

# Graph Convolutional Branch and Bound

v1.0.0

Generated by Doxygen 1.9.6



<b>1 Graph Convolutional Branch and Bound TSP Solver</b>	<b>1</b>
1.1 Ideas	1
1.2 1-Tree Branch and Bound	1
1.3 Graph Convolutional Network	1
1.4 Neural Grafting	2
1.5 Code Documentation	2
1.6 Results	2
<b>2 An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem</b>	<b>3</b>
2.1 Overview	3
2.2 Pre-requisite Downloads	3
2.2.0.1 TSP Datasets	3
2.2.0.2 Pre-trained Models	4
2.3 Usage	4
2.3.0.1 Installation	4
2.3.0.2 Running in Notebook/Visualization Mode	4
2.3.0.3 Running in Script Mode	4
2.3.0.4 Splitting datasets into Training and Validation sets	4
2.3.0.5 Generating new data	5
2.4 Resources	5
<b>3 Main</b>	<b>7</b>
<b>4 Namespace Index</b>	<b>9</b>
4.1 Namespace List	9
<b>5 Class Index</b>	<b>11</b>
5.1 Class List	11
<b>6 File Index</b>	<b>13</b>
6.1 File List	13
<b>7 Namespace Documentation</b>	<b>15</b>
7.1 HybridSolver Namespace Reference	15
7.1.1 Detailed Description	15
7.1.2 Function Documentation	16
7.1.2.1 adjacency_matrix()	16
7.1.2.2 build_c_program()	16
7.1.2.3 create_temp_file()	16
7.1.2.4 get_instance()	17
7.1.2.5 get_nodes()	17
7.1.2.6 hybrid_solver()	17
7.1.3 Variable Documentation	18
7.1.3.1 action	18

7.1.3.2 default . . . . .	18
7.1.3.3 gen_matrix . . . . .	18
7.1.3.4 int . . . . .	18
7.1.3.5 opts . . . . .	18
7.1.3.6 parser . . . . .	19
7.1.3.7 str . . . . .	19
7.1.3.8 type . . . . .	19
7.2 main Namespace Reference . . . . .	19
7.2.1 Detailed Description . . . . .	19
7.2.2 Function Documentation . . . . .	20
7.2.2.1 add_dummy_cities() . . . . .	20
7.2.2.2 cluster_nodes() . . . . .	20
7.2.2.3 compute_prob() . . . . .	20
7.2.2.4 create_temp_file() . . . . .	21
7.2.2.5 fix_instance_size() . . . . .	21
7.2.2.6 get_instance() . . . . .	21
7.2.2.7 main() . . . . .	22
7.2.2.8 write_adjacency_matrix() . . . . .	22
7.2.3 Variable Documentation . . . . .	22
7.2.3.1 category . . . . .	23
7.2.3.2 filepath . . . . .	23
7.2.3.3 model_size . . . . .	23
7.2.3.4 num_nodes . . . . .	23
<b>8 Class Documentation</b> . . . . .	<b>25</b>
8.1 ConstrainedEdge Struct Reference . . . . .	25
8.1.1 Detailed Description . . . . .	25
8.1.2 Member Data Documentation . . . . .	25
8.1.2.1 dest . . . . .	25
8.1.2.2 src . . . . .	26
8.2 DIIElem Struct Reference . . . . .	26
8.2.1 Detailed Description . . . . .	26
8.2.2 Member Data Documentation . . . . .	26
8.2.2.1 next . . . . .	26
8.2.2.2 prev . . . . .	27
8.2.2.3 value . . . . .	27
8.3 Edge Struct Reference . . . . .	27
8.3.1 Detailed Description . . . . .	27
8.3.2 Member Data Documentation . . . . .	27
8.3.2.1 dest . . . . .	28
8.3.2.2 positionInGraph . . . . .	28
8.3.2.3 prob . . . . .	28

8.3.2.4 src	28
8.3.2.5 symbol	28
8.3.2.6 weight	29
8.4 FibonacciHeap Struct Reference	29
8.4.1 Detailed Description	29
8.4.2 Member Data Documentation	29
8.4.2.1 head_tree_list	29
8.4.2.2 min_root	30
8.4.2.3 num_nodes	30
8.4.2.4 num_trees	30
8.4.2.5 tail_tree_list	30
8.5 Forest Struct Reference	30
8.5.1 Detailed Description	31
8.5.2 Member Data Documentation	31
8.5.2.1 num_sets	31
8.5.2.2 sets	31
8.6 Graph Struct Reference	31
8.6.1 Detailed Description	32
8.6.2 Member Data Documentation	32
8.6.2.1 cost	32
8.6.2.2 edges	32
8.6.2.3 edges_matrix	32
8.6.2.4 kind	32
8.6.2.5 nodes	33
8.6.2.6 num_edges	33
8.6.2.7 num_nodes	33
8.6.2.8 orderedEdges	33
8.7 List Struct Reference	33
8.7.1 Detailed Description	34
8.7.2 Member Data Documentation	34
8.7.2.1 head	34
8.7.2.2 size	34
8.7.2.3 tail	34
8.8 ListIterator Struct Reference	35
8.8.1 Detailed Description	35
8.8.2 Member Data Documentation	35
8.8.2.1 curr	35
8.8.2.2 index	35
8.8.2.3 list	36
8.9 MST Struct Reference	36
8.9.1 Detailed Description	36
8.9.2 Member Data Documentation	36

8.9.2.1 cost	37
8.9.2.2 edges	37
8.9.2.3 edges_matrix	37
8.9.2.4 isValid	37
8.9.2.5 nodes	37
8.9.2.6 num_edges	38
8.9.2.7 num_nodes	38
8.9.2.8 prob	38
8.10 Node Struct Reference	38
8.10.1 Detailed Description	39
8.10.2 Member Data Documentation	39
8.10.2.1 neighbours	39
8.10.2.2 num_neighbours	39
8.10.2.3 positionInGraph	39
8.10.2.4 x	39
8.10.2.5 y	40
8.11 OrdTreeNode Struct Reference	40
8.11.1 Detailed Description	40
8.11.2 Member Data Documentation	40
8.11.2.1 head_child_list	41
8.11.2.2 is_root	41
8.11.2.3 key	41
8.11.2.4 left_sibling	41
8.11.2.5 marked	41
8.11.2.6 num_children	42
8.11.2.7 parent	42
8.11.2.8 right_sibling	42
8.11.2.9 tail_child_list	42
8.11.2.10 value	42
8.12 Problem Struct Reference	43
8.12.1 Detailed Description	43
8.12.2 Member Data Documentation	43
8.12.2.1 bestSolution	44
8.12.2.2 bestValue	44
8.12.2.3 candidateNodeId	44
8.12.2.4 end	44
8.12.2.5 exploredBBNodes	44
8.12.2.6 generatedBBNodes	45
8.12.2.7 graph	45
8.12.2.8 interrupted	45
8.12.2.9 num_fixed_edges	45
8.12.2.10 reformulationGraph	45

8.12.2.11 start	46
8.12.2.12 totTreeLevels	46
8.13 Set Struct Reference	46
8.13.1 Detailed Description	46
8.13.2 Member Data Documentation	46
8.13.2.1 curr	47
8.13.2.2 num_in_forest	47
8.13.2.3 parentSet	47
8.13.2.4 rango	47
8.14 SubProblem Struct Reference	47
8.14.1 Detailed Description	48
8.14.2 Member Data Documentation	48
8.14.2.1 constraints	48
8.14.2.2 cycleEdges	49
8.14.2.3 edge_to_branch	49
8.14.2.4 fatherId	49
8.14.2.5 id	49
8.14.2.6 mandatoryEdges	49
8.14.2.7 num_edges_in_cycle	50
8.14.2.8 num_forbidden_edges	50
8.14.2.9 num_mandatory_edges	50
8.14.2.10 oneTree	50
8.14.2.11 prob	50
8.14.2.12 timeToReach	51
8.14.2.13 treeLevel	51
8.14.2.14 type	51
8.14.2.15 value	51
8.15 SubProblemElem Struct Reference	51
8.15.1 Detailed Description	52
8.15.2 Member Data Documentation	52
8.15.2.1 next	52
8.15.2.2 prev	52
8.15.2.3 subProblem	52
8.16 SubProblemsList Struct Reference	53
8.16.1 Detailed Description	53
8.16.2 Member Data Documentation	53
8.16.2.1 head	53
8.16.2.2 size	53
8.16.2.3 tail	54
8.17 SubProblemsListIterator Struct Reference	54
8.17.1 Detailed Description	54
8.17.2 Member Data Documentation	54

8.17.2.1 curr	54
8.17.2.2 index	54
8.17.2.3 list	54
<b>9 File Documentation</b>	<b>55</b>
9.1 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/branch_and_bound.c File Reference	55
9.1.1 Detailed Description	56
9.1.2 Function Documentation	56
9.1.2.1 bound()	56
9.1.2.2 branch()	57
9.1.2.3 branch_and_bound()	57
9.1.2.4 check_feasibility()	57
9.1.2.5 check_hamiltonian()	58
9.1.2.6 compare_candidate_node()	58
9.1.2.7 compare_subproblems()	59
9.1.2.8 constrained_kruskal()	60
9.1.2.9 constrained_prim()	60
9.1.2.10 copy_constraints()	60
9.1.2.11 dfs()	61
9.1.2.12 find_candidate_node()	61
9.1.2.13 infer_constraints()	61
9.1.2.14 initialize_matrix()	62
9.1.2.15 max_edge_path_1Tree()	62
9.1.2.16 mst_to_one_tree()	62
9.1.2.17 nearest_prob_neighbour()	63
9.1.2.18 print_problem()	63
9.1.2.19 print_subProblem()	63
9.1.2.20 set_problem()	64
9.1.2.21 time_limit_reached()	64
9.1.2.22 variable_fixing()	64
9.2 branch_and_bound.c	65
9.3 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/branch_and_bound.h File Reference	79
9.3.1 Detailed Description	80
9.3.2 Function Documentation	80
9.3.2.1 bound()	80
9.3.2.2 branch()	81
9.3.2.3 branch_and_bound()	81
9.3.2.4 check_feasibility()	81
9.3.2.5 check_hamiltonian()	82
9.3.2.6 clean_matrix()	82
9.3.2.7 compare_subproblems()	83



9.3.2.8 copy_constraints()	83
9.3.2.9 dfs()	83
9.3.2.10 find_candidate_node()	84
9.3.2.11 infer_constraints()	84
9.3.2.12 mst_to_one_tree()	85
9.3.2.13 nearest_prob_neighbour()	85
9.3.2.14 print_problem()	85
9.3.2.15 print_subProblem()	86
9.3.2.16 set_problem()	86
9.3.2.17 time_limit_reached()	86
9.3.2.18 variable_fixing()	87
9.3.3 Variable Documentation	87
9.3.3.1 problem	87
9.4 branch_and_bound.h	88
9.5 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/kruskal.c File Reference	89
9.5.1 Detailed Description	89
9.5.2 Function Documentation	89
9.5.2.1 kruskal()	89
9.5.2.2 pivot_quicksort()	90
9.5.2.3 quick_sort()	90
9.5.2.4 swap()	90
9.5.2.5 wrap_quick_sort()	90
9.6 kruskal.c	91
9.7 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/kruskal.h File Reference	93
9.7.1 Detailed Description	93
9.7.2 Function Documentation	93
9.7.2.1 kruskal()	93
9.7.2.2 pivot_quicksort()	94
9.7.2.3 quick_sort()	94
9.7.2.4 swap()	95
9.7.2.5 wrap_quick_sort()	95
9.8 kruskal.h	95
9.9 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/prim.c File Reference	96
9.9.1 Detailed Description	96
9.9.2 Function Documentation	97
9.9.2.1 prim()	97
9.10 prim.c	97
9.11 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/prim.h File Reference	98
9.11.1 Detailed Description	98
9.11.2 Function Documentation	99
9.11.2.1 prim()	99
9.12 prim.h	99

9.13	GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/b_and_b_data.c	File Reference	99
9.13.1	Detailed Description		100
9.13.2	Function Documentation		100
9.13.2.1	add_elem_SubProblemList_bottom()		100
9.13.2.2	add_elem_SubProblemList_index()		101
9.13.2.3	build_list_elem()		101
9.13.2.4	create_SubProblemList_iterator()		101
9.13.2.5	delete_SubProblemList()		102
9.13.2.6	delete_SubProblemList_elem_index()		102
9.13.2.7	delete_SubProblemList_iterator()		102
9.13.2.8	get_current_openSubProblemList_iterator_element()		103
9.13.2.9	get_SubProblemList_elem_index()		103
9.13.2.10	get_SubProblemList_size()		103
9.13.2.11	is_SubProblemList_empty()		104
9.13.2.12	is_SubProblemList_iterator_valid()		104
9.13.2.13	list_openSubProblemList_next()		105
9.13.2.14	new_SubProblemList()		105
9.13.2.15	SubProblemList_iterator_get_next()		105
9.14	b_and_b_data.c		105
9.15	GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/b_and_b_data.h	File Reference	108
9.15.1	Detailed Description		109
9.15.2	Typedef Documentation		110
9.15.2.1	BBNodeType		110
9.15.2.2	ConstraintType		110
9.15.2.3	Problem		110
9.15.2.4	SubProblem		110
9.15.2.5	SubProblemElem		110
9.15.2.6	SubProblemsList		110
9.15.3	Enumeration Type Documentation		110
9.15.3.1	BBNodeType		110
9.15.3.2	ConstraintType		111
9.15.4	Function Documentation		111
9.15.4.1	add_elem_SubProblemList_bottom()		111
9.15.4.2	add_elem_SubProblemList_index()		112
9.15.4.3	create_SubProblemList_iterator()		112
9.15.4.4	delete_SubProblemList()		112
9.15.4.5	delete_SubProblemList_elem_index()		113
9.15.4.6	delete_SubProblemList_iterator()		113
9.15.4.7	get_SubProblemList_elem_index()		113
9.15.4.8	get_SubProblemList_size()		114
9.15.4.9	is_SubProblemList_empty()		114

9.15.4.10	<a href="#">is_SubProblemList_iterator_valid()</a>	115
9.15.4.11	<a href="#">new_SubProblemList()</a>	115
9.15.4.12	<a href="#">SubProblemList_iterator_get_next()</a>	115
9.16	<a href="#">b_and_b_data.h</a>	116
9.17	<a href="#">GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_↔ list/linked_list.h</a> File Reference	117
9.17.1	Macro Definition Documentation	118
9.17.1.1	<a href="#">BRANCHANDBOUND1TREE_LINKED_LIST_H</a>	118
9.17.2	Typedef Documentation	118
9.17.2.1	<a href="#">DllElem</a>	118
9.18	<a href="#">linked_list.h</a>	118
9.19	<a href="#">GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_↔ list/list_functions.c</a> File Reference	119
9.19.1	Detailed Description	119
9.19.2	Function Documentation	120
9.19.2.1	<a href="#">add_elem_list_bottom()</a>	120
9.19.2.2	<a href="#">add_elem_list_index()</a>	120
9.19.2.3	<a href="#">build_dll_elem()</a>	120
9.19.2.4	<a href="#">del_list()</a>	121
9.19.2.5	<a href="#">delete_list_elem_bottom()</a>	121
9.19.2.6	<a href="#">delete_list_elem_index()</a>	121
9.19.2.7	<a href="#">get_list_elem_index()</a>	122
9.19.2.8	<a href="#">get_list_size()</a>	122
9.19.2.9	<a href="#">is_list_empty()</a>	122
9.19.2.10	<a href="#">new_list()</a>	123
9.20	<a href="#">list_functions.c</a>	123
9.21	<a href="#">GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_↔ list/list_functions.h</a> File Reference	125
9.21.1	Detailed Description	126
9.21.2	Function Documentation	126
9.21.2.1	<a href="#">add_elem_list_bottom()</a>	126
9.21.2.2	<a href="#">add_elem_list_index()</a>	127
9.21.2.3	<a href="#">del_list()</a>	127
9.21.2.4	<a href="#">delete_list_elem_bottom()</a>	127
9.21.2.5	<a href="#">delete_list_elem_index()</a>	128
9.21.2.6	<a href="#">get_list_elem_index()</a>	128
9.21.2.7	<a href="#">get_list_size()</a>	129
9.21.2.8	<a href="#">is_list_empty()</a>	129
9.21.2.9	<a href="#">new_list()</a>	129
9.22	<a href="#">list_functions.h</a>	130
9.23	<a href="#">GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_↔ list/list_iterator.c</a> File Reference	130
9.23.1	Detailed Description	131

9.23.2 Function Documentation	131
9.23.2.1 create_list_iterator()	131
9.23.2.2 delete_list_iterator()	132
9.23.2.3 get_current_list_iterator_element()	132
9.23.2.4 is_list_iterator_valid()	132
9.23.2.5 list_iterator_get_next()	133
9.23.2.6 list_iterator_next()	133
9.24 list_iterator.c	133
9.25 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_↔ list/list_iterator.h File Reference	134
9.25.1 Detailed Description	135
9.25.2 Function Documentation	135
9.25.2.1 create_list_iterator()	135
9.25.2.2 delete_list_iterator()	136
9.25.2.3 get_current_list_iterator_element()	136
9.25.2.4 is_list_iterator_valid()	136
9.25.2.5 list_iterator_get_next()	137
9.25.2.6 list_iterator_next()	137
9.26 list_iterator.h	137
9.27 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/fibonacci_heap.c File Reference	138
9.27.1 Detailed Description	138
9.27.2 Function Documentation	139
9.27.2.1 cascading_cut()	139
9.27.2.2 consolidate()	139
9.27.2.3 create_fibonacci_heap()	139
9.27.2.4 create_insert_node()	139
9.27.2.5 create_node()	140
9.27.2.6 cut()	140
9.27.2.7 decrease_value()	140
9.27.2.8 delete_node()	141
9.27.2.9 extract_min()	141
9.27.2.10 insert_node()	141
9.27.2.11 link_trees()	142
9.27.2.12 swap_roots()	142
9.28 fibonacci_heap.c	142
9.29 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/fibonacci_heap.h File Reference	146
9.29.1 Detailed Description	147
9.29.2 Macro Definition Documentation	147
9.29.2.1 BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H	147
9.29.3 Typedef Documentation	147
9.29.3.1 FibonacciHeap	148

9.29.3.2 OrdTreeNode	148
9.29.4 Function Documentation	148
9.29.4.1 create_fibonacci_heap()	148
9.29.4.2 create_insert_node()	148
9.29.4.3 create_node()	149
9.29.4.4 decrease_value()	149
9.29.4.5 extract_min()	149
9.29.4.6 insert_node()	150
9.30 fibonacci_heap.h	150
9.31 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/graph.c File Reference	151
9.31.1 Detailed Description	152
9.31.2 Function Documentation	152
9.31.2.1 create_euclidean_graph()	152
9.31.2.2 create_graph()	152
9.31.2.3 print_graph()	153
9.32 graph.c	153
9.33 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/graph.h File Reference	155
9.33.1 Detailed Description	156
9.33.2 Typedef Documentation	156
9.33.2.1 Edge	156
9.33.2.2 Graph	156
9.33.2.3 GraphKind	156
9.33.2.4 Node	157
9.33.3 Enumeration Type Documentation	157
9.33.3.1 GraphKind	157
9.33.4 Function Documentation	158
9.33.4.1 create_euclidean_graph()	158
9.33.4.2 create_graph()	158
9.33.4.3 print_graph()	159
9.34 graph.h	159
9.35 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mfset.c File Reference	160
9.35.1 Detailed Description	160
9.35.2 Function Documentation	160
9.35.2.1 create_forest()	160
9.35.2.2 create_forest_constrained()	161
9.35.2.3 find()	161
9.35.2.4 merge()	162
9.35.2.5 print_forest()	162
9.36 mfset.c	162
9.37 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mfset.h File Reference	163
9.37.1 Detailed Description	164
9.37.2 Typedef Documentation	165

