while batch.nodes_coord.flatten().tolist() != instance:
            batch = next(dataset)

        x_edges = Variable(torch.LongTensor(batch.edges).type(dtypeLong), requires_grad=False)
        x_edges_values = Variable(torch.FloatTensor(batch.edges_values).type(dtypeFloat),
requires_grad=False)
        x_nodes = Variable(torch.LongTensor(batch.nodes).type(dtypeLong), requires_grad=False)
        x_nodes_coord = Variable(torch.FloatTensor(batch.nodes_coord).type(dtypeFloat),
requires_grad=False)
        y_edges = Variable(torch.LongTensor(batch.edges_target).type(dtypeLong), requires_grad=False)

        # Compute class weights (if uncomputed)
        if type(edge_cw) != torch.Tensor:
            edge_labels = y_edges.cpu().numpy().flatten()
            edge_cw = compute_class_weight("balanced", classes=np.unique(edge_labels), y=edge_labels)

        y_preds, _ = net.forward(x_edges, x_edges_values, x_nodes, x_nodes_coord, y_edges, edge_cw)
        y = F.softmax(y_preds, dim=3)
        # y_bins = y.argmax(dim=3)
        y_probs = y[:, :, :, 1]

    return y_probs, x_edges_values


def 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.
    """
    # Convert to numpy
    num_nodes = y_probs.shape[1]
    y_probs = y_probs.flatten().numpy()
    x_edges_values = x_edges_values.flatten().numpy()

    # stack the arrays horizontally and convert to string data type
    arr_combined = np.stack((x_edges_values, y_probs), axis=1).astype('U')

    # format the strings using a list comprehension
    arr_strings = np.array(['({}, {});'.format(x[0], x[1]) for x in arr_combined])

    filepath = filepath.replace(".csv", "_temp.csv")
    # write arr_strings to file
    with open(filepath, 'w') as f:
        edge = 0
        for item in arr_strings:
            if (edge + 1) % num_nodes == 0:
                f.write("%s\n" % item)
            else:
                f.write("%s" % item)
            edge += 1


def 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.
    """
    config_path = "./logs/tsp" + num_nodes + "/config.json"
    config = get_config(config_path)

    config.gpu_id = "0"
    config.accumulation_steps = 1
    config.val_filepath = "./data/hyb_tsp_" + num_nodes + "/test_100_instances.txt"
    config.test_filepath = "./data/hyb_tsp_" + num_nodes + "/test_100_instances.txt"

    os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
    os.environ["CUDA_VISIBLE_DEVICES"] = str(config.gpu_id)

    if torch.cuda.is_available():
        # print("CUDA available, using GPU ID {}".format(config.gpu_id))
        dtypeFloat = torch.cuda.FloatTensor
        dtypeLong = torch.cuda.LongTensor
        torch.cuda.manual_seed(1)
    else:
        # print("CUDA not available")
        dtypeFloat = torch.FloatTensor
        dtypeLong = torch.LongTensor
        torch.manual_seed(1)
```

```
    net = nn.DataParallel(ResidualGatedGCNModel(config, dtypeFloat, dtypeLong))
    if torch.cuda.is_available():
        net.cuda()

    log_dir = f"./logs/{config.expt_name}/"
    if torch.cuda.is_available():
        checkpoint = torch.load(log_dir + "best_val_checkpoint.tar")
    else:
        checkpoint = torch.load(log_dir + "best_val_checkpoint.tar", map_location='cpu')
    # Load network state
    net.load_state_dict(checkpoint['model_state_dict'])
    config.batch_size = 1
    probs, edges_value = compute_prob(net, config, dtypeLong, dtypeFloat, instance_number)
    write_adjacency_matrix(probs, edges_value, filepath)


if __name__ == "__main__":
    """
    Args:
        sys.argv[1]: The path to the file where the adjacency matrix will be written.
        sys.argv[2]: The number of nodes in the TSP instance.
        sys.argv[3]: The number of the instance to be computed.
    """
    if len(sys.argv) != 4:
        print("\nPlease provide the path to the output file to write in, the number of nodes in the tsp and the "
              "instance number to analyze. The format is: "
              "<filepath> <number of nodes> <instance number>\n")
        sys.exit(1)

    if not isinstance(sys.argv[1], str) or not isinstance(sys.argv[2], str) or not isinstance(sys.argv[3], str):
        print("Error: The arguments must be strings.")
        sys.exit(1)