9.37.2.1 Forest . . . . .	165
9.37.2.2 Set . . . . .	165
9.37.3 Function Documentation . . . . .	165
9.37.3.1 create_forest() . . . . .	165
9.37.3.2 create_forest_constrained() . . . . .	165
9.37.3.3 find() . . . . .	166
9.37.3.4 merge() . . . . .	166
9.37.3.5 print_forest() . . . . .	167
9.38 mfset.h . . . . .	167
9.39 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mst.c File Reference . . . . .	168
9.39.1 Detailed Description . . . . .	168
9.39.2 Function Documentation . . . . .	168
9.39.2.1 add_edge() . . . . .	168
9.39.2.2 create_mst() . . . . .	169
9.39.2.3 print_mst() . . . . .	169
9.39.2.4 print_mst_original_weight() . . . . .	169
9.40 mst.c . . . . .	170
9.41 GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mst.h File Reference . . . . .	171
9.41.1 Detailed Description . . . . .	172
9.41.2 Macro Definition Documentation . . . . .	172
9.41.2.1 BRANCHANDBOUND1TREE_MST_H . . . . .	172
9.41.3 Typedef Documentation . . . . .	173
9.41.3.1 ConstrainedEdge . . . . .	173
9.41.3.2 MST . . . . .	173
9.41.4 Function Documentation . . . . .	173
9.41.4.1 add_edge() . . . . .	173
9.41.4.2 create_mst() . . . . .	173
9.41.4.3 print_mst() . . . . .	174
9.41.4.4 print_mst_original_weight() . . . . .	174
9.42 mst.h . . . . .	174
9.43 GraphConvolutionalBranchandBound/src/HybridSolver/main/graph-convnet-tsp/main.py File Reference . . . . .	175
9.44 main.py . . . . .	176
9.45 GraphConvolutionalBranchandBound/README.md File Reference . . . . .	181
9.46 GraphConvolutionalBranchandBound/src/HybridSolver/main/graph-convnet-tsp/README.md File Reference . . . . .	181
9.47 GraphConvolutionalBranchandBound/src/HybridSolver/main/HybridSolver.py File Reference . . . . .	181
9.48 HybridSolver.py . . . . .	182
9.49 GraphConvolutionalBranchandBound/src/HybridSolver/main/main.c File Reference . . . . .	185
9.49.1 Detailed Description . . . . .	185
9.49.2 Function Documentation . . . . .	186
9.49.2.1 main() . . . . .	186
9.50 main.c . . . . .	186

9.51 GraphConvolutionalBranchandBound/src/HybridSolver/main/problem_settings.h File Reference . .	187
9.51.1 Detailed Description . . . . .	188
9.51.2 Macro Definition Documentation . . . . .	188
9.51.2.1 BETTER_PROB . . . . .	188
9.51.2.2 EPSILON . . . . .	189
9.51.2.3 EPSILON2 . . . . .	189
9.51.2.4 GHOSH_UB . . . . .	189
9.51.2.5 INFINITE . . . . .	189
9.51.2.6 MAX_EDGES_NUM . . . . .	190
9.51.2.7 NUM_HK_INITIAL_ITERATIONS . . . . .	190
9.51.2.8 NUM_HK_ITERATIONS . . . . .	190
9.51.2.9 PRIM . . . . .	190
9.51.2.10 PROB_BRANCH . . . . .	191
9.51.2.11 TIME_LIMIT_SECONDS . . . . .	191
9.51.2.12 TRACE . . . . .	191
9.52 problem_settings.h . . . . .	192
9.53 GraphConvolutionalBranchandBound/src/HybridSolver/main/ReadME.md File Reference . . . . .	192
9.54 GraphConvolutionalBranchandBound/src/HybridSolver/main/tsp_instance_reader.c File Reference .	192
9.54.1 Detailed Description . . . . .	193
9.54.2 Function Documentation . . . . .	193
9.54.2.1 read_tsp_csv_file() . . . . .	193
9.54.2.2 read_tsp_lib_file() . . . . .	194
9.55 tsp_instance_reader.c . . . . .	194
9.56 GraphConvolutionalBranchandBound/src/HybridSolver/main/tsp_instance_reader.h File Reference .	196
9.56.1 Detailed Description . . . . .	196
9.56.2 Function Documentation . . . . .	197
9.56.2.1 read_tsp_csv_file() . . . . .	197
9.56.2.2 read_tsp_lib_file() . . . . .	197
9.57 tsp_instance_reader.h . . . . .	197

<b>Index</b>	<b>199</b>
--------------	------------





# Chapter 1

## Graph Convolutional Branch and Bound TSP Solver

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

### 1.1 Ideas

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

### 1.2 1-Tree Branch and Bound

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

### 1.3 Graph Convolutional Network

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

## 1.4 Neural Grafting

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

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

## 1.5 Code Documentation

All code documentation was completed using [Doxygen](#), and is accessible in both [online](#) and [PDF](#) formats.

## 1.6 Results

Below are some of the results obtained:

## Chapter 2

# An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem

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

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

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

## 2.1 Overview

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

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

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

## 2.2 Pre-requisite Downloads

### 2.2.0.1 TSP Datasets

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

### 2.2.0.2 Pre-trained Models

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

## 2.3 Usage

### 2.3.0.1 Installation

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

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

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

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

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

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

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

### 2.3.0.2 Running in Notebook/Visualization Mode

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

```
jupyter lab
```

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

### 2.3.0.3 Running in Script Mode

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

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

### 2.3.0.4 Splitting datasets into Training and Validation sets

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

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

### 2.3.0.5 Generating new data

New TSP data can be generated using the [Concorde solver](#).

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

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

## 2.4 Resources

- [Optimal TSP Datasets generated with Concorde](#)
- [Paper on arXiv](#)
- [Follow-up workshop paper](#)



## Chapter 3

### Main

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





## Chapter 4

# Namespace Index

### 4.1 Namespace List

Here is a list of all namespaces with brief descriptions:

<a href="#">HybridSolver</a>	.....	<a href="#">15</a>
<a href="#">main</a>	.....	<a href="#">19</a>



## Chapter 5

# Class Index

### 5.1 Class List

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

<a href="#">ConstrainedEdge</a>	A reduced form of an <a href="#">Edge</a> in the <a href="#">Graph</a> , with only the source and destination Nodes . . . . .	25
<a href="#">DlElem</a>	The double linked <a href="#">List</a> element . . . . .	26
<a href="#">Edge</a>	Structure of an <a href="#">Edge</a> . . . . .	27
<a href="#">FibonacciHeap</a>	The Fibonacci Heap datastructure as collection of Heap-ordered Trees . . . . .	29
<a href="#">Forest</a>	A <a href="#">Forest</a> is a list of Sets . . . . .	30
<a href="#">Graph</a>	Structure of a <a href="#">Graph</a> . . . . .	31
<a href="#">List</a>	The double linked list . . . . .	33
<a href="#">ListIterator</a>	The iterator for the <a href="#">List</a> . . . . .	35
<a href="#">MST</a>	Minimum Spanning Tree, or <a href="#">MST</a> , and also a 1-Tree . . . . .	36
<a href="#">Node</a>	Structure of a <a href="#">Node</a> . . . . .	38
<a href="#">OrdTreeNode</a>	A Heap-ordered Tree <a href="#">Node</a> where the key of the parent is $\leq$ the key of its children . . . . .	40
<a href="#">Problem</a>	The struct used to represent the overall problem . . . . .	43
<a href="#">Set</a>	A <a href="#">Set</a> is a node in the <a href="#">Forest</a> . . . . .	46
<a href="#">SubProblem</a>	The struct used to represent a <a href="#">SubProblem</a> or node of the Branch and Bound tree . . . . .	47
<a href="#">SubProblemElem</a>	The element of the list of SubProblems . . . . .	51
<a href="#">SubProblemsList</a>	The list of open SubProblems . . . . .	53
<a href="#">SubProblemsListIterator</a>	The iterator of the list of SubProblems . . . . .	54



## Chapter 6

# File Index

### 6.1 File List

Here is a list of all files with brief descriptions:

GraphConvolutionalBranchandBound/src/HybridSolver/main/ <a href="#">HybridSolver.py</a> . . . . .	181
GraphConvolutionalBranchandBound/src/HybridSolver/main/ <a href="#">main.c</a> Project main file, where you start the program, read the input file and print/write the results . . .	185
GraphConvolutionalBranchandBound/src/HybridSolver/main/ <a href="#">problem_settings.h</a> Contains all the execution settings . . . . .	187
GraphConvolutionalBranchandBound/src/HybridSolver/main/ <a href="#">tsp_instance_reader.c</a> The definition of the function to read input files . . . . .	192
GraphConvolutionalBranchandBound/src/HybridSolver/main/ <a href="#">tsp_instance_reader.h</a> The declaration of the function to read input files . . . . .	196
GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/ <a href="#">branch_and_bound.c</a> The implementation of all the methods used by the Branch and Bound algorithm . . . . .	55
GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/ <a href="#">branch_and_bound.h</a> The declaration of all the methods used by the Branch and Bound algorithm . . . . .	79
GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/ <a href="#">kruskal.c</a> The implementaion of the functions needed to compute the <a href="#">MST</a> with Kruskal's algorithm . . . .	89
GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/ <a href="#">kruskal.h</a> The declaration of the functions needed to compute the <a href="#">MST</a> with Kruskal's algorithm . . . . .	93
GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/ <a href="#">prim.c</a> The implementaion of the functions needed to compute the <a href="#">MST</a> with Prim's algorithm . . . . .	96
GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/ <a href="#">prim.h</a> The declaration of the functions needed to compute the <a href="#">MST</a> with Prim's algorithm . . . . .	98
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/ <a href="#">b_and_b_data.c</a> All the functions needed to manage the list of open subproblems . . . . .	99
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/ <a href="#">b_and_b_data.h</a> The data structures used in the Branch and Bound algorithm . . . . .	108
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/ <a href="#">fibonacci_heap.c</a> This file contains the implementation of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem . . . . .	138
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/ <a href="#">fibonacci_heap.h</a> This file contains the declaration of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem . . . . .	146
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/ <a href="#">graph.c</a> The implementation of the graph data structure . . . . .	151
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/ <a href="#">graph.h</a> The data structures to model the <a href="#">Graph</a> . . . . .	155

GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mfset.c	
This file contains the implementation of the Merge-Find <a href="#">Set</a> datastructure for the Minimum Spanning Tree problem . . . . .	160
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mfset.h	
This file contains the declaration of the Merge-Find <a href="#">Set</a> datastructure for the Minimum Spanning Tree problem . . . . .	163
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mst.c	
This file contains the definition of the Minimum Spanning Tree operations . . . . .	168
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/mst.h	
This file contains the declaration of the Minimum Spanning Tree datastructure . . . . .	171
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_list/linked_list.h	
117	
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_list/list_functions.c	
The definition of the functions to manipulate the <a href="#">List</a> . . . . .	119
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_list/list_functions.h	
The declaration of the functions to manipulate the <a href="#">List</a> . . . . .	125
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_list/list_iterator.c	
The definition of the functions to manipulate the <a href="#">ListIterator</a> . . . . .	130
GraphConvolutionalBranchandBound/src/HybridSolver/main/data_structures/doubly_linked_list/list_iterator.h	
The declaration of the functions to manipulate the <a href="#">ListIterator</a> . . . . .	134
GraphConvolutionalBranchandBound/src/HybridSolver/main/graph-convnet-tsp/main.py	175

## Chapter 7

# Namespace Documentation

### 7.1 HybridSolver Namespace Reference

#### Functions

- def [adjacency\\_matrix](#) (orig\_graph)
- def [create\\_temp\\_file](#) (num\_nodes, str\_grap)
- def [get\\_nodes](#) (graph)
- def [get\\_instance](#) (instance, num\_nodes)
- def [build\\_c\\_program](#) (build\_directory, num\_nodes, hyb\_mode)
- def [hybrid\\_solver](#) (num\_instances, num\_nodes, hyb\_mode, [gen\\_matrix](#))

#### Variables

- argparse [parser](#) = argparse.ArgumentParser()
- [type](#)
- [str](#)
- [default](#)
- [int](#)
- [action](#)
- argparse [opts](#) = parser.parse\_args()
- argparse [gen\\_matrix](#) = False

#### 7.1.1 Detailed Description

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

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

## 7.1.2 Function Documentation

### 7.1.2.1 adjacency\_matrix()

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

Calculates the adjacency matrix of the graph.

Args:

orig\_graph: The original graph.

Returns:

The adjacency matrix of the graph.

Definition at line 22 of file [HybridSolver.py](#).

### 7.1.2.2 build\_c\_program()

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

Builds the C program with the specified number of nodes and whether it is in hybrid mode or not.

Args:

build\_directory: The directory where the CMakeLists.txt file is located and where the executable will be 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.

Definition at line 115 of file [HybridSolver.py](#).

### 7.1.2.3 create\_temp\_file()

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

Creates a temporary file to store the current instance of the TSP for the neural network.

Args:

num\_nodes: The number of nodes in the TSP instance.

str\_grap: The string representation of the graph.

Definition at line 41 of file [HybridSolver.py](#).



#### 7.1.2.4 `get_instance()`

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

Gets the instance of the TSP from the file.

Args:

instance: The instance to get.  
num\_nodes: The number of nodes in the TSP instance.

Returns:

The graph in a list and string format.

Definition at line 81 of file [HybridSolver.py](#).

#### 7.1.2.5 `get_nodes()`

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

From a graph, it returns the nodes in a string format.

Args:

graph: The graph to get the nodes from.

Returns:

The nodes in a string format.

Definition at line 63 of file [HybridSolver.py](#).

#### 7.1.2.6 `hybrid_solver()`

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

The Graph Convolutional Branch-and-Bound Solver.

Args:

num\_instances: The range of instances to run on the Solver.  
num\_nodes: The number of nodes in each TSP instance.  
hyb\_mode: True if the program is in hybrid mode, False otherwise.  
gen\_matrix: True if the adjacency matrix is already generated, False otherwise.

Definition at line 147 of file [HybridSolver.py](#).

### 7.1.3 Variable Documentation

#### 7.1.3.1 action

`HybridSolver.action`

Definition at line 263 of file [HybridSolver.py](#).

#### 7.1.3.2 default

`HybridSolver.default`

Definition at line 261 of file [HybridSolver.py](#).

#### 7.1.3.3 gen\_matrix

`argparse HybridSolver.gen_matrix = False`

Definition at line 268 of file [HybridSolver.py](#).

#### 7.1.3.4 int

`HybridSolver.int`

Definition at line 262 of file [HybridSolver.py](#).

#### 7.1.3.5 opts

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

Definition at line 264 of file [HybridSolver.py](#).

### 7.1.3.6 parser

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

Definition at line 260 of file [HybridSolver.py](#).

### 7.1.3.7 str

```
HybridSolver.str
```

Definition at line 261 of file [HybridSolver.py](#).

### 7.1.3.8 type

```
HybridSolver.type
```

Definition at line 261 of file [HybridSolver.py](#).

## 7.2 main Namespace Reference

### Functions

- def [compute\\_prob](#) (net, config, dtypeLong, dtypeFloat)
- def [write\\_adjacency\\_matrix](#) (graph, y\_probs, x\_edges\_values, nodes\_coord, [filepath](#), [num\\_nodes](#), kmedoids\_labels=None)
- def [add\\_dummy\\_cities](#) (num\_nodes, model\_size)
- def [create\\_temp\\_file](#) (num\_nodes, str\_grap)
- def [cluster\\_nodes](#) (graph, k)
- def [fix\\_instance\\_size](#) (graph, num\_nodes, model\_size=100)
- def [get\\_instance](#) (num\_nodes)
- def [main](#) (filepath, num\_nodes, model\_size)

### Variables

- [category](#)
- sys [filepath](#) = sys.argv[1]
- int [num\\_nodes](#) = int(sys.argv[2])
- int [model\\_size](#) = int(sys.argv[3])

### 7.2.1 Detailed Description

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

## 7.2.2 Function Documentation

### 7.2.2.1 add\_dummy\_cities()

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

This function adds dummy cities to the graph instance. The dummy cities are randomly generated and added to the graph instance and the new instance is saved in a temporary file.

Args:

num\_nodes: The number of nodes of the graph instance.  
model\_size: The size of the Graph Convolutional Network to use.

Definition at line 199 of file [main.py](#).

### 7.2.2.2 cluster\_nodes()

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

Applies the k-medoids clustering to the graph.

Args:

graph: The graph to cluster.  
k: The number of clusters to create.

Returns:

medoids\_str: The medoids of the clusters.  
kmedoids.labels\_: The labels of the clusters.

Definition at line 269 of file [main.py](#).

### 7.2.2.3 compute\_prob()

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

This function computes the probability of the edges being in the optimal tour, by running the GCN.

Args:

net: The Graph Convolutional Network.  
config: The configuration file, from which the parameters are taken.  
dtypeLong: The data type for the long tensors.  
dtypeFloat: The data type for the float tensors.

Returns:

y\_probs: The probability of the edges being in the optimal tour.  
x\_edges\_values: The distance between the nodes.

Definition at line 37 of file [main.py](#).

#### 7.2.2.4 create\_temp\_file()

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

Creates a temporary file with the graph instance.

Args:

num\_nodes: The number of nodes of the graph instance.  
str\_grap: The graph instance.

Definition at line 253 of file [main.py](#).

#### 7.2.2.5 fix\_instance\_size()

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

The function that fixes the instance size with clustering.

It applies the k-medoids clustering to the graph and creates a new instance with the medoids as the new nodes.

Args:

graph: The graph to fix.  
num\_nodes: The number of nodes of the graph instance.  
model\_size: The size of the Graph Convolutional Network to use.

Definition at line 287 of file [main.py](#).

#### 7.2.2.6 get\_instance()

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

The function that reads the current instance from the file.

Args:

num\_nodes: The number of nodes of the graph instance.

Returns:

graph: The graph instance.

Definition at line 312 of file [main.py](#).

### 7.2.2.7 main()

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

The function that runs the Graph Convolutional Network and writes the adjacency matrix to a file for the given input instance.

Args:

filepath: The path to the file where the adjacency matrix will be written.  
 num\_nodes: The number of nodes in the TSP instance.  
 model\_size: The size of the Graph Convolutional Network to use.

Definition at line 345 of file [main.py](#).

### 7.2.2.8 write\_adjacency\_matrix()

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

This function writes the adjacency matrix to a file.

The file is in the format:

```
cities: (x1, y1);(x2, y2);...;(xn, yn)
adjacency matrix:
(0.0, 0.0);(0.23, 0.9);...;(0.15, 0.56)
...
(0.23, 0.9);(0.3, 0.59);...;(0.0, 0.0)
```

where each entry is (distance, probability)

If needed adjusts the size of the graph when the model size is different from the number of nodes in the instance.

Args:

graph: The set of nodes in the graph.  
 y\_probs: The probability of the edges being in the optimal tour.  
 x\_edges\_values: The weight of the edges.  
 nodes\_coord: The nodes coordinates used in the GCN.  
 filepath: The path to the file where the adjacency matrix will be written.  
 num\_nodes: The number of nodes in the TSP instance.  
 kmedoids\_labels: The labels of the k-medoids clustering.

Definition at line 115 of file [main.py](#).

## 7.2.3 Variable Documentation

### 7.2.3.1 category

`main.category`

Definition at line 24 of file [main.py](#).

### 7.2.3.2 filepath

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

Definition at line 421 of file [main.py](#).

### 7.2.3.3 model\_size

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

Definition at line 423 of file [main.py](#).

### 7.2.3.4 num\_nodes

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

Definition at line 422 of file [main.py](#).





## Chapter 8

# Class Documentation

### 8.1 ConstrainedEdge Struct Reference

A reduced form of an [Edge](#) in the [Graph](#), with only the source and destination Nodes.

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

#### Public Attributes

- unsigned short [src](#)  
*The source [Node](#) of the [Edge](#).*
- unsigned short [dest](#)  
*The destination [Node](#) of the [Edge](#).*

#### 8.1.1 Detailed Description

A reduced form of an [Edge](#) in the [Graph](#), with only the source and destination Nodes.

Definition at line [21](#) of file [mst.h](#).

#### 8.1.2 Member Data Documentation

##### 8.1.2.1 [dest](#)

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

The destination [Node](#) of the [Edge](#).

Definition at line [23](#) of file [mst.h](#).

### 8.1.2.2 src

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

The source [Node](#) of the [Edge](#).

Definition at line 22 of file [mst.h](#).

## 8.2 DllElem Struct Reference

The double linked [List](#) element.

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

### Public Attributes

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

### 8.2.1 Detailed Description

The double linked [List](#) element.

Definition at line 27 of file [linked\\_list.h](#).

### 8.2.2 Member Data Documentation

#### 8.2.2.1 next

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

The next element in the [List](#).

Definition at line 29 of file [linked\\_list.h](#).

### 8.2.2.2 prev

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

The previous element in the [List](#).

Definition at line 30 of file [linked\\_list.h](#).