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

    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.

```
import torch
import torch.nn.functional as F
import torch.nn as nn
```

```

import numpy as np


class BatchNormNode(nn.Module):
    """Batch normalization for node features.
    """

    def __init__(self, hidden_dim):
        super(BatchNormNode, self).__init__()
        self.batch_norm = nn.BatchNorm1d(hidden_dim, track_running_stats=False)

    def 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)
        """
        x_trans = x.transpose(1, 2).contiguous()  # Reshape input: (batch_size, hidden_dim, num_nodes)
        x_trans_bn = self.batch_norm(x_trans)
        x_bn = x_trans_bn.transpose(1, 2).contiguous()  # Reshape to original shape
        return x_bn


class BatchNormEdge(nn.Module):
    """Batch normalization for edge features.
    """

    def __init__(self, hidden_dim):
        super(BatchNormEdge, self).__init__()
        self.batch_norm = nn.BatchNorm2d(hidden_dim, track_running_stats=False)

    def 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)
        """
        e_trans = e.transpose(1, 3).contiguous()  # Reshape input: (batch_size, num_nodes, num_nodes, hidden_dim)
        e_trans_bn = self.batch_norm(e_trans)
        e_bn = e_trans_bn.transpose(1, 3).contiguous()  # Reshape to original
        return e_bn


class NodeFeatures(nn.Module):
    """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] )
    """

    def __init__(self, hidden_dim, aggregation="mean"):
        super(NodeFeatures, self).__init__()
        self.aggregation = aggregation
        self.U = nn.Linear(hidden_dim, hidden_dim, True)
        self.V = nn.Linear(hidden_dim, hidden_dim, True)

    def 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)
        """
        Ux = self.U(x)  # B x V x H
        Vx = self.V(x)  # B x V x H
        Vx = Vx.unsqueeze(1)  # extend Vx from "B x V x H" to "B x 1 x V x H"
        gateVx = edge_gate * Vx  # B x V x V x H
        if self.aggregation=="mean":
            x_new = Ux + torch.sum(gateVx, dim=2) / (1e-20 + torch.sum(edge_gate, dim=2))  # B x V x H
        elif self.aggregation=="sum":
            x_new = Ux + torch.sum(gateVx, dim=2)  # B x V x H
        return x_new


class EdgeFeatures(nn.Module):
```

```
    """Convnet features for edges.

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

    def __init__(self, hidden_dim):
        super(EdgeFeatures, self).__init__()
        self.U = nn.Linear(hidden_dim, hidden_dim, True)
        self.V = nn.Linear(hidden_dim, hidden_dim, True)

    def 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)
        """
        Ue = self.U(e)
        Vx = self.V(x)
        Wx = Vx.unsqueeze(1)  # Extend Vx from "B x V x H" to "B x V x 1 x H"
        Vx = Vx.unsqueeze(2)  # extend Vx from "B x V x H" to "B x 1 x V x H"
        e_new = Ue + Vx + Wx
        return e_new


class ResidualGatedGCNLayer(nn.Module):
    """Convnet layer with gating and residual connection.
    """

    def __init__(self, hidden_dim, aggregation="sum"):
        super(ResidualGatedGCNLayer, self).__init__()
        self.node_feat = NodeFeatures(hidden_dim, aggregation)
        self.edge_feat = EdgeFeatures(hidden_dim)
        self.bn_node = BatchNormNode(hidden_dim)
        self.bn_edge = BatchNormEdge(hidden_dim)

    def 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)
        """
        e_in = e
        x_in = x
        # Edge convolution
        e_tmp = self.edge_feat(x_in, e_in)  # B x V x V x H
        # Compute edge gates
        edge_gate = F.sigmoid(e_tmp)
        # Node convolution
        x_tmp = self.node_feat(x_in, edge_gate)
        # Batch normalization
        e_tmp = self.bn_edge(e_tmp)
        x_tmp = self.bn_node(x_tmp)
        # ReLU Activation
        e = F.relu(e_tmp)
        x = F.relu(x_tmp)
        # Residual connection
        x_new = x_in + x
        e_new = e_in + e
        return x_new, e_new


class MLP(nn.Module):
    """Multi-layer Perceptron for output prediction.
    """

    def __init__(self, hidden_dim, output_dim, L=2):
        super(MLP, self).__init__()
        self.L = L
        U = []
        for layer in range(self.L - 1):
            U.append(nn.Linear(hidden_dim, hidden_dim, True))
        self.U = nn.ModuleList(U)
        self.V = nn.Linear(hidden_dim, output_dim, True)

    def forward(self, x):
        """
        Args:
            x: Input features (batch_size, hidden_dim)

        Returns:
```

```
            y: Output predictions (batch_size, output_dim)
        """
        Ux = x
        for U_i in self.U:
            Ux = U_i(Ux)  # B x H
            Ux = F.relu(Ux)  # B x H
        y = self.V(Ux)  # B x O
        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.

```
import torch
import torch.nn.functional as F
import torch.nn as nn

from models.gcn_layers import ResidualGatedGCNLayer, MLP
from utils.model_utils import *


class ResidualGatedGCNModel(nn.Module):
    """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
    """

    def __init__(self, config, dtypeFloat, dtypeLong):
        super(ResidualGatedGCNModel, self).__init__()
        self.dtypeFloat = dtypeFloat
        self.dtypeLong = dtypeLong
        # Define net parameters
        self.num_nodes = config.num_nodes
        self.node_dim = config.node_dim
        self.voc_nodes_in = config['voc_nodes_in']
        self.voc_nodes_out = config['num_nodes']  # config['voc_nodes_out']
        self.voc_edges_in = config['voc_edges_in']
        self.voc_edges_out = config['voc_edges_out']
        self.hidden_dim = config['hidden_dim']
        self.num_layers = config['num_layers']
        self.mlp_layers = config['mlp_layers']
        self.aggregation = config['aggregation']
        # Node and edge embedding layers/lookups
        self.nodes_coord_embedding = nn.Linear(self.node_dim, self.hidden_dim, bias=False)
        self.edges_values_embedding = nn.Linear(1, self.hidden_dim//2, bias=False)
        self.edges_embedding = nn.Embedding(self.voc_edges_in, self.hidden_dim//2)
        # Define GCN Layers
        gcn_layers = []
        for layer in range(self.num_layers):
            gcn_layers.append(ResidualGatedGCNLayer(self.hidden_dim, self.aggregation))
        self.gcn_layers = nn.ModuleList(gcn_layers)
        # Define MLP classifiers
        self.mlp_edges = MLP(self.hidden_dim, self.voc_edges_out, self.mlp_layers)
        # self.mlp_nodes = MLP(self.hidden_dim, self.voc_nodes_out, self.mlp_layers)

```

```
    def 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
        """
        # Node and edge embedding
        x = self.nodes_coord_embedding(x_nodes_coord)  # B x V x H
        e_vals = self.edges_values_embedding(x_edges_values.unsqueeze(3))  # B x V x V x H
        e_tags = self.edges_embedding(x_edges)  # B x V x V x H
        e = torch.cat((e_vals, e_tags), dim=3)
        # GCN layers
        for layer in range(self.num_layers):
            x, e = self.gcn_layers[layer](x, e)  # B x V x H, B x V x V x H
        # MLP classifier
        y_pred_edges = self.mlp_edges(e)  # B x V x V x voc_edges_out
        # y_pred_nodes = self.mlp_nodes(x)  # B x V x voc_nodes_out

        # Compute loss
        edge_cw = torch.Tensor(edge_cw).type(self.dtypeFloat)  # Convert to tensors
        loss = loss_edges(y_pred_edges, y_edges, edge_cw)

        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.

```python
import numpy as np
import torch


class Beamsearch(object):
    """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
    """

    def __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
        """
        # Beamsearch parameters
        self.batch_size = batch_size
        self.beam_size = beam_size
        self.num_nodes = num_nodes
        self.probs_type = probs_type
        # Set data types
        self.dtypeFloat = dtypeFloat
        self.dtypeLong = dtypeLong
        # Set beamsearch starting nodes
        self.start_nodes = torch.zeros(batch_size, beam_size).type(self.dtypeLong)
        if random_start == True:
            # Random starting nodes
            self.start_nodes = torch.randint(0, num_nodes, (batch_size, beam_size)).type(self.dtypeLong)
        # Mask for constructing valid hypothesis
        self.mask = torch.ones(batch_size, beam_size, num_nodes).type(self.dtypeFloat)
        self.update_mask(self.start_nodes)  # Mask the starting node of the beam search
        # Score for each translation on the beam
        self.scores = torch.zeros(batch_size, beam_size).type(self.dtypeFloat)
        self.all_scores = []
        # Backpointers at each time-step
        self.prev_Ks = []
        # Outputs at each time-step
        self.next_nodes = [self.start_nodes]

    def get_current_state(self):
        """Get the output of the beam at the current timestep.
        """
        current_state = (self.next_nodes[-1].unsqueeze(2)
                         .expand(self.batch_size, self.beam_size, self.num_nodes))
        return current_state
```

```
    def get_current_origin(self):
        """Get the backpointers for the current timestep.
        """
        return self.prev_Ks[-1]

    def 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)
        """
        # Compound the previous scores (summing logits == multiplying probabilities)
        if len(self.prev_Ks) > 0:
            if self.probs_type == 'raw':
                beam_lk = trans_probs * self.scores.unsqueeze(2).expand_as(trans_probs)
            elif self.probs_type == 'logits':
                beam_lk = trans_probs + self.scores.unsqueeze(2).expand_as(trans_probs)
        else:
            beam_lk = trans_probs
            # Only use the starting nodes from the beam
            if self.probs_type == 'raw':
                beam_lk[:, 1:] = torch.zeros(beam_lk[:, 1:].size()).type(self.dtypeFloat)
            elif self.probs_type == 'logits':
                beam_lk[:, 1:] = -1e20 * torch.ones(beam_lk[:, 1:].size()).type(self.dtypeFloat)
        # Multiply by mask
        beam_lk = beam_lk * self.mask
        beam_lk = beam_lk.view(self.batch_size, -1)  # (batch_size, beam_size * num_nodes)
        # Get top k scores and indexes (k = beam_size)
        bestScores, bestScoresId = beam_lk.topk(self.beam_size, 1, True, True)
        # Update scores
        self.scores = bestScores
        # Update backpointers
        prev_k = bestScoresId / self.num_nodes
        self.prev_Ks.append(prev_k)
        # Update outputs
        new_nodes = bestScoresId - prev_k * self.num_nodes
        self.next_nodes.append(new_nodes)
        # Re-index mask
        perm_mask = prev_k.unsqueeze(2).expand_as(self.mask)  # (batch_size, beam_size, num_nodes)
        self.mask = self.mask.gather(1, perm_mask)
        # Mask newly added nodes
        self.update_mask(new_nodes)

    def update_mask(self, new_nodes):
        """Sets new_nodes to zero in mask.
        """
        arr = (torch.arange(0, self.num_nodes).unsqueeze(0).unsqueeze(1)
               .expand_as(self.mask).type(self.dtypeLong))
        new_nodes = new_nodes.unsqueeze(2).expand_as(self.mask)
        update_mask = 1 - torch.eq(arr, new_nodes).type(self.dtypeFloat)
        self.mask = self.mask * update_mask
        if self.probs_type == 'logits':
            # Convert 0s in mask to inf
            self.mask[self.mask == 0] = 1e20

    def sort_best(self):
        """Sort the beam.
        """
        return torch.sort(self.scores, 0, True)

    def get_best(self):
        """Get the score and index of the best hypothesis in the beam.
        """
        scores, ids = self.sort_best()
        return scores[1], ids[1]

    def 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)
        """
        assert self.num_nodes == len(self.prev_Ks) + 1

        hyp = -1 * torch.ones(self.batch_size, self.num_nodes).type(self.dtypeLong)
        for j in range(len(self.prev_Ks) - 1, -2, -1):
            hyp[:, j + 1] = self.next_nodes[j + 1].gather(1, k).view(1, self.batch_size)
            k = self.prev_Ks[j].gather(1, k)
        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.