### 8.2.2.3 value

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

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

Definition at line 28 of file [linked\\_list.h](#).

## 8.3 Edge Struct Reference

Structure of an [Edge](#).

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

### Public Attributes

- unsigned short [src](#)  
*ID of the source vertex.*
- unsigned short [dest](#)  
*ID of the destination vertex.*
- unsigned short [symbol](#)  
*Symbol of the [Edge](#), i.e. its unique ID.*
- double [weight](#)  
*Weight of the [Edge](#), 1 if the `data_structures` is not weighted.*
- double [prob](#)  
*Probability of the [Edge](#) to be in an optimal tour.*
- unsigned short [positionInGraph](#)  
*Position of the [Edge](#) in the list of Edges of the [Graph](#).*

### 8.3.1 Detailed Description

Structure of an [Edge](#).

Definition at line 40 of file [graph.h](#).

### 8.3.2 Member Data Documentation

#### 8.3.2.1 dest

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

ID of the destination vertex.

Definition at line 42 of file [graph.h](#).

#### 8.3.2.2 positionInGraph

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

Position of the [Edge](#) in the list of Edges of the [Graph](#).

Definition at line 46 of file [graph.h](#).

#### 8.3.2.3 prob

```
double Edge::prob
```

Probability of the [Edge](#) to be in an optimal tour.

Definition at line 45 of file [graph.h](#).

#### 8.3.2.4 src

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

ID of the source vertex.

Definition at line 41 of file [graph.h](#).

#### 8.3.2.5 symbol

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

Symbol of the [Edge](#), i.e. its unique ID.

Definition at line 43 of file [graph.h](#).

### 8.3.2.6 weight

```
double Edge::weight
```

Weight of the [Edge](#), 1 if the data\_structures is not weighted.

Definition at line 44 of file [graph.h](#).

## 8.4 FibonacciHeap Struct Reference

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

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

### Public Attributes

- [OrdTreeNode](#) \* [min\\_root](#)  
*The root of the Heap-ordered Tree with the minimum value.*
- [OrdTreeNode](#) \* [head\\_tree\\_list](#)  
*The root of the head Tree in the Fibonacci Heap.*
- [OrdTreeNode](#) \* [tail\\_tree\\_list](#)  
*The root of the tail Tree in the Fibonacci Heap.*
- unsigned short [num\\_nodes](#)  
*The number of Nodes in the Heap.*
- unsigned short [num\\_trees](#)  
*The number of Trees in the Heap.*

### 8.4.1 Detailed Description

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

Definition at line 43 of file [fibonacci\\_heap.h](#).

### 8.4.2 Member Data Documentation

#### 8.4.2.1 head\_tree\_list

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

The root of the head Tree in the Fibonacci Heap.

Definition at line 45 of file [fibonacci\\_heap.h](#).

#### 8.4.2.2 min\_root

```
OrdTreeNode* FibonacciHeap::min_root
```

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

Definition at line 44 of file [fibonacci\\_heap.h](#).

#### 8.4.2.3 num\_nodes

```
unsigned short FibonacciHeap::num_nodes
```

The number of Nodes in the Heap.

Definition at line 47 of file [fibonacci\\_heap.h](#).

#### 8.4.2.4 num\_trees

```
unsigned short FibonacciHeap::num_trees
```

The number of Trees in the Heap.

Definition at line 48 of file [fibonacci\\_heap.h](#).

#### 8.4.2.5 tail\_tree\_list

```
OrdTreeNode* FibonacciHeap::tail_tree_list
```

The root of the tail Tree in the Fibonacci Heap.

Definition at line 46 of file [fibonacci\\_heap.h](#).

## 8.5 Forest Struct Reference

A [Forest](#) is a list of Sets.

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

### Public Attributes

- unsigned short [num\\_sets](#)  
*Number of Sets in the [Forest](#).*
- [Set sets](#) [MAX\_VERTEX\_NUM]  
*Array of Sets.*

### 8.5.1 Detailed Description

A [Forest](#) is a list of Sets.

Definition at line 28 of file [mfset.h](#).

### 8.5.2 Member Data Documentation

#### 8.5.2.1 num\_sets

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

Number of Sets in the [Forest](#).

Definition at line 29 of file [mfset.h](#).

#### 8.5.2.2 sets

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

Array of Sets.

Definition at line 30 of file [mfset.h](#).

## 8.6 Graph Struct Reference

Structure of a [Graph](#).

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

### Public Attributes

- [GraphKind](#) kind  
*Type of the [Graph](#).*
- double [cost](#)  
*Sum of the weights of the Edges in the [Graph](#).*
- unsigned short [num\\_nodes](#)  
*Number of Nodes in the [Graph](#).*
- unsigned short [num\\_edges](#)  
*Number of Edges in the [Graph](#).*
- bool [orderedEdges](#)  
*True if the Edges are ordered by weight, false otherwise.*
- [Node](#) nodes [MAX\_VERTEX\_NUM]  
*Array of Nodes.*
- [Edge](#) edges [MAX\_EDGES\_NUM]  
*Array of Edges.*
- [Edge](#) edges\_matrix [MAX\_VERTEX\_NUM][MAX\_VERTEX\_NUM]  
*Adjacency matrix of the [Graph](#).*

### 8.6.1 Detailed Description

Structure of a [Graph](#).

Definition at line 51 of file [graph.h](#).

### 8.6.2 Member Data Documentation

#### 8.6.2.1 cost

```
double Graph::cost
```

Sum of the weights of the Edges in the [Graph](#).

Definition at line 53 of file [graph.h](#).

#### 8.6.2.2 edges

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

Array of Edges.

Definition at line 58 of file [graph.h](#).

#### 8.6.2.3 edges\_matrix

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

Adjacency matrix of the [Graph](#).

Definition at line 59 of file [graph.h](#).

#### 8.6.2.4 kind

```
GraphKind Graph::kind
```

Type of the [Graph](#).

Definition at line 52 of file [graph.h](#).



#### 8.6.2.5 nodes

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

Array of Nodes.

Definition at line 57 of file [graph.h](#).

#### 8.6.2.6 num\_edges

`unsigned short` `Graph::num_edges`

Number of Edges in the [Graph](#).

Definition at line 55 of file [graph.h](#).

#### 8.6.2.7 num\_nodes

`unsigned short` `Graph::num_nodes`

Number of Nodes in the [Graph](#).

Definition at line 54 of file [graph.h](#).

#### 8.6.2.8 orderedEdges

`bool` `Graph::orderedEdges`

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

Definition at line 56 of file [graph.h](#).

## 8.7 List Struct Reference

The double linked list.

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

## Public Attributes

- [DllElem](#) \* [head](#)  
*The head of the list as a [DllElem](#).*
- [DllElem](#) \* [tail](#)  
*The tail of the list as a [DllElem](#).*
- `size_t` [size](#)  
*The current size of the [List](#).*

### 8.7.1 Detailed Description

The double linked list.

Definition at line 35 of file [linked\\_list.h](#).

### 8.7.2 Member Data Documentation

#### 8.7.2.1 head

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

The head of the list as a [DllElem](#).

Definition at line 36 of file [linked\\_list.h](#).

#### 8.7.2.2 size

```
size_t List::size
```

The current size of the [List](#).

Definition at line 38 of file [linked\\_list.h](#).

#### 8.7.2.3 tail

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

The tail of the list as a [DllElem](#).

Definition at line 37 of file [linked\\_list.h](#).

## 8.8 ListIterator Struct Reference

The iterator for the [List](#).

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

### Public Attributes

- [List](#) \* [list](#)  
*The [List](#) to iterate.*
- [DllElem](#) \* [curr](#)  
*The current [DllElem](#) (element) of the [List](#).*
- [size\\_t](#) [index](#)  
*The current index of the element in the [List](#).*

### 8.8.1 Detailed Description

The iterator for the [List](#).

Definition at line 43 of file [linked\\_list.h](#).

### 8.8.2 Member Data Documentation

#### 8.8.2.1 curr

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

The current [DllElem](#) (element) of the [List](#).

Definition at line 45 of file [linked\\_list.h](#).

#### 8.8.2.2 index

```
size\_t ListIterator::index
```

The current index of the element in the [List](#).

Definition at line 46 of file [linked\\_list.h](#).

### 8.8.2.3 list

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

The [List](#) to iterate.

Definition at line 44 of file [linked\\_list.h](#).

## 8.9 MST Struct Reference

Minimum Spanning Tree, or [MST](#), and also a 1-Tree.

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

### Public Attributes

- [bool isValid](#)  
*True if the [MST](#) has the correct number of Edges, false otherwise.*
- [double cost](#)  
*The total cost of the [MST](#), i.e. the sum of the weights of the Edges.*
- [double prob](#)  
*The probability of the [MST](#), i.e. the average of the probabilities of its Edges.*
- [unsigned short num\\_nodes](#)  
*The number of Nodes in the [MST](#).*
- [unsigned short num\\_edges](#)  
*The number of Edges in the [MST](#).*
- [Node nodes](#) [MAX\_VERTEX\_NUM]  
*The set of Nodes in the [MST](#).*
- [Edge edges](#) [MAX\_VERTEX\_NUM]  
*The set of Edges in the [MST](#), these are  $|V|$  because the [MST](#) can be a 1-Tree.*
- [short edges\\_matrix](#) [MAX\_VERTEX\_NUM][MAX\_VERTEX\_NUM]  
*-1 if there is no [Edge](#) between the two Nodes, otherwise the index of the [Edge](#) in the [MST](#).*

### 8.9.1 Detailed Description

Minimum Spanning Tree, or [MST](#), and also a 1-Tree.

Definition at line 28 of file [mst.h](#).

### 8.9.2 Member Data Documentation

### 8.9.2.1 cost

```
double MST::cost
```

The total cost of the [MST](#), i.e. the sum of the weights of the Edges.

Definition at line 30 of file [mst.h](#).

### 8.9.2.2 edges

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

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

Definition at line 35 of file [mst.h](#).

### 8.9.2.3 edges\_matrix

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

-1 if there is no [Edge](#) between the two Nodes, otherwise the index of the [Edge](#) in the [MST](#).

Definition at line 36 of file [mst.h](#).

### 8.9.2.4 isValid

```
bool MST::isValid
```

True if the [MST](#) has the correct number of Edges, false otherwise.

Definition at line 29 of file [mst.h](#).

### 8.9.2.5 nodes

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

The set of Nodes in the [MST](#).

Definition at line 34 of file [mst.h](#).

#### 8.9.2.6 num\_edges

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

The number of Edges in the [MST](#).

Definition at line 33 of file [mst.h](#).

#### 8.9.2.7 num\_nodes

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

The number of Nodes in the [MST](#).

Definition at line 32 of file [mst.h](#).

#### 8.9.2.8 prob

```
double MST::prob
```

The probability of the [MST](#), i.e. the average of the probabilities of its Edges.

Definition at line 31 of file [mst.h](#).

## 8.10 Node Struct Reference

Structure of a [Node](#).

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

### Public Attributes

- double [x](#)  
*x coordinate of the [Node](#).*
- double [y](#)  
*y coordinate of the [Node](#).*
- unsigned short [positionInGraph](#)  
*Position of the [Node](#) in the list of Nodes of the [Graph](#), i.e. its unique ID.*
- unsigned short [num\\_neighbours](#)  
*Number of neighbours of the [Node](#).*
- unsigned short [neighbours](#) [MAX\_VERTEX\_NUM - 1]  
*Array of IDs of the [Node](#)'s neighbors.*

### 8.10.1 Detailed Description

Structure of a [Node](#).

Definition at line 30 of file [graph.h](#).

### 8.10.2 Member Data Documentation

#### 8.10.2.1 neighbours

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

Array of IDs of the [Node](#)'s neighbors.

Definition at line 35 of file [graph.h](#).

#### 8.10.2.2 num\_neighbours

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

Number of neighbours of the [Node](#).

Definition at line 34 of file [graph.h](#).

#### 8.10.2.3 positionInGraph

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

Position of the [Node](#) in the list of Nodes of the [Graph](#), i.e. its unique ID.

Definition at line 33 of file [graph.h](#).

#### 8.10.2.4 x

```
double Node::x
```

x coordinate of the [Node](#).

Definition at line 31 of file [graph.h](#).

### 8.10.2.5 y

```
double Node::y
```

y coordinate of the [Node](#).

Definition at line 32 of file [graph.h](#).

## 8.11 OrdTreeNode Struct Reference

A Heap-ordered Tree [Node](#) where the key of the parent is  $\leq$  the key of its children.

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

### Public Attributes

- unsigned short [key](#)  
*The key of the [Node](#).*
- double [value](#)  
*The value of the [Node](#).*
- struct [OrdTreeNode](#) \* [parent](#)  
*The parent of the [Node](#).*
- struct [OrdTreeNode](#) \* [left\\_sibling](#)  
*The left sibling of the [Node](#).*
- struct [OrdTreeNode](#) \* [right\\_sibling](#)  
*The right sibling of the [Node](#).*
- struct [OrdTreeNode](#) \* [head\\_child\\_list](#)  
*The head of the list of children of the [Node](#).*
- struct [OrdTreeNode](#) \* [tail\\_child\\_list](#)  
*The tail of the list of children of the [Node](#).*
- unsigned short [num\\_children](#)  
*The number of children of the [Node](#).*
- bool [marked](#)  
*True if the [Node](#) has lost a child, false otherwise.*
- bool [is\\_root](#)  
*True if the [Node](#) is a root, false otherwise.*

### 8.11.1 Detailed Description

A Heap-ordered Tree [Node](#) where the key of the parent is  $\leq$  the key of its children.

Definition at line 26 of file [fibonacci\\_heap.h](#).

### 8.11.2 Member Data Documentation



### 8.11.2.1 head\_child\_list

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

The head of the list of children of the [Node](#).

Definition at line 34 of file [fibonacci\\_heap.h](#).

### 8.11.2.2 is\_root

```
bool OrdTreeNode::is_root
```

True if the [Node](#) is a root, false otherwise.

Definition at line 38 of file [fibonacci\\_heap.h](#).

### 8.11.2.3 key

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

The key of the [Node](#).

Definition at line 27 of file [fibonacci\\_heap.h](#).

### 8.11.2.4 left\_sibling

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

The left sibling of the [Node](#).

Definition at line 31 of file [fibonacci\\_heap.h](#).

### 8.11.2.5 marked

```
bool OrdTreeNode::marked
```

True if the [Node](#) has lost a child, false otherwise.

Definition at line 37 of file [fibonacci\\_heap.h](#).

#### 8.11.2.6 num\_children

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

The number of children of the [Node](#).

Definition at line 36 of file [fibonacci\\_heap.h](#).

#### 8.11.2.7 parent

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

The parent of the [Node](#).

Definition at line 29 of file [fibonacci\\_heap.h](#).

#### 8.11.2.8 right\_sibling

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

The right sibling of the [Node](#).

Definition at line 32 of file [fibonacci\\_heap.h](#).

#### 8.11.2.9 tail\_child\_list

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

The tail of the list of children of the [Node](#).

Definition at line 35 of file [fibonacci\\_heap.h](#).

#### 8.11.2.10 value

```
double OrdTreeNode::value
```

The value of the [Node](#).

Definition at line 28 of file [fibonacci\\_heap.h](#).

## 8.12 Problem Struct Reference

The struct used to represent the overall problem.

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

### Public Attributes

- [Graph graph](#)  
*The [Graph](#) of the problem.*
- [Graph reformulationGraph](#)  
*The [Graph](#) used to perform the dual reformulation of [Edge](#) weights.*
- unsigned short [candidateNodeid](#)  
*The id of the candidate node.*
- unsigned short [totTreeLevels](#)  
*The total number of levels in the Branch and Bound tree.*
- [SubProblem bestSolution](#)  
*The best solution found so far.*
- double [bestValue](#)  
*The cost of the best solution found so far.*
- unsigned int [generatedBBNodes](#)  
*The number of nodes generated in the Branch and Bound tree.*
- unsigned int [exploredBBNodes](#)  
*The number of nodes explored in the Branch and Bound tree.*
- unsigned int [num\\_fixed\\_edges](#)  
*The number of fixed edges in the Branch and Bound tree.*
- bool [interrupted](#)  
*True if the algorithm has been interrupted by timeout.*
- clock\_t [start](#)  
*The time when the algorithm started.*
- clock\_t [end](#)  
*The time when the algorithm ended.*

### 8.12.1 Detailed Description

The struct used to represent the overall problem.

Definition at line 62 of file [b\\_and\\_b\\_data.h](#).

### 8.12.2 Member Data Documentation

#### 8.12.2.1 bestSolution

`SubProblem Problem::bestSolution`

The best solution found so far.

Definition at line 67 of file [b\\_and\\_b\\_data.h](#).

#### 8.12.2.2 bestValue

`double Problem::bestValue`

The cost of the best solution found so far.

Definition at line 68 of file [b\\_and\\_b\\_data.h](#).

#### 8.12.2.3 candidateNodeId

`unsigned short Problem::candidateNodeId`

The id of the candidate node.

Definition at line 65 of file [b\\_and\\_b\\_data.h](#).

#### 8.12.2.4 end

`clock_t Problem::end`

The time when the algorithm ended.

Definition at line 74 of file [b\\_and\\_b\\_data.h](#).

#### 8.12.2.5 exploredBBNodes

`unsigned int Problem::exploredBBNodes`

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

Definition at line 70 of file [b\\_and\\_b\\_data.h](#).

### 8.12.2.6 generatedBBNodes

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

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

Definition at line 69 of file [b\\_and\\_b\\_data.h](#).

### 8.12.2.7 graph

```
Graph Problem::graph
```

The [Graph](#) of the problem.

Definition at line 63 of file [b\\_and\\_b\\_data.h](#).

### 8.12.2.8 interrupted

```
bool Problem::interrupted
```

True if the algorithm has been interrupted by timeout.

Definition at line 72 of file [b\\_and\\_b\\_data.h](#).

### 8.12.2.9 num\_fixed\_edges

```
unsigned int Problem::num_fixed_edges
```

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

Definition at line 71 of file [b\\_and\\_b\\_data.h](#).

### 8.12.2.10 reformulationGraph

```
Graph Problem::reformulationGraph
```

The [Graph](#) used to perform the dual reformulation of [Edge](#) weights.

Definition at line 64 of file [b\\_and\\_b\\_data.h](#).

### 8.12.2.11 start

```
clock_t Problem::start
```

The time when the algorithm started.

Definition at line 73 of file [b\\_and\\_b\\_data.h](#).

### 8.12.2.12 totTreeLevels

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

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

Definition at line 66 of file [b\\_and\\_b\\_data.h](#).

## 8.13 Set Struct Reference

A [Set](#) is a node in the [Forest](#).

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

### Public Attributes

- struct [Set](#) \* [parentSet](#)  
*Pointer to the parent [Set](#) in a tree representation of the [Forest](#).*
- unsigned short [rango](#)  
*Rank of the [Set](#), used to optimize the find operation.*
- [Node](#) [curr](#)  
*Current [Node](#).*
- unsigned short [num\\_in\\_forest](#)  
*Number of the position of the [Set](#) in the [Forest](#).*

### 8.13.1 Detailed Description

A [Set](#) is a node in the [Forest](#).

Definition at line 19 of file [mfset.h](#).

### 8.13.2 Member Data Documentation

### 8.13.2.1 curr

```
Node Set::curr
```

Current [Node](#).

Definition at line 22 of file [mfset.h](#).

### 8.13.2.2 num\_in\_forest

```
unsigned short Set::num_in_forest
```

Number of the position of the [Set](#) in the [Forest](#).

Definition at line 23 of file [mfset.h](#).

### 8.13.2.3 parentSet

```
struct Set* Set::parentSet
```

Pointer to the parent [Set](#) in a tree representation of the [Forest](#).

Definition at line 20 of file [mfset.h](#).

### 8.13.2.4 rango

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

Rank of the [Set](#), used to optimize the find operation.

Definition at line 21 of file [mfset.h](#).

## 8.14 SubProblem Struct Reference

The struct used to represent a [SubProblem](#) or node of the Branch and Bound tree.