```
import time
import numpy as np
from scipy.spatial.distance import pdist, squareform
from sklearn.utils import shuffle


class DotDict(dict):
    """Wrapper around in-built dict class to access members through the dot operation.
    """

    def __init__(self, **kwds):
        self.update(kwds)
        self.__dict__ = self


class GoogleTSPReader(object):
    """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
    """

    def __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)
        """
        self.num_nodes = num_nodes
        self.num_neighbors = num_neighbors
        self.batch_size = batch_size
        self.filepath = filepath
        self.filedata = shuffle(open(filepath, "r").readlines())  # Always shuffle upon reading data
        self.max_iter = (len(self.filedata) // batch_size)

    def __iter__(self):
        for batch in range(self.max_iter):
            start_idx = batch * self.batch_size
            end_idx = (batch + 1) * self.batch_size
            yield self.process_batch(self.filedata[start_idx:end_idx])

    def process_batch(self, lines):
        """Helper function to convert raw lines into a mini-batch as a DotDict.
        """
        batch_edges = []
        batch_edges_values = []
        batch_edges_target = []  # Binary classification targets (0/1)
        batch_nodes = []
        batch_nodes_target = []  # Multi-class classification targets (`num_nodes` classes)
        batch_nodes_coord = []
        batch_tour_nodes = []
        batch_tour_len = []

        for line_num, line in enumerate(lines):
```

```
            line = line.split(" ")  # Split into list

            # Compute signal on nodes
            nodes = np.ones(self.num_nodes)  # All 1s for TSP...

            # Convert node coordinates to required format
            nodes_coord = []
            for idx in range(0, 2 * self.num_nodes, 2):
                nodes_coord.append([float(line[idx]), float(line[idx + 1])])

            # Compute distance matrix
            W_val = squareform(pdist(nodes_coord, metric='euclidean'))

            # Compute adjacency matrix
            if self.num_neighbors == -1:
                W = np.ones((self.num_nodes, self.num_nodes))  # Graph is fully connected
            else:
                W = np.zeros((self.num_nodes, self.num_nodes))
                # Determine k-nearest neighbors for each node
                knns = np.argpartition(W_val, kth=self.num_neighbors, axis=-1)[:,
      self.num_neighbors::-1]
                # Make connections
                for idx in range(self.num_nodes):
                    W[idx][knns[idx]] = 1
            np.fill_diagonal(W, 2)  # Special token for self-connections

            # Convert tour nodes to required format
            # Don't add final connection for tour/cycle
            tour_nodes = [int(node) - 1 for node in line[line.index('output') + 1:-1]][:-1]

            # Compute node and edge representation of tour + tour_len
            tour_len = 0
            nodes_target = np.zeros(self.num_nodes)
            edges_target = np.zeros((self.num_nodes, self.num_nodes))
            for idx in range(len(tour_nodes) - 1):
                i = tour_nodes[idx]
                j = tour_nodes[idx + 1]
                nodes_target[i] = idx  # node targets: ordering of nodes in tour
                edges_target[i][j] = 1
                edges_target[j][i] = 1
                tour_len += W_val[i][j]

            # Add final connection of tour in edge target
            nodes_target[j] = len(tour_nodes) - 1
            edges_target[j][tour_nodes[0]] = 1
            edges_target[tour_nodes[0]][j] = 1
            tour_len += W_val[j][tour_nodes[0]]

            # Concatenate the data
            batch_edges.append(W)
            batch_edges_values.append(W_val)
            batch_edges_target.append(edges_target)
            batch_nodes.append(nodes)
            batch_nodes_target.append(nodes_target)
            batch_nodes_coord.append(nodes_coord)
            batch_tour_nodes.append(tour_nodes)
            batch_tour_len.append(tour_len)

        # From list to tensors as a DotDict
        batch = DotDict()
        batch.edges = np.stack(batch_edges, axis=0)
        batch.edges_values = np.stack(batch_edges_values, axis=0)
        batch.edges_target = np.stack(batch_edges_target, axis=0)
        batch.nodes = np.stack(batch_nodes, axis=0)
        batch.nodes_target = np.stack(batch_nodes_target, axis=0)
        batch.nodes_coord = np.stack(batch_nodes_coord, axis=0)
        batch.tour_nodes = np.stack(batch_tour_nodes, axis=0)
        batch.tour_len = np.stack(batch_tour_len, axis=0)
        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.

```
import torch
import torch.nn.functional as F

import numpy as np


def tour_nodes_to_W(nodes):
    """Helper function to convert ordered list of tour nodes to edge adjacency matrix.
    """
    W = np.zeros((len(nodes), len(nodes)))
    for idx in range(len(nodes) - 1):
        i = int(nodes[idx])
        j = int(nodes[idx + 1])
        W[i][j] = 1
        W[j][i] = 1
    # Add final connection of tour in edge target
    W[j][int(nodes[0])] = 1
    W[int(nodes[0])][j] = 1
    return W


def tour_nodes_to_tour_len(nodes, W_values):
    """Helper function to calculate tour length from ordered list of tour nodes.
    """
    tour_len = 0
    for idx in range(len(nodes) - 1):
        i = nodes[idx]
        j = nodes[idx + 1]
        tour_len += W_values[i][j]
    # Add final connection of tour in edge target
    tour_len += W_values[j][nodes[0]]
    return tour_len


def W_to_tour_len(W, W_values):
    """Helper function to calculate tour length from edge adjacency matrix.
    """
    tour_len = 0
    for i in range(W.shape[0]):
        for j in range(W.shape[1]):
            if W[i][j] == 1:
                tour_len += W_values[i][j]
    tour_len /= 2  # Divide by 2 because adjacency matrices are symmetric
    return tour_len


def is_valid_tour(nodes, num_nodes):
    """Sanity check: tour visits all nodes given.
    """
    return sorted(nodes) == [i for i in range(num_nodes)]


def 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
    """
```

```
    y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
    y = y.argmax(dim=3)  # B x V x V
    # Divide by 2 because edges_values is symmetric
    tour_lens = (y.float() * x_edges_values.float()).sum(dim=1).sum(dim=1) / 2
    mean_tour_len = tour_lens.sum().to(dtype=torch.float).item() / tour_lens.numel()
    return mean_tour_len


def 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
    """
    y = bs_nodes.cpu().numpy()
    W_val = x_edges_values.cpu().numpy()
    running_tour_len = 0
    for batch_idx in range(y.shape[0]):
        for y_idx in range(y[batch_idx].shape[0] - 1):
            i = y[batch_idx][y_idx]
            j = y[batch_idx][y_idx + 1]
            running_tour_len += W_val[batch_idx][i][j]
        running_tour_len += W_val[batch_idx][j][0]  # Add final connection to tour/cycle
    return running_tour_len / y.shape[0]


def 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.
    """
    ks = []
    for _ in range(max_iter):
        batch = next(iter(dataset))
        for idx in range(batch.edges.shape[0]):
            for row in range(dataset.num_nodes):
                # Compute indices of current node's neighbors in the TSP solution
                connections = np.where(batch.edges_target[idx][row]==1)[0]
                # Compute sorted list of indices of nearest neighbors (ascending order)
                sorted_neighbors = np.argsort(batch.edges_values[idx][row], axis=-1)
                for conn_idx in connections:
                    ks.append(np.where(sorted_neighbors==conn_idx)[0][0])
    # print("Ks array counts: ", np.unique(ks, return_counts=True))
    # print(f"Mean: {np.mean(ks)}, StdDev: {np.std(ks)}")
    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, dtypeFloat, 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.

```
import torch
import torch.nn.functional as F
import torch.nn as nn

from utils.beamsearch import *
from utils.graph_utils import *


def 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

    """
    # Node loss
    y = F.log_softmax(y_pred_nodes, dim=2)  # B x V x voc_nodes_out
    y = y.permute(0, 2, 1)  # B x voc_nodes x V
    loss_nodes = nn.NLLLoss(node_cw)(y, y_nodes)
    return loss_nodes


def 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

    """
    # Edge loss
    y = F.log_softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
    y = y.permute(0, 3, 1, 2)  # B x voc_edges x V x V
    loss_edges = nn.NLLLoss(edge_cw)(y, y_edges)
    return loss_edges


def 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)

    """
    if probs_type == 'raw':
        # Compute softmax over edge prediction matrix
        y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
        # Consider the second dimension only
        y = y[:, :, :, 1]  # B x V x V
    elif probs_type == 'logits':
        # Compute logits over edge prediction matrix
        y = F.log_softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
        # Consider the second dimension only
        y = y[:, :, :, 1]  # B x V x V
        y[y == 0] = -1e-20  # Set 0s (i.e. log(1)s) to very small negative number
    # Perform beamsearch
    beamsearch = Beamsearch(beam_size, batch_size, num_nodes, dtypeFloat, dtypeLong, probs_type,
random_start)
    trans_probs = y.gather(1, beamsearch.get_current_state())
    for step in range(num_nodes - 1):
        beamsearch.advance(trans_probs)
```

```
        trans_probs = y.gather(1, beamsearch.get_current_state())
    # Find TSP tour with highest probability among beam_size candidates
    ends = torch.zeros(batch_size, 1).type(dtypeLong)
    return beamsearch.get_hypothesis(ends)


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):
    """
    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)

    """
    if probs_type == 'raw':
        # Compute softmax over edge prediction matrix
        y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
        # Consider the second dimension only
        y = y[:, :, :, 1]  # B x V x V
    elif probs_type == 'logits':
        # Compute logits over edge prediction matrix
        y = F.log_softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
        # Consider the second dimension only
        y = y[:, :, :, 1]  # B x V x V
        y[y == 0] = -1e-20  # Set 0s (i.e. log(1)s) to very small negative number
    # Perform beamsearch
    beamsearch = Beamsearch(beam_size, batch_size, num_nodes, dtypeFloat, dtypeLong, probs_type, random_start)
    trans_probs = y.gather(1, beamsearch.get_current_state())
    for step in range(num_nodes - 1):
        beamsearch.advance(trans_probs)
        trans_probs = y.gather(1, beamsearch.get_current_state())
    # Initially assign shortest_tours as most probable tours i.e. standard beamsearch
    ends = torch.zeros(batch_size, 1).type(dtypeLong)
    shortest_tours = beamsearch.get_hypothesis(ends)
    # Compute current tour lengths
    shortest_lens = [1e6] * len(shortest_tours)
    for idx in range(len(shortest_tours)):
        shortest_lens[idx] = tour_nodes_to_tour_len(shortest_tours[idx].cpu().numpy(),
                                                    x_edges_values[idx].cpu().numpy())
    # Iterate over all positions in beam (except position 0 --> highest probability)
    for pos in range(1, beam_size):
        ends = pos * torch.ones(batch_size, 1).type(dtypeLong)  # New positions
        hyp_tours = beamsearch.get_hypothesis(ends)
        for idx in range(len(hyp_tours)):
            hyp_nodes = hyp_tours[idx].cpu().numpy()
            hyp_len = tour_nodes_to_tour_len(hyp_nodes, x_edges_values[idx].cpu().numpy())
            # Replace tour in shortest_tours if new length is shorter than current best
            if hyp_len < shortest_lens[idx] and is_valid_tour(hyp_nodes, num_nodes):
                shortest_tours[idx] = hyp_tours[idx]
                shortest_lens[idx] = hyp_len
    return shortest_tours


def 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
    """
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
    return optimizer

```