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

## Public Attributes

- [BBNodeType](#) type  
*The label of the [SubProblem](#).*
- unsigned int [id](#)  
*The id of the [SubProblem](#), an incremental number.*
- unsigned int [fatherId](#)  
*The id of the father of the [SubProblem](#).*
- double [value](#)  
*The cost of the [SubProblem](#).*
- unsigned short [treeLevel](#)  
*The level of the [SubProblem](#) in the Branch and Bound tree.*
- float [timeToReach](#)  
*The time needed to reach the [SubProblem](#), in seconds.*
- [MST](#) [oneTree](#)  
*The 1Tree of the [SubProblem](#).*
- unsigned short [num\\_edges\\_in\\_cycle](#)  
*The number of edges in the cycle of the [SubProblem](#).*
- double [prob](#)  
*The probability of the [SubProblem](#) to be the best tour.*
- [ConstrainedEdge](#) [cycleEdges](#) [MAX\_VERTEX\_NUM]  
*The edges in the cycle of the [SubProblem](#).*
- unsigned short [num\\_forbidden\\_edges](#)  
*The number of forbidden edges in the [SubProblem](#).*
- unsigned short [num\\_mandatory\\_edges](#)  
*The number of mandatory edges in the [SubProblem](#).*
- int [edge\\_to\\_branch](#)  
*The id of the edge to branch in the [SubProblem](#).*
- [ConstrainedEdge](#) [mandatoryEdges](#) [MAX\_VERTEX\_NUM]  
*The mandatory edges in the [SubProblem](#).*
- [ConstraintType](#) [constraints](#) [MAX\_VERTEX\_NUM][MAX\_VERTEX\_NUM]  
*The constraints of the edges in the [SubProblem](#).*

### 8.14.1 Detailed Description

The struct used to represent a [SubProblem](#) or node of the Branch and Bound tree.

Definition at line 42 of file [b\\_and\\_b\\_data.h](#).

### 8.14.2 Member Data Documentation

#### 8.14.2.1 constraints

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

The constraints of the edges in the [SubProblem](#).

Definition at line 57 of file [b\\_and\\_b\\_data.h](#).



### 8.14.2.2 cycleEdges

`ConstrainedEdge SubProblem::cycleEdges[MAX_VERTEX_NUM]`

The edges in the cycle of the [SubProblem](#).

Definition at line 52 of file [b\\_and\\_b\\_data.h](#).

### 8.14.2.3 edge\_to\_branch

`int SubProblem::edge_to_branch`

The id of the edge to branch in the [SubProblem](#).

Definition at line 55 of file [b\\_and\\_b\\_data.h](#).

### 8.14.2.4 fatherId

`unsigned int SubProblem::fatherId`

The id of the father of the [SubProblem](#).

Definition at line 45 of file [b\\_and\\_b\\_data.h](#).

### 8.14.2.5 id

`unsigned int SubProblem::id`

The id of the [SubProblem](#), an incremental number.

Definition at line 44 of file [b\\_and\\_b\\_data.h](#).

### 8.14.2.6 mandatoryEdges

`ConstrainedEdge SubProblem::mandatoryEdges[MAX_VERTEX_NUM]`

The mandatory edges in the [SubProblem](#).

Definition at line 56 of file [b\\_and\\_b\\_data.h](#).

#### 8.14.2.7 num\_edges\_in\_cycle

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

The number of edges in the cycle of the [SubProblem](#).

Definition at line 50 of file [b\\_and\\_b\\_data.h](#).

#### 8.14.2.8 num\_forbidden\_edges

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

The number of forbidden edges in the [SubProblem](#).

Definition at line 53 of file [b\\_and\\_b\\_data.h](#).

#### 8.14.2.9 num\_mandatory\_edges

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

The number of mandatory edges in the [SubProblem](#).

Definition at line 54 of file [b\\_and\\_b\\_data.h](#).

#### 8.14.2.10 oneTree

```
MST SubProblem::oneTree
```

The 1Tree of the [SubProblem](#).

Definition at line 49 of file [b\\_and\\_b\\_data.h](#).

#### 8.14.2.11 prob

```
double SubProblem::prob
```

The probability of the [SubProblem](#) to be the best tour.

Definition at line 51 of file [b\\_and\\_b\\_data.h](#).

#### 8.14.2.12 timeToReach

```
float SubProblem::timeToReach
```

The time needed to reach the [SubProblem](#), in seconds.

Definition at line 48 of file [b\\_and\\_b\\_data.h](#).

#### 8.14.2.13 treeLevel

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

The level of the [SubProblem](#) in the Branch and Bound tree.

Definition at line 47 of file [b\\_and\\_b\\_data.h](#).

#### 8.14.2.14 type

```
BBNodeType SubProblem::type
```

The label of the [SubProblem](#).

Definition at line 43 of file [b\\_and\\_b\\_data.h](#).

#### 8.14.2.15 value

```
double SubProblem::value
```

The cost of the [SubProblem](#).

Definition at line 46 of file [b\\_and\\_b\\_data.h](#).

## 8.15 SubProblemElem Struct Reference

The element of the list of SubProblems.

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

## Public Attributes

- [SubProblem](#) `subProblem`  
*The [SubProblem](#).*
- struct [SubProblemElem](#) \* `next`  
*The next element of the list.*
- struct [SubProblemElem](#) \* `prev`  
*The previous element of the list.*

### 8.15.1 Detailed Description

The element of the list of SubProblems.

Definition at line 79 of file [b\\_and\\_b\\_data.h](#).

### 8.15.2 Member Data Documentation

#### 8.15.2.1 next

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

The next element of the list.

Definition at line 81 of file [b\\_and\\_b\\_data.h](#).

#### 8.15.2.2 prev

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

The previous element of the list.

Definition at line 82 of file [b\\_and\\_b\\_data.h](#).

#### 8.15.2.3 subProblem

```
SubProblem SubProblemElem::subProblem
```

The [SubProblem](#).

Definition at line 80 of file [b\\_and\\_b\\_data.h](#).

## 8.16 SubProblemsList Struct Reference

The list of open SubProblems.

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

### Public Attributes

- [SubProblemElem](#) \* [head](#)  
*The head of the list.*
- [SubProblemElem](#) \* [tail](#)  
*The tail of the list.*
- [size\\_t](#) [size](#)  
*The size of the list.*

### 8.16.1 Detailed Description

The list of open SubProblems.

Definition at line 87 of file [b\\_and\\_b\\_data.h](#).

### 8.16.2 Member Data Documentation

#### 8.16.2.1 head

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

The head of the list.

Definition at line 88 of file [b\\_and\\_b\\_data.h](#).

#### 8.16.2.2 size

```
size\_t SubProblemsList::size
```

The size of the list.

Definition at line 90 of file [b\\_and\\_b\\_data.h](#).

### 8.16.2.3 tail

`SubProblemElem* SubProblemsList::tail`

The tail of the list.

Definition at line 89 of file [b\\_and\\_b\\_data.h](#).

## 8.17 SubProblemsListIterator Struct Reference

The iterator of the list of SubProblems.

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

### Public Attributes

- `SubProblemsList * list`  
*The list to iterate.*
- `SubProblemElem * curr`  
*The current element of the list.*
- `size_t index`  
*The index of the current element of the list.*

### 8.17.1 Detailed Description

The iterator of the list of SubProblems.

Definition at line 95 of file [b\\_and\\_b\\_data.h](#).

### 8.17.2 Member Data Documentation

#### 8.17.2.1 curr

`SubProblemElem* SubProblemsListIterator::curr`

The current element of the list.

Definition at line 97 of file [b\\_and\\_b\\_data.h](#).

#### 8.17.2.2 index

`size_t SubProblemsListIterator::index`

The index of the current element of the list.

Definition at line 98 of file [b\\_and\\_b\\_data.h](#).

#### 8.17.2.3 list

`SubProblemsList* SubProblemsListIterator::list`

The list to iterate.

Definition at line 96 of file [b\\_and\\_b\\_data.h](#).

## Chapter 9

# File Documentation

### 9.1 GraphConvolutionalBranchandBound/src/Hybrid↔ Solver/main/algorithms/branch\_and\_bound.c File Reference

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

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

#### Functions

- void `dfs` (`SubProblem` \*subProblem)  
*A Depth First Search algorithm on a [Graph](#).*
- bool `check_hamiltonian` (`SubProblem` \*subProblem)  
*Function that checks if the 1Tree of a [SubProblem](#) is a tour.*
- `BBNodeType` `mst_to_one_tree` (`SubProblem` \*currentSubproblem, `Graph` \*graph)  
*Transforms a [MST](#) into a 1Tree.*
- void `initialize_matrix` (`SubProblem` \*subProblem)
- bool `infer_constraints` (`SubProblem` \*subProblem)  
*Infer the values of some edge variables of a [SubProblem](#).*
- void `copy_constraints` (`SubProblem` \*subProblem, const `SubProblem` \*otherSubProblem)  
*Copy the matrix of constraints of a [SubProblem](#) into another.*
- void `branch` (`SubProblemsList` \*openSubProblems, `SubProblem` \*currentSubProblem)  
*The Shuttler's branching rule.*
- double `max_edge_path_1Tree` (`SubProblem` \*currentSubProb, double \*replacement\_costs, unsigned short start\_node, unsigned short end\_node)
- int `variable_fixing` (`SubProblem` \*currentSubProb)  
*The function used to fix the edge variables to be mandatory or forbidden.*
- void `constrained_kruskal` (`Graph` \*graph, `SubProblem` \*subProblem, unsigned short candidateId)
- void `constrained_prim` (`Graph` \*graph, `SubProblem` \*subProblem, unsigned short candidateId)
- bool `compare_subproblems` (const `SubProblem` \*a, const `SubProblem` \*b)  
*Compare two OPEN SubProblems.*
- void `bound` (`SubProblem` \*currentSubProb)  
*The Held-Karp bound function with the subgradient algorithm.*
- bool `time_limit_reached` (void)

- *Check if the time limit has been reached.*
- void `nearest_prob_neighbour` (unsigned short start\_node)
  - This function is used to find the first feasible tour.*
- bool `compare_candidate_node` (SubProblem \*a, SubProblem \*b)
- unsigned short `find_candidate_node` (void)
  - Select the candidate Node, i.e. the starting vertex of the tour.*
- bool `check_feasibility` (Graph \*graph)
  - Check if the Graph associated to the Problem is feasible.*
- void `branch_and_bound` (Problem \*current\_problem)
  - The Branch and Bound algorithm.*
- void `set_problem` (Problem \*current\_problem)
  - Define the problem to solve.*
- void `print_subProblem` (const SubProblem \*subProblem)
  - Get all metrics of a certain SubProblem.*
- void `print_problem` (void)
  - Get all metrics of the problem.*

### 9.1.1 Detailed Description

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

#### Author

Lorenzo Sciandra

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

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file `branch_and_bound.c`.

### 9.1.2 Function Documentation

#### 9.1.2.1 bound()

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

The Held-Karp bound function with the subgradient algorithm.

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



## Parameters

<i>current_problem</i>	The pointer to the <a href="#">SubProblem</a> or branch-and-bound <a href="#">Node</a> in the tree.
------------------------	---

Definition at line 657 of file [branch\\_and\\_bound.c](#).

### 9.1.2.2 branch()

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

The Shutler's branching rule.

Every [SubProblem](#) is branched into 2 new SubProblems, one including the "edge\_to\_branch" and the other not. More details at <http://www.jstor.org/stable/254144>.

## Parameters

<i>openSubProblems</i>	The list of open SubProblems, to which the new SubProblems will be added.
<i>subProblem</i>	The <a href="#">SubProblem</a> to branch.

Definition at line 256 of file [branch\\_and\\_bound.c](#).

### 9.1.2.3 branch\_and\_bound()

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

The Branch and Bound algorithm.

This is the main function of the Branch and Bound algorithm. It stores all the open SubProblems in a [SubProblemsList](#) and analyzes them one by one with the [branch\(\)](#) and [held\\_karp\\_bound\(\)](#) functions.

## Parameters

<i>current_problem</i>	The pointer to the problem to solve.
------------------------	--------------------------------------

Definition at line 1012 of file [branch\\_and\\_bound.c](#).

### 9.1.2.4 check\_feasibility()

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

Check if the [Graph](#) associated to the [Problem](#) is feasible.

A [Graph](#) is feasible if every [Node](#) has at least degree 2.

#### Parameters

<i>graph</i>	The <a href="#">Graph</a> to check.
--------------	-------------------------------------

#### Returns

true if the [Graph](#) is feasible, false otherwise.

Definition at line 997 of file [branch\\_and\\_bound.c](#).

### 9.1.2.5 `check_hamiltonian()`

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

Function that checks if the 1Tree of a [SubProblem](#) is a tour.

This is done by simply check if all the edges are in the cycle passing through the candidate [Node](#).

#### Parameters

<i>subProblem</i>	The <a href="#">SubProblem</a> to check.
-------------------	--

#### Returns

true if the [SubProblem](#) is a Hamiltonian cycle, false otherwise.

Definition at line 82 of file [branch\\_and\\_bound.c](#).

### 9.1.2.6 `compare_candidate_node()`

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

Definition at line 951 of file [branch\\_and\\_bound.c](#).

### 9.1.2.7 compare\_subproblems()

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

Compare two OPEN SubProblems.

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

## Parameters

<i>a</i>	The first <a href="#">SubProblem</a> to compare.
<i>b</i>	The second <a href="#">SubProblem</a> to compare.

## Returns

true if the first [SubProblem](#) is better than the second, false otherwise.

Definition at line 647 of file [branch\\_and\\_bound.c](#).

**9.1.2.8 constrained\_kruskal()**

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

Definition at line 503 of file [branch\\_and\\_bound.c](#).

**9.1.2.9 constrained\_prim()**

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

Definition at line 565 of file [branch\\_and\\_bound.c](#).

**9.1.2.10 copy\_constraints()**

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

Copy the matrix of constraints of a [SubProblem](#) into another.

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

## Parameters

<i>subProblem</i>	The <a href="#">SubProblem</a> to which the ConstraintType will be copied.
<i>otherSubProblem</i>	The <a href="#">SubProblem</a> from which the ConstraintType will be copied.

Definition at line 234 of file [branch\\_and\\_bound.c](#).

#### 9.1.2.11 dfs()

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

A Depth First Search algorithm on a [Graph](#).

This function is used to find the cycle in the 1Tree [SubProblem](#), passing through the candidate [Node](#).

##### Parameters

<i>subProblem</i>	The <a href="#">SubProblem</a> to inspect.
-------------------	--

Definition at line 18 of file [branch\\_and\\_bound.c](#).

#### 9.1.2.12 find\_candidate\_node()

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

Select the candidate [Node](#), i.e. the starting vertex of the tour.

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

##### Returns

the candidate [Node](#) id.

Definition at line 961 of file [branch\\_and\\_bound.c](#).

#### 9.1.2.13 infer\_constraints()

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

Infer the values of some edge variables of a [SubProblem](#).

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

## Parameters

<i>subProblem</i>	The <a href="#">SubProblem</a> to which we want to infer the variables values.
-------------------	--

## Returns

true if the subproblem remains feasible, false otherwise.

Definition at line [185](#) of file [branch\\_and\\_bound.c](#).

**9.1.2.14 initialize\_matrix()**

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

Definition at line [167](#) of file [branch\\_and\\_bound.c](#).

**9.1.2.15 max\_edge\_path\_1Tree()**

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

Definition at line [342](#) of file [branch\\_and\\_bound.c](#).

**9.1.2.16 mst\_to\_one\_tree()**

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

Transforms a [MST](#) into a 1Tree.

This is done by adding the two least-cost edges incident to the candidate [Node](#) in the [MST](#).

## Parameters

<i>currentSubproblem</i>	The <a href="#">SubProblem</a> to which the <a href="#">MST</a> belongs.
<i>graph</i>	The <a href="#">Graph</a> of the <a href="#">Problem</a> .

## Returns

an enum value that indicates if the [SubProblem](#) is feasible or not.

Definition at line 88 of file [branch\\_and\\_bound.c](#).

### 9.1.2.17 nearest\_prob\_neighbour()

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

This function is used to find the first feasible tour.

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

## Parameters

<i>start_node</i>	The <a href="#">Node</a> from which the tour will start.
-------------------	--

Definition at line 844 of file [branch\\_and\\_bound.c](#).

### 9.1.2.18 print\_problem()

```
void print_problem (
    void )
```

Get all metrics of the problem.

It is used at the end of the algorithm to print the solution obtained. It calls the [print\\_subProblem\(\)](#) function on the best [SubProblem](#) found.

Definition at line 1138 of file [branch\\_and\\_bound.c](#).

### 9.1.2.19 print\_subProblem()

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

Get all metrics of a certain [SubProblem](#).

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

**Parameters**

<i>subProblem</i>	The <a href="#">SubProblem</a> to print.
-------------------	--

Definition at line [1073](#) of file [branch\\_and\\_bound.c](#).

**9.1.2.20 set\_problem()**

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

Define the problem to solve.

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

**Parameters**

<i>current_problem</i>	The pointer to the problem to solve.
------------------------	--------------------------------------

Definition at line [1068](#) of file [branch\\_and\\_bound.c](#).

**9.1.2.21 time\_limit\_reached()**

```
bool time_limit_reached (
    void )
```

Check if the time limit has been reached.

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

**Returns**

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

Definition at line [839](#) of file [branch\\_and\\_bound.c](#).

**9.1.2.22 variable\_fixing()**

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

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

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



## Parameters

<i>subProblem</i>	The <a href="#">SubProblem</a> that we want to add the constraints to.
-------------------	--

## Returns

the num of variables fixed.

Definition at line 421 of file [branch\\_and\\_bound.c](#).

## 9.2 branch\_and\_bound.c

[Go to the documentation of this file.](#)

```

00001
00015 #include "branch_and_bound.h"
00016
00017
00018 void dfs(SubProblem *subProblem) {
00019     List *stack = new_list();
00020     unsigned short num_nodes = subProblem->oneTree.num_nodes;
00021     Node start;
00022     int parentId[num_nodes];
00023     unsigned short pathLength[num_nodes];
00024     bool visited[num_nodes];
00025
00026     for (unsigned short i = 0; i < num_nodes; i++) {
00027         Node current = subProblem->oneTree.nodes[i];
00028         if (current.positionInGraph == problem->graph.nodes[problem->candidateNodeId].positionInGraph)
00029         {
00030             start = current;
00031             parentId[i] = -1;
00032             pathLength[i] = 0;
00033             visited[i] = false;
00034         }
00035
00036         add_elem_list_bottom(stack, &start);
00037
00038         while (stack->size > 0 && parentId[start.positionInGraph] == -1) {
00039             Node *currentNode = get_list_elem_index(stack, 0);
00040             delete_list_elem_index(stack, 0);
00041             if (!visited[currentNode->positionInGraph]) {
00042                 visited[currentNode->positionInGraph] = true;
00043                 for (unsigned short i = 0; i < currentNode->num_neighbours; i++) {
00044                     unsigned short dest = currentNode->neighbours[i];
00045                     if (!visited[dest]) {
00046                         pathLength[dest] = pathLength[currentNode->positionInGraph] + 1;
00047                         parentId[dest] = currentNode->positionInGraph;
00048                         Node *neighbour = &subProblem->oneTree.nodes[dest];
00049                         add_elem_list_index(stack, neighbour, 0);
00050                     } else if (parentId[dest] == -1) {
00051                         // start node
00052                         unsigned short path = pathLength[currentNode->positionInGraph] + 1;
00053                         if (path > 2) {
00054                             parentId[dest] = currentNode->positionInGraph;
00055                             pathLength[dest] = path;
00056                         }
00057                     }
00058                 }
00059             }
00060         }
00061         del_list(stack);
00062
00063         int fromNode = -1;
00064         int toNode = start.positionInGraph;
00065
00066         //printf("Path Length: %d\n", pathLength[toNode]);
00067
00068         if (pathLength[toNode] > 2) {
00069             while (fromNode != start.positionInGraph) {
00070                 fromNode = parentId[toNode];
00071                 Edge current_in_cycle = problem->graph.edges_matrix[fromNode][toNode];
00072                 subProblem->cycleEdges[subProblem->num_edges_in_cycle].src = current_in_cycle.src;
00073                 subProblem->cycleEdges[subProblem->num_edges_in_cycle].dest = current_in_cycle.dest;