```

def 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

    """
    y = F.softmax(y_pred, dim=3)  # B x V x V x voc_edges
    y = y.argmax(dim=3)  # B x V x V

    # Edge error: Mask out edges which are not connected
    mask_no_edges = x_edges.long()
    err_edges, _ = _edge_error(y, y_target, mask_no_edges)

    # TSP tour edges error: Mask out edges which are not on true TSP tours
    mask_no_tour = y_target
    err_tour, err_idx_tour = _edge_error(y, y_target, mask_no_tour)

    # TSP tour edges + positively predicted edges error:
    # Mask out edges which are not on true TSP tours or are not predicted positively by model
    mask_no_tsp = ((y_target + y) > 0).long()
    err_tsp, err_idx_tsp = _edge_error(y, y_target, mask_no_tsp)

    return 100 * err_edges, 100 * err_tour, 100 * err_tsp, err_idx_tour, err_idx_tsp


def _edge_error(y, y_target, mask):
    """
    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

    """
    # Compute equalities between pred and target
    acc = (y == y_target).long()
    # Multipy by mask => set equality to 0 on disconnected edges
    acc = (acc * mask)
    #  Get accuracy of each y in the batch (sum of 1s in acc_edges divided by sum of 1s in edges mask)
    acc = acc.sum(dim=1).sum(dim=1).to(dtype=torch.float) /
mask.sum(dim=1).sum(dim=1).to(dtype=torch.float)
    # Compute indices which are not perfect
    err_idx = (acc < 1.0)
    # Take mean over batch
    acc = acc.sum().to(dtype=torch.float).item() / acc.numel()
    # Compute error
    err = 1.0 - acc
    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.

```
import torch
import torch.nn.functional as F

import matplotlib
import matplotlib.pyplot as plt
import networkx as nx

from utils.graph_utils import *


def 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

    """

    def _edges_to_node_pairs(W):
        """Helper function to convert edge matrix into pairs of adjacent nodes.
        """
        pairs = []
        for r in range(len(W)):
            for c in range(len(W)):
                if W[r][c] == 1:
                    pairs.append((r, c))
        return pairs

    G = nx.from_numpy_matrix(W_val)
    pos = dict(zip(range(len(x_coord)), x_coord.tolist()))
    adj_pairs = _edges_to_node_pairs(W)
    target_pairs = _edges_to_node_pairs(W_target)
    colors = ['g'] + ['b'] * (len(x_coord) - 1)  # Green for 0th node, blue for others
    nx.draw_networkx_nodes(G, pos, node_color=colors, node_size=50)
    nx.draw_networkx_edges(G, pos, edgelist=adj_pairs, alpha=0.3, width=0.5)
    nx.draw_networkx_edges(G, pos, edgelist=target_pairs, alpha=1, width=1, edge_color='r')
    p.set_title(title)
    return p


def 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

    """

    def _edges_to_node_pairs(W):
        """Helper function to convert edge matrix into pairs of adjacent nodes.
        """
        pairs = []
        edge_preds = []
```

```
        for r in range(len(W)):
            for c in range(len(W)):
                if W[r][c] > 0.25:
                    pairs.append((r, c))
                    edge_preds.append(W[r][c])
        return pairs, edge_preds

    G = nx.from_numpy_matrix(W_val)
    pos = dict(zip(range(len(x_coord)), x_coord.tolist()))
    node_pairs, edge_color = _edges_to_node_pairs(W_pred)
    node_color = ['g'] + ['b'] * (len(x_coord) - 1)  # Green for 0th node, blue for others
    nx.draw_networkx_nodes(G, pos, node_color=node_color, node_size=50)
    nx.draw_networkx_edges(G, pos, edgelist=node_pairs, edge_color=edge_color, edge_cmap=plt.cm.Reds, width=0.75)
    p.set_title(title)
    return p


def 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

    """
    y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
    y_bins = y.argmax(dim=3)  # Binary predictions: B x V x V
    y_probs = y[:,:,:,1]  # Prediction probabilities: B x V x V
    for f_idx, idx in enumerate(np.random.choice(len(y), num_plots, replace=False)):
        f = plt.figure(f_idx, figsize=(10, 5))
        x_coord = x_nodes_coord[idx].cpu().numpy()
        W = x_edges[idx].cpu().numpy()
        W_val = x_edges_values[idx].cpu().numpy()
        W_target = y_edges[idx].cpu().numpy()
        W_sol_bins = y_bins[idx].cpu().numpy()
        W_sol_probs = y_probs[idx].cpu().numpy()
        plt1 = f.add_subplot(121)
        plot_tsp(plt1, x_coord, W, W_val, W_target, 'Groundtruth: {:.3f}'.format(W_to_tour_len(W_target, W_val)))
        plt2 = f.add_subplot(122)
        plot_tsp_heatmap(plt2, x_coord, W_val, W_sol_probs, 'Prediction Heatmap')
        plt.show()


def 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

    """
    y = F.softmax(y_pred_edges, dim=3)  # B x V x V x voc_edges
    y_bins = y.argmax(dim=3)  # Binary predictions: B x V x V
    y_probs = y[:,:,:,1]  # Prediction probabilities: B x V x V
    #print(y_probs)
    for f_idx, idx in enumerate(np.random.choice(len(y), num_plots, replace=False)):
        f = plt.figure(f_idx, figsize=(15, 5))
        x_coord = x_nodes_coord[idx].cpu().numpy()
        W = x_edges[idx].cpu().numpy()
        W_val = x_edges_values[idx].cpu().numpy()
        W_target = y_edges[idx].cpu().numpy()
        W_sol_bins = y_bins[idx].cpu().numpy()
        W_sol_probs = y_probs[idx].cpu().numpy()
        W_bs = tour_nodes_to_W(bs_nodes[idx].cpu().numpy())
        plt1 = f.add_subplot(131)
        plot_tsp(plt1, x_coord, W, W_val, W_target, 'Groundtruth: {:.3f}'.format(W_to_tour_len(W_target, W_val)))
        plt2 = f.add_subplot(132)
        plot_tsp_heatmap(plt2, x_coord, W_val, W_sol_probs, 'Prediction Heatmap')
        plt3 = f.add_subplot(133)
        plot_tsp(plt3, x_coord, W, W_val, W_bs, 'Beamsearch: {:.3f}'.format(W_to_tour_len(W_bs, W_val)))
        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.