```

```

00074         subProblem->num_edges_in_cycle++;
00075         toNode = fromNode;
00076     }
00077 }
00078
00079 }
00080
00081
00082 bool check_hamiltonian(SubProblem *subProblem) {
00083     dfs(subProblem);
00084     return subProblem->num_edges_in_cycle == subProblem->oneTree.num_edges;
00085 }
00086
00087
00088 BBNodeType mst_to_one_tree(SubProblem *currentSubproblem, Graph *graph) {
00089     Node candidate = graph->nodes[problem->candidateNodeId];
00090     int bestEdgesPos[2];
00091     double bestEdgesWeight[2];
00092     unsigned short src = candidate.positionInGraph;
00093     unsigned short others [candidate.num_neighbours];
00094     unsigned short num_others = 0;
00095     unsigned short toAdd = 0;
00096
00097     for (unsigned short i = 0; i < candidate.num_neighbours && toAdd<=2; i++) {
00098         unsigned short dest = candidate.neighbours[i];
00099
00100         if (currentSubproblem->num_mandatory_edges > 0 &&
00101             currentSubproblem->constraints[src][dest] == MANDATORY) {
00102             bestEdgesWeight[toAdd] = graph->edges_matrix[src][dest].weight;
00103             bestEdgesPos[toAdd] = graph->edges_matrix[src][dest].positionInGraph;
00104             toAdd++;
00105         } else if (currentSubproblem->num_forbidden_edges == 0 ||
00106             currentSubproblem->constraints[src][dest] == NOTHING) {
00107             others[num_others] = dest;
00108             num_others++;
00109         }
00110     }
00111
00112     if (toAdd > 2 || (toAdd + num_others) < 2) {
00113         return CLOSED_UNFEASIBLE;
00114     } else if (toAdd == 2) {
00115         add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[0]]);
00116         add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[1]]);
00117         return OPEN;
00118     } else if (toAdd == 1) {
00119         add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[0]]);
00120
00121         double bestFoundWeight = INFINITE;
00122         int bestFoundPos = -1;
00123
00124         for (unsigned short j = 0; j < num_others; j++) {
00125             unsigned short dest = others[j];
00126             Edge candidateEdge = graph->edges_matrix[src][dest];
00127
00128             if (bestFoundWeight > candidateEdge.weight) {
00129                 bestFoundPos = candidateEdge.positionInGraph;
00130                 bestFoundWeight = candidateEdge.weight;
00131             }
00132         }
00133
00134         add_edge(&currentSubproblem->oneTree, &graph->edges[bestFoundPos]);
00135
00136         return OPEN;
00137     } else {
00138         bestEdgesPos[0] = -1;
00139         bestEdgesPos[1] = -1;
00140         bestEdgesWeight[0] = INFINITE;
00141         bestEdgesWeight[1] = INFINITE;
00142
00143         for (unsigned short j = 0; j < num_others; j++) {
00144             unsigned short dest = others[j];
00145             Edge candidateEdge = graph->edges_matrix[src][dest];
00146
00147             if (bestEdgesWeight[0] > candidateEdge.weight) {
00148                 bestEdgesPos[1] = bestEdgesPos[0];
00149                 bestEdgesWeight[1] = bestEdgesWeight[0];
00150                 bestEdgesPos[0] = candidateEdge.positionInGraph;
00151                 bestEdgesWeight[0] = candidateEdge.weight;
00152             } else if (bestEdgesWeight[1] > candidateEdge.weight) {
00153                 bestEdgesPos[1] = candidateEdge.positionInGraph;
00154                 bestEdgesWeight[1] = candidateEdge.weight;
00155             }
00156         }
00157     }
00158
00159     add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[0]]);
00160     add_edge(&currentSubproblem->oneTree, &graph->edges[bestEdgesPos[1]]);

```

```

00161
00162     return OPEN;
00163 }
00164 }
00165
00166
00167 void initialize_matrix(SubProblem *subProblem) {
00168
00169     subProblem->num_mandatory_edges = 0;
00170     subProblem->num_forbidden_edges = 0;
00171     for (short i = 0; i < MAX_VERTEX_NUM; i++) {
00172         for (short j = i; j < MAX_VERTEX_NUM; j++) {
00173             if(j == i){
00174                 subProblem->constraints[i][j] = FORBIDDEN;
00175
00176             } else{
00177                 subProblem->constraints[i][j] = NOTHING;
00178                 subProblem->constraints[j][i] = NOTHING;
00179             }
00180         }
00181     }
00182 }
00183
00184
00185 bool infer_constraints(SubProblem * subProblem){
00186
00187     bool valid = true;
00188
00189     for (short i = 0; i < MAX_VERTEX_NUM && valid; i++) {
00190
00191         short num_nothing_node = 0;
00192         short num_mandatory_node = 0;
00193         short num_forbidden_node = 0;
00194         short nothing_nodes[MAX_VERTEX_NUM];
00195
00196         for (short j = 0; j < MAX_VERTEX_NUM; j++) {
00197             if (subProblem->constraints[i][j] == NOTHING) {
00198                 nothing_nodes[num_nothing_node] = j;
00199                 num_nothing_node++;
00200             } else if (subProblem->constraints[i][j] == MANDATORY) {
00201                 num_mandatory_node++;
00202             } else {
00203                 num_forbidden_node++;
00204             }
00205         }
00206
00207         if (num_mandatory_node == 2) {
00208             for (short j = 0; j < num_nothing_node; j++) {
00209                 subProblem->constraints[i][nothing_nodes[j]] = FORBIDDEN;
00210                 subProblem->constraints[nothing_nodes[j]][i] = FORBIDDEN;
00211                 subProblem->num_forbidden_edges++;
00212                 problem->num_fixed_edges++;
00213             }
00214         } else if (num_mandatory_node > 2 || (MAX_VERTEX_NUM - num_forbidden_node < 2)) {
00215             valid = false;
00216         } else if (MAX_VERTEX_NUM - num_forbidden_node == 2) {
00217
00218             if (num_nothing_node + num_mandatory_node == 2) {
00219                 for (short j = 0; j < num_nothing_node; j++) {
00220                     subProblem->constraints[i][nothing_nodes[j]] = MANDATORY;
00221                     subProblem->constraints[nothing_nodes[j]][i] = MANDATORY;
00222                     subProblem->mandatoryEdges[subProblem->num_mandatory_edges].src = i;
00223                     subProblem->mandatoryEdges[subProblem->num_mandatory_edges].dest =
nothing_nodes[j];
00224                     subProblem->num_mandatory_edges++;
00225                     problem->num_fixed_edges++;
00226                 }
00227             }
00228         }
00229     }
00230     return valid;
00231 }
00232
00233
00234 void copy_constraints(SubProblem *subProblem, const SubProblem *otherSubProblem) {
00235     subProblem->num_mandatory_edges = 0;
00236     subProblem->num_forbidden_edges = 0;
00237     for (short i = 0; i < MAX_VERTEX_NUM; i++) {
00238
00239         for (short j = i; j < MAX_VERTEX_NUM; j++) {
00240             subProblem->constraints[i][j] = otherSubProblem->constraints[i][j];
00241             subProblem->constraints[j][i] = otherSubProblem->constraints[j][i];
00242
00243             if(subProblem->constraints[i][j] == MANDATORY){
00244                 subProblem->mandatoryEdges[subProblem->num_mandatory_edges].src = i;
00245                 subProblem->mandatoryEdges[subProblem->num_mandatory_edges].dest = j;
00246                 subProblem->num_mandatory_edges++;

```

```

00247         }
00248         else if(subProblem->constraints[i][j] == FORBIDDEN){
00249             subProblem->num_forbidden_edges++;
00250         }
00251     }
00252 }
00253 }
00254
00255 void branch(SubProblemsList *openSubProblems, SubProblem *currentSubProblem){
00256     if (currentSubProblem->treeLevel + 1 > problem->totTreeLevels) {
00257         problem->totTreeLevels++;
00258     }
00259     Edge to_branch = problem->graph.edges[currentSubProblem->edge_to_branch];
00260     double prob_edge = to_branch.prob;
00261
00262     for (short i = 0; i < 2; i++) {
00263         problem->generatedBBNodes++;
00264         SubProblem child;
00265         child.num_edges_in_cycle = 0;
00266         child.type = OPEN;
00267         child.prob = currentSubProblem->prob;
00268         child.id = problem->generatedBBNodes;
00269         child.fatherId = currentSubProblem->id;
00270         child.value = currentSubProblem->value;
00271         child.treeLevel = currentSubProblem->treeLevel + 1;
00272         child.edge_to_branch = -1;
00273         copy_constraints(&child, currentSubProblem);
00274
00275         if (HYBRID) {
00276             if (prob_edge >= PROB_BRANCH) {
00277                 if (i == 0) {
00278                     child.constraints[to_branch.src][to_branch.dest] = MANDATORY;
00279                     child.constraints[to_branch.dest][to_branch.src] = MANDATORY;
00280                     child.mandatoryEdges[child.num_mandatory_edges].src = to_branch.src;
00281                     child.mandatoryEdges[child.num_mandatory_edges].dest = to_branch.dest;
00282                     child.num_mandatory_edges++;
00283                 } else {
00284                     child.constraints[to_branch.src][to_branch.dest] = FORBIDDEN;
00285                     child.constraints[to_branch.dest][to_branch.src] = FORBIDDEN;
00286                     child.num_forbidden_edges++;
00287                 }
00288             } else {
00289                 if (i == 0) {
00290                     child.constraints[to_branch.src][to_branch.dest] = FORBIDDEN;
00291                     child.constraints[to_branch.dest][to_branch.src] = FORBIDDEN;
00292                     child.num_forbidden_edges++;
00293                 } else {
00294                     child.constraints[to_branch.src][to_branch.dest] = MANDATORY;
00295                     child.constraints[to_branch.dest][to_branch.src] = MANDATORY;
00296                     child.mandatoryEdges[child.num_mandatory_edges].src = to_branch.src;
00297                     child.mandatoryEdges[child.num_mandatory_edges].dest = to_branch.dest;
00298                     child.num_mandatory_edges++;
00299                 }
00300             }
00301         }
00302     }
00303 }
00304 }
00305 }
00306 else{
00307     if (i == 0) {
00308         child.constraints[to_branch.src][to_branch.dest] = MANDATORY;
00309         child.constraints[to_branch.dest][to_branch.src] = MANDATORY;
00310         child.mandatoryEdges[child.num_mandatory_edges].src = to_branch.src;
00311         child.mandatoryEdges[child.num_mandatory_edges].dest = to_branch.dest;
00312         child.num_mandatory_edges++;
00313     } else {
00314         child.constraints[to_branch.src][to_branch.dest] = FORBIDDEN;
00315         child.constraints[to_branch.dest][to_branch.src] = FORBIDDEN;
00316         child.num_forbidden_edges++;
00317     }
00318 }
00319
00320 if(infer_constraints(&child)){
00321     long position = -1;
00322
00323     SubProblemsListIterator *subProblem_iterators = create_SubProblemList_iterator(
00324         openSubProblems);
00325     for (size_t j = 0; j < openSubProblems->size && position == -1; j++) {
00326         SubProblem *open_subProblem = SubProblemList_iterator_get_next(subProblem_iterators);
00327         if (compare_subproblems(&child, open_subProblem)) {
00328             position = (long) j;
00329         }
00330     }
00331     delete_SubProblemList_iterator(subProblem_iterators);
00332     if (position == -1) {
00333         add_elem_SubProblemList_bottom(openSubProblems, &child);

```

```

00334         } else {
00335             add_elem_SubProblemList_index(openSubProblems, &child, position);
00336         }
00337     }
00338 }
00339 }
00340
00341
00342 double max_edge_path_lTree(SubProblem *currentSubProb, double *replacement_costs,
00343     unsigned short start_node, unsigned short end_node){
00344     List *stack = new_list();
00345     unsigned short num_nodes = currentSubProb->oneTree.num_nodes;
00346     Node start;
00347
00348     bool visited[num_nodes];
00349     int parentId[num_nodes];
00350
00351     for (unsigned short i = 0; i < num_nodes; i++) {
00352         Node current = currentSubProb->oneTree.nodes[i];
00353         if (current.positionInGraph == start_node) {
00354             start = current;
00355         }
00356         visited[i] = false;
00357         parentId[i] = -1;
00358     }
00359
00360     add_elem_list_bottom(stack, &start);
00361
00362     while (stack->size > 0 && parentId[end_node] == -1) {
00363         Node *currentNode = get_list_elem_index(stack, 0);
00364         delete_list_elem_index(stack, 0);
00365         if (!visited[currentNode->positionInGraph]) {
00366             visited[currentNode->positionInGraph] = true;
00367             for (unsigned short i = 0; i < currentNode->num_neighbours; i++) {
00368                 unsigned short dest = currentNode->neighbours[i];
00369                 if (!visited[dest] && problem->candidateNodeId != dest) {
00370                     parentId[dest] = currentNode->positionInGraph;
00371                     Node *neighbour = &currentSubProb->oneTree.nodes[dest];
00372                     add_elem_list_index(stack, neighbour, 0);
00373                 }
00374             }
00375         }
00376     }
00377     del_list(stack);
00378
00379     unsigned short fromNode = -1;
00380     unsigned short toNode = end_node;
00381     double max_edge_weight = -1;
00382     double edge_nin_ltree_weight = problem->graph.edges_matrix[start_node][end_node].weight;
00383
00384     while (fromNode != start_node) {
00385         fromNode = parentId[toNode];
00386     }
00387     short position_in_l_tree = -1;
00388
00389     if (currentSubProb->oneTree.edges_matrix[fromNode][toNode] != -1) {
00390         if (currentSubProb->constraints[fromNode][toNode] == NOTHING) {
00391             position_in_l_tree = currentSubProb->oneTree.edges_matrix[fromNode][toNode];
00392             Edge current_in_path = currentSubProb->oneTree.edges[position_in_l_tree];
00393             double current_edge_weight =
00394                 problem->graph.edges_matrix[current_in_path.src][current_in_path.dest].weight;
00395             if (current_edge_weight > max_edge_weight) {
00396                 max_edge_weight = current_edge_weight;
00397             }
00398             if (edge_nin_ltree_weight < current_edge_weight +
00399                 replacement_costs[current_in_path.positionInGraph]) {
00400                 replacement_costs[current_in_path.positionInGraph] = edge_nin_ltree_weight +
00401                     current_edge_weight;
00402             }
00403             toNode = fromNode;
00404         } else {
00405             printf("ERROR: Edge not found in l-tree\n");
00406             exit(1);
00407         }
00408     }
00409
00410     return max_edge_weight == -1 ? -INFINITE: max_edge_weight;
00411 }
00412
00413
00414
00415
00416
00417

```

```

00418 }
00419
00420
00421 int variable_fixing(SubProblem *currentSubProb){
00422     int num_fixed = 0;
00423     double replacement_costs [MAX_VERTEX_NUM - 2]; // for all the edges in the 1-tree except the best
replacement to maintain the 1-tree
00424     double best_candidate_replacement = INFINITE;
00425
00426     for (int i = 0; i < MAX_VERTEX_NUM - 2; i++){
00427         replacement_costs[i] = INFINITE;
00428     }
00429
00430     for (unsigned int i = 0; i < problem->graph.num_edges; i++){
00431         Edge current_edge = problem->graph.edges[i];
00432
00433         if (currentSubProb->constraints[current_edge.src][current_edge.dest] == NOTHING){
00434             if (currentSubProb->oneTree.edges_matrix[current_edge.src][current_edge.dest] == -1){
00435
00436                 double max_edge_path = 0;
00437
00438                 if (current_edge.dest != problem->candidateNodeId && current_edge.src !=
problem->candidateNodeId){
00439                     max_edge_path = max_edge_path_1Tree(currentSubProb,
replacement_costs, current_edge.src, current_edge.dest);
00440                 }
00441
00442                 else{
00443
00444                     double first_candidate_edge_weight =
problem->graph.edges_matrix[problem->candidateNodeId][currentSubProb->oneTree.nodes[problem->candidateNodeId].neighbour
00445                     double second_candidate_edge_weight =
problem->graph.edges_matrix[problem->candidateNodeId][currentSubProb->oneTree.nodes[problem->candidateNodeId].neighbour
00446
00447                     if (first_candidate_edge_weight > second_candidate_edge_weight){
00448                         max_edge_path = first_candidate_edge_weight;
00449                     } else{
00450                         max_edge_path = second_candidate_edge_weight;
00451                     }
00452                 }
00453
00454                 if (currentSubProb->oneTree.cost + current_edge.weight - max_edge_path >=
problem->bestValue){
00455                     currentSubProb->constraints[current_edge.src][current_edge.dest] = FORBIDDEN;
00456                     currentSubProb->constraints[current_edge.dest][current_edge.src] = FORBIDDEN;
00457                     currentSubProb->num_forbidden_edges++;
00458                     num_fixed++;
00459                 }
00460
00461             }
00462
00463             if (current_edge.src == problem->candidateNodeId || current_edge.dest ==
problem->candidateNodeId){
00464                 if (current_edge.weight < best_candidate_replacement){
00465                     best_candidate_replacement = current_edge.weight;
00466                 }
00467             }
00468         }
00469     }
00470
00471
00472     for (int i = 0; i < MAX_VERTEX_NUM; i++){
00473         Edge edge_1tree = currentSubProb->oneTree.edges[i];
00474         double edge_1tree_weight =
problem->graph.edges_matrix[edge_1tree.src][edge_1tree.dest].weight;
00475
00476         if (currentSubProb->constraints[edge_1tree.src][edge_1tree.dest] == NOTHING) {
00477             if (edge_1tree.src != problem->candidateNodeId && edge_1tree.dest !=
problem->candidateNodeId) {
00478                 if (currentSubProb->oneTree.cost + replacement_costs[i] - edge_1tree_weight >=
problem->bestValue) {
00479                     currentSubProb->constraints[edge_1tree.src][edge_1tree.dest] = MANDATORY;
00480                     currentSubProb->constraints[edge_1tree.dest][edge_1tree.src] = MANDATORY;
00481                     currentSubProb->mandatoryEdges[currentSubProb->num_mandatory_edges].src =
edge_1tree.src;
00482                     currentSubProb->mandatoryEdges[currentSubProb->num_mandatory_edges].dest =
edge_1tree.dest;
00483                     currentSubProb->num_mandatory_edges++;
00484                     num_fixed++;
00485                 }
00486             } else {
00487                 if (currentSubProb->oneTree.cost + best_candidate_replacement - edge_1tree_weight >=
problem->bestValue){
00488                     currentSubProb->constraints[edge_1tree.src][edge_1tree.dest] = MANDATORY;
00489                     currentSubProb->constraints[edge_1tree.dest][edge_1tree.src] = MANDATORY;
00490                     currentSubProb->mandatoryEdges[currentSubProb->num_mandatory_edges].src =
edge_1tree.src;

```

```

00491         currentSubProb->mandatoryEdges[currentSubProb->num_mandatory_edges].dest =
    edge_ltree.dest;
00492         currentSubProb->num_mandatory_edges++;
00493         num_fixed++;
00494     }
00495 }
00496 }
00497 }
00498 }
00499 return num_fixed;
00500 }
00501
00502
00503 void constrained_kruskal(Graph * graph, SubProblem * subProblem, unsigned short candidateId) {
00504     create_mst(&subProblem->oneTree, graph->nodes, graph->num_nodes);
00505     Forest forest;
00506     create_forest_constrained(&forest, graph->nodes, graph->num_nodes, candidateId);
00507     wrap_quick_sort(graph);
00508
00509     unsigned short num_edges_inMST = 0;
00510     for (unsigned short i = 0; i < subProblem->num_mandatory_edges; i++) {
00511         ConstrainedEdge current_mandatory = subProblem->mandatoryEdges[i];
00512         Edge mandatory_edge = graph->edges_matrix[current_mandatory.src][current_mandatory.dest];
00513         unsigned short src = mandatory_edge.src;
00514         unsigned short dest = mandatory_edge.dest;
00515
00516         if (src != candidateId && dest != candidateId) {
00517
00518             Set *set1_root = find(&forest.sets[src]);
00519             Set *set2_root = find(&forest.sets[dest]);
00520             if (set1_root->num_in_forest != set2_root->num_in_forest) {
00521                 merge(set1_root, set2_root);
00522                 // add the edge to the MST
00523                 add_edge(&subProblem->oneTree, &mandatory_edge);
00524                 num_edges_inMST++;
00525             }
00526         }
00527     }
00528
00529     unsigned short num_edges_inG = 0;
00530
00531     while (num_edges_inG < graph->num_edges && num_edges_inMST < graph->num_nodes - 2) {
00532
00533         Edge current_edge = graph->edges[num_edges_inG];
00534
00535         unsigned short src = current_edge.src;
00536         unsigned short dest = current_edge.dest;
00537
00538         if (src != candidateId && dest != candidateId) {
00539
00540             if (subProblem->num_forbidden_edges == 0 || subProblem->constraints[src][dest] !=
FORBIDDEN) {
00541
00542                 Set *set1_root = find(&forest.sets[src]);
00543                 Set *set2_root = find(&forest.sets[dest]);
00544
00545                 if (set1_root->num_in_forest != set2_root->num_in_forest) {
00546                     merge(set1_root, set2_root);
00547                     // add the edge to the MST
00548                     add_edge(&subProblem->oneTree, &current_edge);
00549                     num_edges_inMST++;
00550                 }
00551             }
00552         }
00553
00554         num_edges_inG++;
00555     }
00556
00557     if (num_edges_inMST == graph->num_nodes - 2) {
00558         subProblem->oneTree.isValid = true;
00559     } else {
00560         subProblem->oneTree.isValid = false;
00561     }
00562 }
00563
00564
00565 void constrained_prim(Graph * graph, SubProblem * subProblem, unsigned short candidateId) {
00566
00567     Graph graph_copy = *graph;
00568     create_mst(&subProblem->oneTree, graph->nodes, graph->num_nodes);
00569     FibonacciHeap heap;
00570     create_fibonacci_heap(&heap);
00571
00572     OrdTreeNode tree_nodes[graph->num_nodes];
00573     bool in_heap [graph->num_nodes];
00574     double values [graph->num_nodes];
00575     int fathers [graph->num_nodes];