```
"""
    @file: HybridSolver.py
    @author Lorenzo Sciandra
    @brief First it builds the program in C, specifying the number of nodes to use and whether it is in hybrid mode or not.
    Then it runs the graph conv net on the instance, and finally it runs the Branch and Bound.
    It can be run on a single instance or a range of instances.
    The input matrix is generated by the neural network and stored in the data folder. The output is stored in the results folder.
    @version 0.1.0
    @date 2023-04-18
    @copyright Copyright (c) 2023, license MIT

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


import subprocess
import sys
import os
import time


def 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 built.
        num_nodes: The number of nodes to use in the C program.
        hyb_mode: 1 if the program is in hybrid mode, 0 otherwise.
    """
    source_directory = "../"
    cmake_command = [
        "cmake",
        "-S" + source_directory,
        "-B" + build_directory,
        "-DCMAKE_BUILD_TYPE=Release",
        "-DMAX_VERTEX_NUM=" + str(num_nodes),
        "-DHYBRID=" + str(hyb_mode)
    ]
    print(cmake_command)
    make_command = [
        "make",
        "-C" + build_directory,
```

```
        "-j"
    ]
    try:
        subprocess.check_call(cmake_command)
        subprocess.check_call(make_command)
    except subprocess.CalledProcessError as e:
        print("Build failed:")
        print(e.output)
        raise Exception("Build failed")


def hybrid_solver(num_instances, num_nodes, hyb_mode):
    """
    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.
    """
    build_directory = "../cmake-build/CMakeFiles/BranchAndBound1Tree.dir"
    hybrid = 1 if hyb_mode else 0
    build_c_program(build_directory, num_nodes, hybrid)

    if "-" in num_instances:
        instances = num_instances.split("-")
        start_instance = 1 if int(instances[0]) == 0 else int(instances[0])
        end_instance = int(instances[1])
    else:
        start_instance = 1
        end_instance = int(num_instances)

    print("Starting instance: " + str(start_instance))
    print("Ending instance: " + str(end_instance))

    for i in range(start_instance, end_instance + 1):
        start_time = time.time()
        input_file = "../data/AdjacencyMatrix/tsp_" + str(num_nodes) + "_nodes/tsp_test_" + str(i) + ".csv"
        absolute_input_path = os.path.abspath(input_file)
        result_mode = "hybrid" if hyb_mode else "classic"
        output_file = "../results/AdjacencyMatrix/tsp_" + str(num_nodes) + "_nodes_" + result_mode \
                      + "/tsp_result_" + str(i) + ".txt"
        if hyb_mode:
            absolute_python_path = os.path.abspath("./graph-convnet-tsp/main.py")
            result = subprocess.run(['python3', absolute_python_path, absolute_input_path, str(num_nodes), str(i)], cwd="./graph-convnet-tsp",
                                    check=True)
            if result.returncode == 0:
                print('Neural Network completed successfully on instance ' + str(i) + ' / ' + str(end_instance))
            else:
                print('Neural Network failed on instance ' + str(i) + ' / ' + str(end_instance))
            os.rename(absolute_input_path.replace(".csv", "_temp.csv"), absolute_input_path)

        absolute_output_path = os.path.abspath(output_file)
        cmd = [build_directory + "/BranchAndBound1Tree", absolute_input_path, absolute_output_path]
        result = subprocess.run(cmd)
        if result.returncode == 0:
            print('Branch-and-Bound completed successfully on instance ' + str(i) + ' / ' + str(end_instance))
        else:
            print('Branch-and-Bound failed on instance ' + str(i) + ' / ' + str(end_instance))
        end_time = time.time()
        # append to the output file the time taken to solve the instance
        with open(output_file, "a") as f:
            f.write("Time taken: " + str(end_time - start_time) + "s\n")


if __name__ == "__main__":
    """
    Args:
        sys.argv[1]: The range of instances to run on the Solver.
        sys.argv[2]: The number of nodes to use in the C program.
        sys.argv[3]: "y" if the program is in hybrid mode, "n" otherwise.
    """

    if len(sys.argv) != 4:
        print("\nERROR: Please provide the number of instances to run on the Solver, the number of nodes to select the "
              "correct Neural Network and yes or no to run on hybrid mode or not.\nUsage: "
              "python3 HybridSolver.py <num instances> <num nodes> <y/n>  or\n"
              "python3 HybridSolver.py <num start instance>-<num end instance> <num nodes> <y/n>\n")
        sys.exit(1)

    if not isinstance(sys.argv[1], str) or not isinstance(sys.argv[2], str) or not isinstance(sys.argv[3], str):
        print("ERROR: The arguments must be strings.")
        sys.exit(1)
```

```

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

    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

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.

```
#ifndef BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
#define BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
#include <stdio.h>
#include <limits.h>
#include <float.h>
#include <string.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <errno.h>
#include <pthread.h>
#define INFINITE FLT_MAX
// #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.
// #define HYBRID 0 -- no longer in this file, but in the CMakeLists.txt to be able to change it from the command line.
#define MAX_EDGES_NUM (MAX_VERTEX_NUM * (MAX_VERTEX_NUM - 1) / 2)
#define INIT_UB (sqrt(MAX_VERTEX_NUM) * 1.27f)
#define TRACE() fprintf(stderr, "%s (%d): %s\n", __FILE__, __LINE__, __func__)
#define APPROXIMATION 0.00001f
#define EPSILON (INIT_UB / 1000)
#define EPSILON2 (0.33f * EPSILON)
#define BETTER_PROB 0.1f
#define TIME_LIMIT_SECONDS 600
#define NUM_HK_INITIAL_ITERATIONS (((((float) MAX_VERTEX_NUM * MAX_VERTEX_NUM)/50) + 0.5f) + MAX_VERTEX_NUM + 15)
#define NUM_HK_ITERATIONS (((float) MAX_VERTEX_NUM / 4) + 5)
#endif //BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
```

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

Copyright (c) 2023, license MIT

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             }
```

```
            nodes[i].positionInGraph = i;
            nodes[i].x = x;
            nodes[i].y = y;
            nodes[i].num_neighbours = 0;
            add_elem_list_bottom(nodes_list, &nodes[i]);
            i++;
        } else {
            end_of_file = true;
        }
    }
    free(line);
    if (fclose(fp) == EOF) {
        perror("Error while closing the file.\n");
        printf("\nFile: %s\n", filename);
        exit(1);
    }
    create_euclidean_graph(graph, nodes_list);
}


void read_tsp_csv_file(Graph *graph, char *filename) {
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        perror("Error while opening the file.\n");
        printf("\nFile: %s\n", filename);
        exit(1);
    }
    graph->cost = 0;
    graph->num_edges = 0;
    graph->num_nodes = 0;
    graph->kind = WEIGHTED_GRAPH;
    graph->orderedEdges = false;
    unsigned short i = 0;
    unsigned short z = 0;
    char *line = NULL;
    size_t len = 0;
    while (getline(&line, &len, fp) != -1) {
        graph->nodes[i].positionInGraph = i;
        graph->nodes[i].x = 0;
        graph->nodes[i].y = 0;
        graph->nodes[i].num_neighbours = 0;
        char *token = strtok(line, ";");
        unsigned short j = 0;
        while (token != NULL && strcmp(token, "\n") != 0) {
            if (j != i) {
                double weight = 0, prob = 0;

                int result = sscanf(token, "(%lf, %lf)", &weight, &prob);
                if (result != 2) {
                    perror("Error while reading the file.\n");
                    printf("\nFile: %s\n", filename);
                    exit(1);
                }

                if (weight > 0) {
                    if (j > i) {
                        graph->nodes[i].neighbours[graph->nodes[i].num_neighbours] = j;
                        graph->nodes[i].num_neighbours++;
                        graph->num_edges++;
                        graph->edges[z].src = i;
                        graph->edges[z].dest = j;
                        graph->edges[z].prob = HYBRID ? prob : 0;
                        graph->edges[z].symbol = z + 1;
                        graph->edges[z].positionInGraph = z;
                        graph->edges[z].weight = weight;
                        graph->cost += graph->edges[z].weight;
                        graph->nodes[j].positionInGraph = j;
                        graph->edges_matrix[i][j] = graph->edges[z];
                        graph->edges_matrix[j][i] = graph->edges[z];
                        z++;
                    } else {
                        graph->nodes[i].neighbours[graph->nodes[i].num_neighbours] = j;
                        graph->nodes[i].num_neighbours++;
                    }
                }
            }
            token = strtok(NULL, ";");
            j++;
        }
        graph->num_nodes++;
        i++;
    }
    free(line);
    if (fclose(fp) == EOF) {
        perror("Error while closing the file.\n");
        printf("\nFile: %s\n", filename);
        exit(1);
```

```
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

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.

```
#ifndef BRANCHANDBOUND1TREE_TSP_INSTANCE_READER_H
#define BRANCHANDBOUND1TREE_TSP_INSTANCE_READER_H
#include "data_structures/graph.h"
void read_tsp_lib_file(Graph * graph, char * filename);
void read_tsp_csv_file(Graph * graph, char * filename);
#endif //BRANCHANDBOUND1TREE_TSP_INSTANCE_READER_H
```

Index