```

```

00576
00577     bool start = true;
00578
00579     for (unsigned short i = 0; i < graph->num_nodes; i++){
00580         in_heap[i] = false;
00581         fathers[i] = -1;
00582         values[i] = start ? FLT_MAX/2 : FLT_MAX;
00583         start = false;
00584     }
00585
00586     for (unsigned short i = 0; i < subProblem->num_mandatory_edges; i++){
00587         ConstrainedEdge current_mandatory = subProblem->mandatoryEdges[i];
00588         unsigned short src = current_mandatory.src;
00589         unsigned short dest = current_mandatory.dest;
00590
00591         if(src != candidateId && dest != candidateId) {
00592
00593             graph_copy.edges_matrix[src][dest].weight = -((double) problem->graph.num_nodes);
00594             graph_copy.edges_matrix[dest][src].weight = -((double) problem->graph.num_nodes);
00595             graph_copy.edges[graph->edges_matrix[src][dest].positionInGraph].weight = -((double)
problem->graph.num_nodes);
00596         }
00597     }
00598
00599
00600     for (unsigned short i = 0; i < graph_copy.num_nodes; i++) {
00601         if(i != candidateId){
00602             create_insert_node(&heap, &tree_nodes[i], i, values[i]);
00603             in_heap[i] = true;
00604         }
00605     }
00606
00607     while(heap.num_nodes != 0){
00608         int min_pos = extract_min(&heap);
00609         if(min_pos == -1){
00610             fprintf(stderr, "Error: min_pos == -1\n");
00611             exit(1);
00612         }
00613         else{
00614             in_heap[min_pos] = false;
00615             for(unsigned short i = 0; i < graph_copy.nodes[min_pos].num_neighbours; i++){
00616                 unsigned short neigh = graph_copy.nodes[min_pos].neighbours[i];
00617                 if(neigh != min_pos && in_heap[neigh]) {
00618                     double weight = graph_copy.edges_matrix[min_pos][neigh].weight;
00619                     if (weight < tree_nodes[neigh].value){
00620                         if (subProblem->num_forbidden_edges == 0 ||
subProblem->constraints[min_pos][neigh] != FORBIDDEN){
00621                             fathers[neigh] = min_pos;
00622                             decrease_value(&heap, &tree_nodes[neigh], weight);
00623                         }
00624                     }
00625                 }
00626             }
00627         }
00628     }
00629
00630     for(unsigned short i = 0; i < graph->num_nodes; i++){
00631         if(fathers[i] != -1){
00632             add_edge(&subProblem->oneTree, &graph->edges_matrix[i][fathers[i]]);
00633         }
00634     }
00635
00636     if(subProblem->oneTree.num_edges == graph->num_nodes - 2){
00637         subProblem->oneTree.isValid = true;
00638     }
00639     else{
00640         subProblem->oneTree.isValid = false;
00641     }
00642 }
00643 }
00644
00645 // a better than b?
00646 bool compare_subproblems(const SubProblem *a, const SubProblem *b) {
00647     if (HYBRID) {
00648         return (b->value - a->value) > EPSILON ||
00649             ( (b->value > a->value) && (a->prob - b->prob) >= BETTER_PROB);
00650     } else {
00651         return (b->value - a->value) > EPSILON;
00652     }
00653 }
00654 }
00655
00656 void bound(SubProblem *currentSubProb) {
00657     if ((problem->bestSolution.value - currentSubProb->value) > EPSILON || currentSubProb->treeLevel
== 1) {

```



```

00660     currentSubProb->timeToReach = ((float) (clock() - problem->start)) / CLOCKS_PER_SEC;
00661     problem->exploredBBNodes++;
00662
00663     double pi[MAX_VERTEX_NUM] = {0};
00664     double v[MAX_VERTEX_NUM] = {0};
00665     double v_old[MAX_VERTEX_NUM] = {0};
00666     double total_pi = 0;
00667     int k = 10;
00668     int max_iter = currentSubProb->treeLevel == 1 ? (int) NUM_HK_INITIAL_ITERATIONS : (int)
NUM_HK_ITERATIONS;
00669     max_iter += k;
00670     double best_lower_bound = 0;
00671     double t_0;
00672     SubProblemsList generatedSubProblems;
00673     new_SubProblemList(&generatedSubProblems);
00674     Graph *used_graph = &problem->graph;
00675     bool first_iter = true;
00676
00677     double prob_branch [MAX_EDGES_NUM] = {0};
00678
00679     for (int iter = 1; iter <= max_iter; iter++) {
00680
00681         SubProblem analyzedSubProblem = *currentSubProb;
00682         BBNodeType type;
00683         if(!first_iter) {
00684             for (unsigned short j = 0; j < problem->graph.num_edges; j++) {
00685                 if ((pi[used_graph->edges[j].src] +
00686                     pi[used_graph->edges[j].dest]) != 0) {
00687                     used_graph->edges[j].weight += (pi[used_graph->edges[j].src] +
00688                                                         pi[used_graph->edges[j].dest]);
00689
00690                     used_graph->edges_matrix[used_graph->edges[j].src][used_graph->edges[j].dest].weight =
used_graph->edges[j].weight;
00691
00692                     used_graph->edges_matrix[used_graph->edges[j].dest][used_graph->edges[j].src].weight =
used_graph->edges[j].weight;
00693
00694                     used_graph->orderedEdges = false;
00695                 }
00696             }
00697             if (PRIM) {
00698                 constrained_prim(used_graph, &analyzedSubProblem, problem->candidateNodeId);
00699             }
00700             else {
00701                 constrained_kruskal(used_graph, &analyzedSubProblem, problem->candidateNodeId);
00702             }
00703             if (analyzedSubProblem.oneTree.isValid) {
00704                 type = mst_to_one_tree(&analyzedSubProblem, used_graph);
00705             }
00706             if (type == OPEN) {
00707                 analyzedSubProblem.prob = analyzedSubProblem.oneTree.prob;
00708                 analyzedSubProblem.type = type;
00709                 analyzedSubProblem.value = 0;
00710
00711                 for (int e = 0; e < problem->graph.num_nodes; e++) {
00712                     Edge *edge = &analyzedSubProblem.oneTree.edges[e];
00713                     analyzedSubProblem.value +=
problem->graph.edges_matrix[edge->src][edge->dest].weight;
00714
00715                     if (iter > max_iter - k) {
00716
00717                         prob_branch[problem->graph.edges_matrix[edge->src][edge->dest].positionInGraph] =
((prob_branch[problem->graph.edges_matrix[edge->src][edge->dest].positionInGraph] *
(double) k) + 1) / ((double) k);
00718                     }
00719                 }
00720
00721                 if (problem->bestValue - analyzedSubProblem.value <= EPSILON) {
00722                     analyzedSubProblem.type = CLOSED_BOUND;
00723                 } else {
00724                     analyzedSubProblem.num_edges_in_cycle = 0;
00725                     if (check_hamiltonian(&analyzedSubProblem)) {
00726                         problem->bestValue = analyzedSubProblem.value;
00727                         analyzedSubProblem.type = iter == 1 ? CLOSED_1TREE: CLOSED_SUBGRADIENT;
00728                         problem->bestSolution = analyzedSubProblem;
00729                         printf("Updated best value: %f, at time: %f\n", problem->bestValue,
((float) (clock() - problem->start)) / CLOCKS_PER_SEC);
00730                     } else {
00731                         analyzedSubProblem.type = OPEN;
00732                     }
00733                 }
00734             }
00735
00736             if(analyzedSubProblem.type != OPEN){
00737                 if(iter == 1 || (analyzedSubProblem.type == CLOSED_SUBGRADIENT)) {

```

```

00738         currentSubProb->type = analyzedSubProblem.type;
00739         return;
00740     }
00741 }
00742
00743 double current_lb = analyzedSubProblem.oneTree.cost - (2 * total_pi);
00744
00745 if (current_lb > best_lower_bound || first_iter) {
00746     best_lower_bound = current_lb;
00747     if (first_iter) {
00748         first_iter = false;
00749         used_graph = &problem->reformulationGraph;
00750     }
00751
00752     t_0 = best_lower_bound / (2 * MAX_VERTEX_NUM);
00753 }
00754
00755 // change the graph to the original one, because the dual variables are calculated
on the original graph
00756 *used_graph = problem->graph;
00757 add_elem_SubProblemList_bottom(&generatedSubProblems, &analyzedSubProblem);
00758
00759 for (unsigned short i = 0; i < problem->graph.num_nodes; i++) {
00760     v[i] = (double) (analyzedSubProblem.oneTree.nodes[i].num_neighbours - 2);
00761 }
00762
00763 double t =
00764 (max_iter - 1))) * t_0)
00765     (((double) (iter - 1)) * (((double) (2 * max_iter) - 5) / (2 * (double)
00766     - (((double) iter - 2) * t_0) +
00767     ((t_0 * ((double) iter - 1) * ((double) iter - 2)) /
00768     (2 * ((double) max_iter - 1) * ((double) max_iter - 2)))));
00769
00770 total_pi = 0;
00771
00772 for (unsigned short j = 0; j < problem->graph.num_nodes; j++) {
00773     if (v[j] != 0) {
00774         pi[j] += (double) ((0.6 * t * v[j]) + (0.4 * t * v_old[j]));
00775         v_old[j] = v[j];
00776         total_pi += pi[j];
00777     }
00778 }
00779
00780 else{
00781     currentSubProb->type = CLOSED_UNFEASIBLE;
00782     return;
00783 }
00784 } else {
00785     currentSubProb->type = CLOSED_UNFEASIBLE;
00786     return;
00787 }
00788
00789 }
00790
00791 *currentSubProb = *get_SubProblemList_elem_index(&generatedSubProblems, 0);
00792 currentSubProb->oneTree.cost = currentSubProb->value; // mi serve valore originale per
variabile fixing
00793 SubProblem best_subproblem = *currentSubProb;
00794 SubProblemsListIterator *subProblem_iterators =
create_SubProblemList_iterator(&generatedSubProblems);
00795 for (size_t j = 0; j < generatedSubProblems.size; j++) {
00796     SubProblem *generatedSubProblem = SubProblemList_iterator_get_next(subProblem_iterators);
00797     if (generatedSubProblem->value > best_subproblem.value &&
00798         generatedSubProblem->value <= best_lower_bound) {
00799         best_subproblem = *generatedSubProblem;
00800     }
00801 }
00802 currentSubProb->type = best_subproblem.type;
00803 currentSubProb->value = best_subproblem.value; // metto valore massimo trovato per migliorare
ordinamento
00804 delete_SubProblemList_iterator(subProblem_iterators);
00805 delete_SubProblemList(&generatedSubProblems);
00806 //printf("\nBest lower bound: %f, Best value: %f\n", best_lower_bound, best_subproblem.value);
00807
00808 if (currentSubProb->type == OPEN) {
00809     double best_prob_branch = -1;
00810     double best_prob = -1;
00811
00812     for (int i = 0; i < problem->graph.num_edges; i++) {
00813         Edge current_edge = problem->graph.edges[i];
00814
00815         if (currentSubProb->constraints[current_edge.src][current_edge.dest] == NOTHING) {
00816             if (best_subproblem.oneTree.edges_matrix[current_edge.src][current_edge.dest] !=
00817                 -1) {

```

```

00819             if ((fabs(prob_branch[i] - 0.5) < fabs(best_prob_branch - 0.5)) ||
00820                 (HYBRID && (fabs(prob_branch[i] - 0.5) == fabs(best_prob_branch - 0.5))
00821                     && (current_edge.prob > best_prob))) {
00822
00823                 best_prob_branch = prob_branch[i];
00824                 best_prob = current_edge.prob;
00825                 currentSubProb->edge_to_branch = i;
00826             }
00827         }
00828     }
00829 }
00830 }
00831 //printf("\nBest edge to branch: %d, with score %f and prob %f; BEST VALUE: %f\n",
currentSubProb->edge_to_branch, best_prob_branch, best_prob, problem->bestValue);
00832 }
00833 } else {
00834     currentSubProb->type = CLOSED_BOUND;
00835 }
00836 }
00837
00838
00839 bool time_limit_reached(void) {
00840     return ((clock() - problem->start) / CLOCKS_PER_SEC) > TIME_LIMIT_SECONDS;
00841 }
00842
00843
00844 void nearest_prob_neighbour(unsigned short start_node) {
00845     SubProblem nn_subProblem;
00846     nn_subProblem.num_forbidden_edges = 0;
00847     nn_subProblem.num_mandatory_edges = 0;
00848     nn_subProblem.num_edges_in_cycle = 0;
00849     nn_subProblem.timeToReach = ((float) (clock() - problem->start)) / CLOCKS_PER_SEC;
00850     create_mst(&nn_subProblem.oneTree, problem->graph.nodes, problem->graph.num_nodes);
00851     unsigned short current_node = start_node;
00852     bool visited[MAX_VERTEX_NUM] = {false};
00853     ConstrainedEdge cycleEdge;
00854
00855     for (unsigned short visited_count = 0; visited_count < problem->graph.num_nodes; visited_count++)
00856     {
00857         if (visited_count == problem->graph.num_nodes - 1) {
00858             add_edge(&nn_subProblem.oneTree, &problem->graph.edges_matrix[current_node][start_node]);
00859             cycleEdge.src = current_node;
00860             cycleEdge.dest = start_node;
00861             nn_subProblem.cycleEdges[nn_subProblem.num_edges_in_cycle] = cycleEdge;
00862             nn_subProblem.num_edges_in_cycle++;
00863         } else {
00864             double best_edge_value = INFINITE;
00865             unsigned short best_neighbour = current_node;
00866             for (unsigned short i = 0; i < problem->graph.nodes[current_node].num_neighbours; i++) {
00867
00868                 if
00869                 (problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].weight <
00870                  best_edge_value
00871                  && !visited[problem->graph.nodes[current_node].neighbours[i]]) {
00872                     best_edge_value =
00873                     problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].weight;
00874                     best_neighbour = problem->graph.nodes[current_node].neighbours[i];
00875                 }
00876             }
00877             add_edge(&nn_subProblem.oneTree,
00878 &problem->graph.edges_matrix[current_node][best_neighbour]);
00879             cycleEdge.src = current_node;
00880             cycleEdge.dest = best_neighbour;
00881             nn_subProblem.cycleEdges[nn_subProblem.num_edges_in_cycle] = cycleEdge;
00882             nn_subProblem.num_edges_in_cycle++;
00883             visited[current_node] = true;
00884             current_node = best_neighbour;
00885         }
00886     }
00887     nn_subProblem.value = nn_subProblem.oneTree.cost;
00888     nn_subProblem.oneTree.isValid = true;
00889     nn_subProblem.type = CLOSED_NN;
00890     nn_subProblem.prob = nn_subProblem.oneTree.prob;
00891
00892     if (HYBRID) {
00893         SubProblem prob_nn_subProblem;
00894         prob_nn_subProblem.num_forbidden_edges = 0;
00895         prob_nn_subProblem.num_mandatory_edges = 0;
00896         prob_nn_subProblem.num_edges_in_cycle = 0;
00897         create_mst(&prob_nn_subProblem.oneTree, problem->graph.nodes, problem->graph.num_nodes);
00898         bool prob_visited[MAX_VERTEX_NUM] = {false};
00899         current_node = start_node;
00900
00901         for (unsigned short visited_count = 0; visited_count < problem->graph.num_nodes;
00902             visited_count++) {

```

```

00900         if (visited_count == problem->graph.num_nodes - 1) {
00901             add_edge(&prob_nn_subProblem.oneTree,
&problem->graph.edges_matrix[current_node][start_node]);
00902             cycleEdge.src = current_node;
00903             cycleEdge.dest = start_node;
00904             prob_nn_subProblem.cycleEdges[prob_nn_subProblem.num_edges_in_cycle] = cycleEdge;
00905             prob_nn_subProblem.num_edges_in_cycle++;
00906         } else {
00907             double best_edge_prob = -1;
00908             unsigned short best_neighbour = current_node;
00909             for (unsigned short i = 0; i < problem->graph.nodes[current_node].num_neighbours; i++)
{
00910
00911                 if
(problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].prob >
00912                     best_edge_prob
&& !prob_visited[problem->graph.nodes[current_node].neighbours[i]]) {
00913                     best_edge_prob =
problem->graph.edges_matrix[current_node][problem->graph.nodes[current_node].neighbours[i]].prob;
00914                     best_neighbour = problem->graph.nodes[current_node].neighbours[i];
00915                 }
00916             }
00917             add_edge(&prob_nn_subProblem.oneTree,
&problem->graph.edges_matrix[current_node][best_neighbour]);
00918             cycleEdge.src = current_node;
00919             cycleEdge.dest = best_neighbour;
00920             prob_nn_subProblem.cycleEdges[prob_nn_subProblem.num_edges_in_cycle] = cycleEdge;
00921             prob_nn_subProblem.num_edges_in_cycle++;
00922             prob_visited[current_node] = true;
00923             current_node = best_neighbour;
00924         }
00925     }
00926 }
00927 prob_nn_subProblem.value = prob_nn_subProblem.oneTree.cost;
00928 prob_nn_subProblem.oneTree.isValid = true;
00929 prob_nn_subProblem.type = CLOSED_NN_HYBRID;
00930 prob_nn_subProblem.prob = prob_nn_subProblem.oneTree.prob;
00931
00932 bool better_prob = prob_nn_subProblem.value < nn_subProblem.value;
00933 SubProblem *best = better_prob ? &prob_nn_subProblem : &nn_subProblem;
00934
00935 if (best->value < problem->bestValue) {
00936     problem->bestValue = best->value;
00937     problem->bestSolution = *best;
00938 }
00939
00940 } else {
00941     if (nn_subProblem.value < problem->bestValue) {
00942         problem->bestValue = nn_subProblem.value;
00943         problem->bestSolution = nn_subProblem;
00944     }
00945 }
00946
00947 }
00948
00949 // "a" is better than "b" if its LB is higher
00950 bool compare_candidate_node(SubProblem * a, SubProblem * b){
00951     if (HYBRID) {
00952         return (a->value - b->value) > EPSILON ||
((a->value > b->value) && (a->prob - b->prob) >= BETTER_PROB);
00953     } else {
00954         return (a->value - b->value) >= EPSILON;
00955     }
00956 }
00957
00958 }
00959
00960 unsigned short find_candidate_node(void) {
00961     unsigned short best_candidate = 0;
00962     SubProblem best_subProblem;
00963     best_subProblem.value = 0;
00964     best_subProblem.prob = 0;
00965
00966     for (unsigned short i = 0; i < problem->graph.num_nodes; i++) {
00967         SubProblem currentCandidate;
00968         problem->candidateNodeId = i;
00969         currentCandidate.num_forbidden_edges = 0;
00970         currentCandidate.num_mandatory_edges = 0;
00971
00972         nearest_prob_neighbour(i);
00973
00974         if (PRIM){
00975             constrained_prim(&problem->graph, &currentCandidate, i);
00976         }
00977         else{
00978             constrained_kruskal(&problem->graph, &currentCandidate, i);
00979         }
00980     }
00981 }

```

```

00982     mst_to_one_tree(&currentCandidate, &problem->graph);
00983     currentCandidate.value = currentCandidate.oneTree.cost;
00984     currentCandidate.prob = currentCandidate.oneTree.prob;
00985
00986     if (compare_candidate_node(&currentCandidate, &best_subProblem)) {
00987         best_candidate = i;
00988         best_subProblem.value = currentCandidate.value;
00989         best_subProblem.prob = currentCandidate.prob;
00990     }
00991 }
00992 }
00993 return best_candidate;
00994 }
00995
00996
00997 bool check_feasibility(Graph *graph) {
00998
00999     bool feasible = true;
01000     for (short i = 0; feasible && i < graph->num_nodes; i++) {
01001         Node current_node = graph->nodes[i];
01002         if (current_node.num_neighbours < 2) {
01003             feasible = false;
01004             printf("\nThe graph is not feasible for the BB algorithm. Node %i has less than 2
neighbors.",
01005                 current_node.positionInGraph);
01006         }
01007     }
01008     return feasible;
01009 }
01010
01011
01012 void branch_and_bound(Problem *current_problem) {
01013
01014     problem = current_problem;
01015
01016     if (check_feasibility(&problem->graph)) {
01017
01018         problem->start = clock();
01019         problem->bestValue = INFINITE;
01020         problem->candidateNodeId = find_candidate_node();
01021         problem->bestSolution.id = 0;
01022         problem->bestSolution.treeLevel = 0;
01023         problem->bestSolution.fatherId = -1;
01024         problem->exploredBBNodes = 0;
01025         problem->generatedBBNodes = 0;
01026         problem->totTreeLevels = 1;
01027         problem->interrupted = false;
01028         problem->reformulationGraph = problem->graph;
01029         problem->generatedBBNodes++;
01030         problem->num_fixed_edges = 0;
01031
01032         SubProblem subProblem;
01033         subProblem.treeLevel = 1;
01034         subProblem.id = problem->generatedBBNodes;
01035         subProblem.fatherId = 0;
01036         subProblem.type = OPEN;
01037         subProblem.prob = 0;
01038         subProblem.value = INFINITE;
01039         subProblem.num_edges_in_cycle = 0;
01040         subProblem.edge_to_branch = -1;
01041         initialize_matrix(&subProblem);
01042
01043         SubProblemsList subProblems;
01044         new_SubProblemList(&subProblems);
01045         add_elem_SubProblemList_bottom(&subProblems, &subProblem);
01046
01047         while (subProblems.size != 0 && !time_limit_reached()) {
01048             SubProblem current_sub_problem = *get_SubProblemList_elem_index(&subProblems, 0);
01049             delete_SubProblemList_elem_index(&subProblems, 0);
01050             bound(&current_sub_problem);
01051             if (current_sub_problem.type == OPEN && current_sub_problem.edge_to_branch != -1) {
01052                 problem->num_fixed_edges += variable_fixing(&current_sub_problem);
01053                 branch(&subProblems, &current_sub_problem);
01054             }
01055         }
01056
01057         if (time_limit_reached()) {
01058             problem->interrupted = true;
01059         }
01060
01061         problem->end = clock();
01062         delete_SubProblemList(&subProblems);
01063     }
01064 }
01065 }
01066
01067

```

```

01068 void set_problem(Problem *current_problem) {
01069     problem = current_problem;
01070 }
01071
01072
01073 void print_subProblem(const SubProblem *subProblem) {
01074     char *type;
01075     if (subProblem->type == OPEN) {
01076         type = "OPEN";
01077     } else if (subProblem->type == CLOSED_UNFEASIBLE) {
01078         type = "CLOSED_UNFEASIBLE";
01079     } else if (subProblem->type == CLOSED_BOUND) {
01080         type = "CLOSED_BOUND";
01081     } else if (subProblem->type == CLOSED_NN) {
01082         type = "CLOSED_NEAREST_NEIGHBOR";
01083     } else if (subProblem->type == CLOSED_NN_HYBRID) {
01084         type = "CLOSED_NEAREST_NEIGHBOR_HYBRID";
01085     } else {
01086         type = "CLOSED_SUBGRADIENT";
01087     }
01088     printf("\nSUBPROBLEM with cost = %lf, type = %s, level of the BB tree = %i, prob_tour = %lf,
BBNode number = %u and time to obtain = %lfs, edge to branch = %i\n",
01090         subProblem->value, type, subProblem->treeLevel, subProblem->prob, subProblem->id,
01091         subProblem->timeToReach, subProblem->edge_to_branch);
01092
01093     //print_mst_original_weight(&subProblem->oneTree, &problem->graph);
01094
01095     printf("\nCycle with %i edges:", subProblem->num_edges_in_cycle);
01096     unsigned short last_dest = 0;
01097     for (unsigned short i = 0; i < subProblem->num_edges_in_cycle; i++) {
01098         ConstrainedEdge edge_cycle = subProblem->cycleEdges[i];
01099         unsigned short src = edge_cycle.src;
01100         unsigned short dest = edge_cycle.dest;
01101
01102         if (i == 0) {
01103             if (src == subProblem->cycleEdges[subProblem->num_edges_in_cycle - 1].dest ||
01104                 src == subProblem->cycleEdges[subProblem->num_edges_in_cycle - 1].src) {
01105                 printf(" %i <-> %i,", src, dest);
01106                 last_dest = dest;
01107             }
01108             else {
01109                 printf(" %i <-> %i,", dest, src);
01110                 last_dest = src;
01111             }
01112         } else {
01113             if (src == last_dest) {
01114                 if (i == subProblem->num_edges_in_cycle - 1) {
01115                     printf(" %i <-> %i", src, dest);
01116                 } else {
01117                     printf(" %i <-> %i,", src, dest);
01118                 }
01119                 last_dest = dest;
01120             }
01121             else {
01122                 if (i == subProblem->num_edges_in_cycle - 1) {
01123                     printf(" %i <-> %i", dest, src);
01124                 } else {
01125                     printf(" %i <-> %i,", dest, src);
01126                 }
01127                 last_dest = src;
01128             }
01129         }
01130     }
01131 }
01132
01133 printf("\n%i Mandatory edges:", subProblem->num_mandatory_edges);
01134 printf("\n%i Forbidden edges:", subProblem->num_forbidden_edges);
01135 printf("\n");
01136 }
01137
01138 void print_problem(void) {
01139     printf("Optimal tour found with candidate node = %i, elapsed time = %lfs and interrupted = %s\n",
01140         problem->candidateNodeId, ((double) (problem->end - problem->start)) / CLOCKS_PER_SEC,
01141         problem->interrupted ? "TRUE" : "FALSE");
01142
01143     printf("\nMST solved with: %s algorithm\n", PRIM ? "Prim" : "Kruskal");
01144
01145     printf("\nGHOSH_UB = %lf, SUBPROBLEM_UB = %lf\n", GHOSH_UB, EPSILON);
01146
01147     printf("\nB-&-B tree with generated BBNodes = %u, explored BBNodes = %u and max tree level =
01148         %u\n",
01149         problem->generatedBBNodes, problem->exploredBBNodes, problem->totTreeLevels);
01150
01151     printf("\nNumber of fixed edges = %i\n", problem->num_fixed_edges);
01152     print_subProblem(&problem->bestSolution);

```

```
01153 }
```

### 9.3 GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/branch\_and\_bound.h File Reference

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

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

#### Functions

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

## Variables

- static `Problem * problem`

*The pointer to the problem to solve.*

### 9.3.1 Detailed Description

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

#### Author

Lorenzo Sciandra

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

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [branch\\_and\\_bound.h](#).

### 9.3.2 Function Documentation

#### 9.3.2.1 bound()

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

The Held-Karp bound function with the subgradient algorithm.

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

#### Parameters

<code>current_problem</code>	The pointer to the <code>SubProblem</code> or branch-and-bound <code>Node</code> in the tree.
------------------------------	---

Definition at line 657 of file [branch\\_and\\_bound.c](#).



### 9.3.2.2 branch()

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

The Shutler's branching rule.

Every [SubProblem](#) is branched into 2 new SubProblems, one including the "edge\_to\_branch" and the other not. More details at <http://www.jstor.org/stable/254144>.

#### Parameters

<i>openSubProblems</i>	The list of open SubProblems, to which the new SubProblems will be added.
<i>subProblem</i>	The <a href="#">SubProblem</a> to branch.

Definition at line 256 of file [branch\\_and\\_bound.c](#).

### 9.3.2.3 branch\_and\_bound()

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

The Branch and Bound algorithm.

This is the main function of the Branch and Bound algorithm. It stores all the open SubProblems in a [SubProblemsList](#) and analyzes them one by one with the [branch\(\)](#) and [held\\_karp\\_bound\(\)](#) functions.

#### Parameters

<i>current_problem</i>	The pointer to the problem to solve.
------------------------	--------------------------------------

Definition at line 1012 of file [branch\\_and\\_bound.c](#).

### 9.3.2.4 check\_feasibility()

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

Check if the [Graph](#) associated to the [Problem](#) is feasible.

A [Graph](#) is feasible if every [Node](#) has at least degree 2.

**Parameters**

<i>graph</i>	The <a href="#">Graph</a> to check.
--------------	-------------------------------------

**Returns**

true if the [Graph](#) is feasible, false otherwise.

Definition at line 997 of file [branch\\_and\\_bound.c](#).

**9.3.2.5 check\_hamiltonian()**

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

Function that checks if the 1Tree of a [SubProblem](#) is a tour.

This is done by simply check if all the edges are in the cycle passing through the candidate [Node](#).

**Parameters**

<i>subProblem</i>	The <a href="#">SubProblem</a> to check.
-------------------	--

**Returns**

true if the [SubProblem](#) is a Hamiltonian cycle, false otherwise.

Definition at line 82 of file [branch\\_and\\_bound.c](#).

**9.3.2.6 clean\_matrix()**

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

Clean the matrix of constraints of a [SubProblem](#).

This function is used to initialize the matrix of [ConstraintType](#) for a [SubProblem](#).

**Parameters**

<i>subProblem</i>	The <a href="#">SubProblem</a> with no <a href="#">ConstraintType</a> .
-------------------	---

### 9.3.2.7 compare\_subproblems()

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

Compare two OPEN SubProblems.

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

#### Parameters

<i>a</i>	The first <a href="#">SubProblem</a> to compare.
<i>b</i>	The second <a href="#">SubProblem</a> to compare.

#### Returns

true if the first [SubProblem](#) is better than the second, false otherwise.

Definition at line 647 of file [branch\\_and\\_bound.c](#).

### 9.3.2.8 copy\_constraints()

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

Copy the matrix of constraints of a [SubProblem](#) into another.

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

#### Parameters

<i>subProblem</i>	The <a href="#">SubProblem</a> to which the ConstraintType will be copied.
<i>otherSubProblem</i>	The <a href="#">SubProblem</a> from which the ConstraintType will be copied.

Definition at line 234 of file [branch\\_and\\_bound.c](#).

### 9.3.2.9 dfs()

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

A Depth First Search algorithm on a [Graph](#).

This function is used to find the cycle in the 1Tree [SubProblem](#), passing through the candidate [Node](#).

**Parameters**

<i>subProblem</i>	The <a href="#">SubProblem</a> to inspect.
-------------------	--

Definition at line 18 of file [branch\\_and\\_bound.c](#).

**9.3.2.10 find\_candidate\_node()**

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

Select the candidate [Node](#), i.e. the starting vertex of the tour.

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

**Returns**

the candidate [Node](#) id.

Definition at line 961 of file [branch\\_and\\_bound.c](#).

**9.3.2.11 infer\_constraints()**

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

Infer the values of some edge variables of a [SubProblem](#).

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

**Parameters**

<i>subProblem</i>	The <a href="#">SubProblem</a> to which we want to infer the variables values.
-------------------	--

**Returns**

true if the subproblem remains feasible, false otherwise.

Definition at line 185 of file [branch\\_and\\_bound.c](#).

### 9.3.2.12 mst\_to\_one\_tree()

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

Transforms a [MST](#) into a 1Tree.

This is done by adding the two least-cost edges incident to the candidate [Node](#) in the [MST](#).

#### Parameters

<i>currentSubproblem</i>	The <a href="#">SubProblem</a> to which the <a href="#">MST</a> belongs.
<i>graph</i>	The <a href="#">Graph</a> of the <a href="#">Problem</a> .

#### Returns

an enum value that indicates if the [SubProblem](#) is feasible or not.

Definition at line [88](#) of file [branch\\_and\\_bound.c](#).

### 9.3.2.13 nearest\_prob\_neighbour()

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

This function is used to find the first feasible tour.

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

#### Parameters

<i>start_node</i>	The <a href="#">Node</a> from which the tour will start.
-------------------	--

Definition at line [844](#) of file [branch\\_and\\_bound.c](#).

### 9.3.2.14 print\_problem()

```
void print_problem (
    void )
```

Get all metrics of the problem.

It is used at the end of the algorithm to print the solution obtained. It calls the [print\\_subProblem\(\)](#) function on the best [SubProblem](#) found.

Definition at line [1138](#) of file [branch\\_and\\_bound.c](#).

#### 9.3.2.15 [print\\_subProblem\(\)](#)

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

Get all metrics of a certain [SubProblem](#).

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

##### Parameters

<i>subProblem</i>	The <a href="#">SubProblem</a> to print.
-------------------	--

Definition at line [1073](#) of file [branch\\_and\\_bound.c](#).

#### 9.3.2.16 [set\\_problem\(\)](#)

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

Define the problem to solve.

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

##### Parameters

<i>current_problem</i>	The pointer to the problem to solve.
------------------------	--------------------------------------

Definition at line [1068](#) of file [branch\\_and\\_bound.c](#).

#### 9.3.2.17 [time\\_limit\\_reached\(\)](#)

```
bool time_limit_reached (
    void )
```

Check if the time limit has been reached.

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

#### Returns

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

Definition at line 839 of file [branch\\_and\\_bound.c](#).

#### 9.3.2.18 variable\_fixing()

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

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

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

#### Parameters

<i>subProblem</i>	The <a href="#">SubProblem</a> that we want to add the constraints to.
-------------------	--

#### Returns

the num of variables fixed.

Definition at line 421 of file [branch\\_and\\_bound.c](#).

### 9.3.3 Variable Documentation

#### 9.3.3.1 problem

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

The pointer to the problem to solve.

Definition at line 22 of file [branch\\_and\\_bound.h](#).

## 9.4 branch\_and\_bound.h

[Go to the documentation of this file.](#)

```
00001
00014 #ifndef BRANCHANDBOUND1TREE_BRANCH_AND_BOUND_H
00015 #define BRANCHANDBOUND1TREE_BRANCH_AND_BOUND_H
00016 #include "kruskal.h"
00017 #include "prim.h"
00018 #include "../data_structures/b_and_b_data.h"
00019
00020
00022 static Problem * problem;
00023
00024
00026
00030 void dfs(SubProblem *subProblem);
00031
00032
00034
00040 bool check_hamiltonian(SubProblem *subProblem);
00041
00042
00044
00050 BNodeType mst_to_one_tree(SubProblem *currentSubproblem, Graph *graph);
00051
00052
00054
00058 void clean_matrix(SubProblem *subProblem);
00059
00060
00062
00068 void copy_constraints(SubProblem *subProblem, const SubProblem *otherSubProblem);
00069
00070
00072
00078 bool compare_subproblems(const SubProblem *a, const SubProblem *b);
00079
00080
00082
00088 void branch(SubProblemsList *openSubProblems, SubProblem *subProblem);
00089
00090
00092
00098 int variable_fixing (SubProblem * subProblem);
00099
00100
00102
00108 bool infer_constraints(SubProblem * subProblem);
00109
00110
00112
00117 void bound(SubProblem *currentSubProb);
00118
00119
00121
00125 bool time_limit_reached(void);
00126
00127
00129
00135 void nearest_prob_neighbour(unsigned short start_node);
00136
00137
00139
00144 unsigned short find_candidate_node(void);
00145
00146
00148
00153 void branch_and_bound(Problem * current_problem);
00154
00155
00157
00162 bool check_feasibility(Graph * graph);
00163
00164
00166
00170 void set_problem(Problem * current_problem);
00171
00172
00174
00178 void print_subProblem(const SubProblem *subProblem);
00179
00180
00182
00185 void print_problem(void);
00186
```



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

## 9.5 GraphConvolutionalBranchandBound/src/Hybrid Solver/main/algorithms/kruskal.c File Reference

The implementaion of the functions needed to compute the [MST](#) with Kruskal's algorithm.

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

### Functions

- static void [swap](#) ([Graph](#) \*graph, unsigned short swap\_1, unsigned short swap\_2)
- static int [pivot\\_quicksort](#) ([Graph](#) \*graph, unsigned short first, unsigned short last)
- static void [quick\\_sort](#) ([Graph](#) \*graph, unsigned short first, unsigned short last)
- void [wrap\\_quick\\_sort](#) ([Graph](#) \*graph)
  - The wrapper of the quick sort algorithm.*
- void [kruskal](#) ([Graph](#) \*graph, [MST](#) \*mst)
  - The Kruskal algorithm to find the Minimum Spanning Tree  $O(|E| \log |V|)$*

### 9.5.1 Detailed Description

The implementaion of the functions needed to compute the [MST](#) with Kruskal's algorithm.

#### Author

Lorenzo Sciandra

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

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [kruskal.c](#).

### 9.5.2 Function Documentation

#### 9.5.2.1 kruskal()

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

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

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

## Parameters

<i>graph</i>	The <a href="#">Graph</a> from which we want to find the <a href="#">MST</a> .
<i>mst</i>	The Minimum Spanning Tree.

Definition at line 131 of file [kruskal.c](#).

### 9.5.2.2 pivot\_quicksort()

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

Definition at line 39 of file [kruskal.c](#).

### 9.5.2.3 quick\_sort()

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

Definition at line 110 of file [kruskal.c](#).

### 9.5.2.4 swap()

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

Definition at line 18 of file [kruskal.c](#).

### 9.5.2.5 wrap\_quick\_sort()

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

The wrapper of the quick sort algorithm.

If the [Graph](#) is not sorted, this function calls the quick sort algorithm to sort the edges of the [Graph](#).

## Parameters

<code>graph</code>	The <a href="#">Graph</a> to which we want to sort the edges.
--------------------	---

Definition at line 123 of file [kruskal.c](#).

## 9.6 kruskal.c

[Go to the documentation of this file.](#)

```

00001
00015 #include "kruskal.h"
00016
00017
00018 static void swap(Graph *graph, unsigned short swap_1, unsigned short swap_2) {
00019
00020     Edge * edges = graph->edges;
00021
00022     //printf("\nswap values %lf - %lf, at %d - %d\n", edges[swap_1].weight, edges[swap_2].weight,
00023     swap_1, swap_2);
00024
00025     graph->edges_matrix[edges[swap_1].src][edges[swap_1].dest].positionInGraph = swap_2;
00026     graph->edges_matrix[edges[swap_2].src][edges[swap_2].dest].positionInGraph = swap_1;
00027
00028     graph->edges_matrix[edges[swap_1].dest][edges[swap_1].src].positionInGraph = swap_2;
00029     graph->edges_matrix[edges[swap_2].dest][edges[swap_2].src].positionInGraph = swap_1;
00030
00031     edges[swap_1].positionInGraph = swap_2;
00032     edges[swap_2].positionInGraph = swap_1;
00033     Edge temp = edges[swap_1];
00034     edges[swap_1] = edges[swap_2];
00035     edges[swap_2] = temp;
00036 }
00037
00038
00039 static int pivot_quicksort(Graph * graph, unsigned short first, unsigned short last) {
00040     Edge * edges = graph->edges;
00041     Edge last_edge = edges[last];
00042     Edge first_edge = edges[first];
00043     unsigned short middle = (first + last) / 2;
00044     Edge middle_edge = edges[middle];
00045     double pivot_weight = first_edge.weight;
00046     double pivot_prob = first_edge.prob;
00047
00048     if (last_edge.weight > first_edge.weight ||
00049         (HYBRID && last_edge.weight == first_edge.weight && last_edge.prob < first_edge.prob)){
00050         if ((last_edge.weight > middle_edge.weight) ||
00051             (HYBRID && last_edge.weight == middle_edge.weight && last_edge.prob <
00052             middle_edge.prob)) {
00053             if ((middle_edge.weight > first_edge.weight) ||
00054                 (HYBRID && middle_edge.weight == first_edge.weight && middle_edge.prob <
00055                 first_edge.prob)){
00056                 pivot_weight = middle_edge.weight;
00057                 pivot_prob = middle_edge.prob;
00058                 swap(graph, first, middle);
00059             } else {
00060                 pivot_weight = last_edge.weight;
00061                 pivot_prob = last_edge.prob;
00062                 swap(graph, first, last);
00063             }
00064         } else {
00065             if (last_edge.weight > middle_edge.weight ||
00066                 (HYBRID && last_edge.weight == middle_edge.weight && last_edge.prob <
00067                 middle_edge.prob)) {
00068                 pivot_weight = last_edge.weight;
00069                 pivot_prob = last_edge.prob;
00070                 swap(graph, first, last);
00071             } else if ((first_edge.weight > middle_edge.weight) ||
00072                 (HYBRID && first_edge.weight == middle_edge.weight && first_edge.prob <
00073                 middle_edge.prob)) {
00074                 pivot_weight = middle_edge.weight;
00075                 pivot_prob = middle_edge.prob;
00076                 swap(graph, first, middle);
00077             }
00078         }
00079     }
00080 }

```

```

00077     unsigned short j = last;
00078     unsigned short i = first + 1;
00079     bool condition = true;
00080     while (condition) {
00081         Edge i_edge = edges[i];
00082         while (i <= j && ((!HYBRID && pivot_weight >= i_edge.weight) ||
00083             (HYBRID && (pivot_weight > i_edge.weight || (pivot_weight == i_edge.weight
&& pivot_prob <= i_edge.prob))))){
00084             i += 1;
00085             i_edge = edges[i];
00086         }
00087         Edge j_edge = edges[j];
00088         while (i <= j && (j_edge.weight > pivot_weight
|| (HYBRID && (j_edge.weight == pivot_weight && j_edge.prob < pivot_prob))))
00089         {
00090             j -= 1;
00091             j_edge = edges[j];
00092         }
00093         if (i <= j) {
00094             swap(graph, i, j);
00095         } else {
00096             condition = false;
00097         }
00098     }
00099 }
00100
00101 if(j != first){
00102     swap( graph, first, j);
00103 }
00104
00105 return j;
00106 }
00107 }
00108
00109
00110 static void quick_sort(Graph * graph, unsigned short first, unsigned short last) {
00111     if (first < last) {
00112         unsigned short pivot = pivot_quicksort(graph, first, last);
00113         if(pivot - 1 > first) {
00114             quick_sort(graph, first, pivot - 1);
00115         }
00116         if(pivot + 1 < last) {
00117             quick_sort(graph, pivot + 1, last);
00118         }
00119     }
00120 }
00121
00122
00123 void wrap_quick_sort(Graph * graph) {
00124     if (!graph->orderedEdges) {
00125         graph->orderedEdges = true;
00126         quick_sort(graph, 0, graph->num_edges - 1);
00127     }
00128 }
00129
00130
00131 void kruskal(Graph * graph, MST * mst) {
00132     create_mst(mst, graph->nodes, graph->num_nodes);
00133     Forest forest;
00134     create_forest(&forest, graph->nodes, graph->num_nodes);
00135     wrap_quick_sort(graph);
00136     unsigned short num_edges_inG = 0;
00137     unsigned short num_edges_inMST = 0;
00138
00139     while (num_edges_inG < graph->num_edges && num_edges_inMST < graph->num_nodes - 1) {
00140         // get the edge with the minimum weight
00141         Edge current_edge = graph->edges[num_edges_inG];
00142         unsigned short src = current_edge.src;
00143         unsigned short dest = current_edge.dest;
00144
00145         Set *set1_root = find(&forest.sets[src]);
00146         Set *set2_root = find(&forest.sets[dest]);
00147
00148         if (set1_root->num_in_forest != set2_root->num_in_forest) {
00149             merge(set1_root, set2_root);
00150             // add the edge to the MST
00151             add_edge(mst, &current_edge);
00152             num_edges_inMST++;
00153         }
00154         num_edges_inG++;
00155     }
00156     if (num_edges_inMST == graph->num_nodes - 1) {
00157         mst->isValid = true;
00158     }
00159 }

```

## 9.7 GraphConvolutionalBranchandBound/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)  
*Swaps two edges in the list of edges in the [Graph](#).*
- static int [pivot\\_quicksort](#) ([Graph](#) \*graph, unsigned short first, unsigned short last)  
*The core of the quick sort algorithm.*
- static void [quick\\_sort](#) ([Graph](#) \*graph, unsigned short first, unsigned short last)  
*The quick sort algorithm  $O(n \log n)$ .*
- void [wrap\\_quick\\_sort](#) ([Graph](#) \*graph)  
*The wrapper of the quick sort algorithm.*
- void [kruskal](#) ([Graph](#) \*graph, [MST](#) \*mst)  
*The Kruskal algorithm to find the Minimum Spanning Tree  $O(|E| \log |V|)$*

### 9.7.1 Detailed Description

The declaration of the functions needed to compute the [MST](#) with Kruskal's algorithm.

#### Author

Lorenzo Sciandra

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

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [kruskal.h](#).

### 9.7.2 Function Documentation

#### 9.7.2.1 kruskal()

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

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

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

## Parameters

<i>graph</i>	The <a href="#">Graph</a> from which we want to find the <a href="#">MST</a> .
<i>mst</i>	The Minimum Spanning Tree.

Definition at line 131 of file [kruskal.c](#).

### 9.7.2.2 pivot\_quicksort()

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

The core of the quick sort algorithm.

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

## Parameters

<i>graph</i>	The <a href="#">Graph</a> to which we want to sort the edges.
<i>first</i>	The index of the first <a href="#">Edge</a> to consider in the list of edges.
<i>last</i>	The index of the last <a href="#">Edge</a> to consider in the list of edges.

## Returns

the index of the pivot.

### 9.7.2.3 quick\_sort()

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

The quick sort algorithm  $O(n \log n)$ .

It is used to sort the edges of the [Graph](#) in ascending order in  $O(n \log n)$ . It is recursive.

## Parameters

<i>graph</i>	The <a href="#">Graph</a> to which we want to sort the edges.
<i>first</i>	The index of the first <a href="#">Edge</a> to consider in the list of edges.
<i>last</i>	The index of the last <a href="#">Edge</a> to consider in the list of edges.

### 9.7.2.4 swap()

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

Swaps two edges in the list of edges in the [Graph](#).

This function is used to swap two edges in the list of edges in the [Graph](#).

#### Parameters

<i>graph</i>	The <a href="#">Graph</a> to which the edges belong.
<i>swap_1</i>	The index of the first <a href="#">Edge</a> to swap.
<i>swap_2</i>	The index of the second <a href="#">Edge</a> to swap.

### 9.7.2.5 wrap\_quick\_sort()

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

The wrapper of the quick sort algorithm.

If the [Graph](#) is not sorted, this function calls the quick sort algorithm to sort the edges of the [Graph](#).

#### Parameters

<i>graph</i>	The <a href="#">Graph</a> to which we want to sort the edges.
--------------	---

Definition at line 123 of file [kruskal.c](#).

## 9.8 kruskal.h

[Go to the documentation of this file.](#)

```
00001
00015 #ifndef BRANCHANDBOUNDITREE_KRUSKAL_H
00016 #define BRANCHANDBOUNDITREE_KRUSKAL_H
00017 #include "../data_structures/mst.h"
00018
00019
00021
00027 static void swap(Graph * graph, unsigned short swap_1, unsigned short swap_2);
00028
00029
00031
```

```

00039 static int pivot_quicksort(Graph * graph, unsigned short first, unsigned short last);
00040
00041
00043
00049 static void quick_sort(Graph * graph, unsigned short first, unsigned short last);
00050
00051
00053
00057 void wrap_quick_sort(Graph * graph);
00058
00059
00061
00066 void kruskal(Graph * graph, MST * mst);
00067
00068
00069 #endif //BRANCHANDBOUND1TREE_KRUSKAL_H

```

## 9.9 GraphConvolutionalBranchandBound/src/Hybrid↩ Solver/main/algorithms/prim.c File Reference

The implementaion of the functions needed to compute the [MST](#) with Prim's algorithm.

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

### Functions

- void [prim](#) (const [Graph](#) \*graph, [MST](#) \*mst)  
*The Prim algorithm to find the Minimum Spanning Tree  $O(|E| + |V| \log |V|)$*

#### 9.9.1 Detailed Description

The implementaion of the functions needed to compute the [MST](#) with Prim's algorithm.

#### Author

Lorenzo Sciandra

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

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [prim.c](#).



## 9.9.2 Function Documentation

### 9.9.2.1 prim()

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

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

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

#### Parameters

<i>graph</i>	The <a href="#">Graph</a> from which we want to find the <a href="#">MST</a> .
<i>mst</i>	The Minimum Spanning Tree.

Definition at line 16 of file [prim.c](#).

## 9.10 prim.c

[Go to the documentation of this file.](#)

```
00001
00014 #include "prim.h"
00015
00016 void prim(const Graph * graph, MST * mst){
00017     create_mst(mst, graph->nodes, graph->num_nodes);
00018     FibonacciHeap heap;
00019     create_fibonacci_heap(&heap);
00020
00021     OrdTreeNode tree_nodes[graph->num_nodes];
00022     bool in_heap [graph->num_nodes];
00023     int fathers [graph->num_nodes];
00024     bool start = true;
00025
00026     for (unsigned short i = 0; i < graph->num_nodes; i++) {
00027         if(start){
00028             create_insert_node(&heap, &tree_nodes[i], i, 0);
00029             start = false;
00030         } else{
00031             create_insert_node(&heap, &tree_nodes[i], i, DBL_MAX);
00032         }
00033         in_heap[i] = true;
00034         fathers[i] = -1;
00035     }
00036
00037     while(heap.num_nodes != 0){
00038         int min_pos = extract_min(&heap);
00039         if(min_pos == -1){
00040             fprintf(stderr, "Error: min_pos == -1\n");
00041             exit(1);
00042         }
00043         else{
00044             in_heap[min_pos] = false;
00045             for(unsigned short i = 0; i < graph->nodes[min_pos].num_neighbours; i++){
00046                 unsigned short neigh = graph->nodes[min_pos].neighbours[i];
00047                 if(neigh != min_pos && in_heap[neigh]) {
00048                     double weight = graph->edges_matrix[min_pos][neigh].weight;
00049                     if (weight < tree_nodes[neigh].value) {
00050                         fathers[neigh] = min_pos;
00051                         decrease_value(&heap, &tree_nodes[neigh], weight);
00052                     }
00053                 }
00054             }
00055         }
00056     }
00057 }
```

```

00053         }
00054     }
00055 }
00056 }
00057
00058 for(unsigned short i = 0; i < graph->num_nodes; i++){
00059     if(fathers[i] != -1){
00060         add_edge(mst, &graph->edges_matrix[i][fathers[i]]);
00061     }
00062 }
00063
00064 if(mst->num_edges == graph->num_nodes - 1){
00065     mst->isValid = true;
00066 }
00067 else{
00068     mst->isValid = false;
00069 }
00070 }

```

## 9.11 GraphConvolutionalBranchandBound/src/Hybrid↩ Solver/main/algorithms/prim.h File Reference

The declaration of the functions needed to compute the [MST](#) with Prim's algorithm.

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

### Functions

- void [prim](#) (const [Graph](#) \*graph, [MST](#) \*mst)

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

#### 9.11.1 Detailed Description

The declaration of the functions needed to compute the [MST](#) with Prim's algorithm.

#### Author

Lorenzo Sciandra

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

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [prim.h](#).

## 9.11.2 Function Documentation

### 9.11.2.1 prim()

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

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

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

#### Parameters

<i>graph</i>	The <a href="#">Graph</a> from which we want to find the <a href="#">MST</a> .
<i>mst</i>	The Minimum Spanning Tree.

Definition at line 16 of file [prim.c](#).

## 9.12 prim.h

[Go to the documentation of this file.](#)

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

## 9.13 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/b\_and\_b\_data.c File Reference

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

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

### Functions

- void [new\\_SubProblemList](#) (SubProblemsList \*list)  
Create a new [SubProblem List](#).
- void [delete\\_SubProblemList](#) (SubProblemsList \*list)  
Delete a [SubProblem List](#).

- bool `is_SubProblemList_empty` (`SubProblemsList *list`)  
*Check if a `SubProblem List` is empty.*
- `SubProblemElem *` `build_list_elem` (`SubProblem *value`, `SubProblemElem *next`, `SubProblemElem *prev`)
- `size_t` `get_SubProblemList_size` (`SubProblemsList *list`)  
*Get the size of a `SubProblem List`.*
- void `add_elem_SubProblemList_bottom` (`SubProblemsList *list`, `SubProblem *element`)  
*Add a `SubProblem` to the bottom of a `SubProblem List`.*
- void `add_elem_SubProblemList_index` (`SubProblemsList *list`, `SubProblem *element`, `size_t index`)  
*Add a `SubProblem` at a specific index of a `SubProblem List`.*
- void `delete_SubProblemList_elem_index` (`SubProblemsList *list`, `size_t index`)  
*Remove a `SubProblem` from a specific index of a `SubProblem List`.*
- `SubProblem *` `get_SubProblemList_elem_index` (`SubProblemsList *list`, `size_t index`)  
*Get a `SubProblem` from a specific index of a `SubProblem List`.*
- `SubProblemsListIterator *` `create_SubProblemList_iterator` (`SubProblemsList *list`)  
*Create a new `SubProblem List` iterator on a `SubProblem List`.*
- bool `is_SubProblemList_iterator_valid` (`SubProblemsListIterator *iterator`)  
*Check if a `SubProblem List` iterator is valid.*
- `SubProblem *` `get_current_openSubProblemList_iterator_element` (`SubProblemsListIterator *iterator`)
- void `list_openSubProblemList_next` (`SubProblemsListIterator *iterator`)
- `SubProblem *` `SubProblemList_iterator_get_next` (`SubProblemsListIterator *iterator`)  
*Get the next element of a `SubProblem List` iterator.*
- void `delete_SubProblemList_iterator` (`SubProblemsListIterator *iterator`)  
*Delete a `SubProblem List` iterator.*

### 9.13.1 Detailed Description

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

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file `b_and_b_data.c`.

### 9.13.2 Function Documentation

#### 9.13.2.1 `add_elem_SubProblemList_bottom()`

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

Add a `SubProblem` to the bottom of a `SubProblem List`.

## Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to modify.
<i>element</i>	The <a href="#">SubProblem</a> to add.

Definition at line 59 of file [b\\_and\\_b\\_data.c](#).

9.13.2.2 `add_elem_SubProblemList_index()`

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

Add a [SubProblem](#) at a specific index of a [SubProblem List](#).

## Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to modify.
<i>element</i>	The <a href="#">SubProblem</a> to add.
<i>index</i>	The index where to add the <a href="#">SubProblem</a> .

`list` is clearer way but it is already checked inside `get_list_size`

Definition at line 75 of file [b\\_and\\_b\\_data.c](#).

9.13.2.3 `build_list_elem()`

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

Definition at line 44 of file [b\\_and\\_b\\_data.c](#).

9.13.2.4 `create_SubProblemList_iterator()`

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

Create a new [SubProblem List](#) iterator on a [SubProblem List](#).

**Parameters**

<i>list</i>	The <a href="#">SubProblem List</a> to iterate.
-------------	---

**Returns**

the [SubProblem List](#) iterator.

Definition at line 159 of file [b\\_and\\_b\\_data.c](#).

**9.13.2.5 delete\_SubProblemList()**

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

Delete a [SubProblem List](#).

**Parameters**

<i>list</i>	The <a href="#">SubProblem List</a> to delete.
-------------	--

Definition at line 23 of file [b\\_and\\_b\\_data.c](#).

**9.13.2.6 delete\_SubProblemList\_elem\_index()**

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

Remove a [SubProblem](#) from a specific index of a [SubProblem List](#).

**Parameters**

<i>list</i>	The <a href="#">SubProblem List</a> to modify.
<i>index</i>	The index of the <a href="#">SubProblem</a> to remove.

Definition at line 113 of file [b\\_and\\_b\\_data.c](#).

**9.13.2.7 delete\_SubProblemList\_iterator()**

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

Delete a [SubProblem List](#) iterator.

## Parameters

<i>iterator</i>	The <a href="#">SubProblem List</a> iterator.
-----------------	---

Definition at line 198 of file [b\\_and\\_b\\_data.c](#).

**9.13.2.8 `get_current_openSubProblemList_iterator_element()`**

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

Definition at line 174 of file [b\\_and\\_b\\_data.c](#).

**9.13.2.9 `get_SubProblemList_elem_index()`**

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

Get a [SubProblem](#) from a specific index of a [SubProblem List](#).

## Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to inspect.
<i>index</i>	The index of the <a href="#">SubProblem</a> to get.

## Returns

The [SubProblem](#) at the specified index.

Definition at line 146 of file [b\\_and\\_b\\_data.c](#).

**9.13.2.10 `get_SubProblemList_size()`**

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

Get the size of a [SubProblem List](#).

## Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to inspect.
-------------	---

**Returns**

The size of the [SubProblem List](#).

Definition at line 54 of file [b\\_and\\_b\\_data.c](#).

**9.13.2.11 is\_SubProblemList\_empty()**

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

Check if a [SubProblem List](#) is empty.

**Parameters**

<i>list</i>	The <a href="#">SubProblem List</a> to check.
-------------	---

**Returns**

True if the [SubProblem List](#) is empty, false otherwise.

Definition at line 39 of file [b\\_and\\_b\\_data.c](#).

**9.13.2.12 is\_SubProblemList\_iterator\_valid()**

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

Check if a [SubProblem List](#) iterator is valid.

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

**Parameters**

<i>iterator</i>	The <a href="#">SubProblem List</a> iterator to check.
-----------------	--

**Returns**

True if the [SubProblem List](#) iterator is valid, false otherwise.

Definition at line 170 of file [b\\_and\\_b\\_data.c](#).



### 9.13.2.13 list\_openSubProblemList\_next()

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

Definition at line 178 of file [b\\_and\\_b\\_data.c](#).

### 9.13.2.14 new\_SubProblemList()

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

Create a new [SubProblem List](#).

#### Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to create.
-------------	--

Definition at line 17 of file [b\\_and\\_b\\_data.c](#).

### 9.13.2.15 SubProblemList\_iterator\_get\_next()

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

Get the next element of a [SubProblem List](#) iterator.

#### Parameters

<i>iterator</i>	The <a href="#">SubProblem List</a> iterator.
-----------------	---

#### Returns

The next element of the [List](#) pointed by the iterator.

Definition at line 188 of file [b\\_and\\_b\\_data.c](#).

## 9.14 b\_and\_b\_data.c

[Go to the documentation of this file.](#)

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

```

00019     list->head = list->tail = NULL;
00020 }
00021
00022
00023 void delete_SubProblemList(SubProblemsList * list){
00024     if (!list) {
00025         return;
00026     }
00027
00028     SubProblemElem *current = list->head;
00029     SubProblemElem *next;
00030
00031     while (current) {
00032         next = current->next;
00033         free(current);
00034         current = next;
00035     }
00036 }
00037
00038
00039 bool is_SubProblemList_empty(SubProblemsList *list){
00040     return (list == NULL || list->size == 0);
00041 }
00042
00043
00044 SubProblemElem *build_list_elem(SubProblem *value, SubProblemElem *next, SubProblemElem *prev) {
00045     SubProblemElem *e = malloc(sizeof(SubProblemElem));
00046     e->subProblem = *value;
00047     e->next = next;
00048     e->prev = prev;
00049
00050     return e;
00051 }
00052
00053
00054 size_t get_SubProblemList_size(SubProblemsList *list){
00055     return (list != NULL) ? list->size : 0;
00056 }
00057
00058
00059 void add_elem_SubProblemList_bottom(SubProblemsList *list, SubProblem *element){
00060     if (list == NULL) {
00061         return;
00062     }
00063
00064     SubProblemElem *e = build_list_elem(element, NULL, list->tail);
00065
00066     if (is_SubProblemList_empty(list))
00067         list->head = e;
00068     else
00069         list->tail->next = e;
00070     list->tail = e;
00071     list->size++;
00072 }
00073
00074
00075 void add_elem_SubProblemList_index(SubProblemsList *list, SubProblem *element, size_t index){
00076     if (!list || index > get_SubProblemList_size(list)) {
00077         return;
00078     }
00079
00080     // support element is a temporary pointer which avoids losing data
00081     SubProblemElem *e;
00082     SubProblemElem *supp = list->head;
00083
00084     for (size_t i = 0; i < index; ++i)
00085         supp = supp->next;
00086
00087     if (supp == list->head) {
00088         e = build_list_elem(element, supp, NULL);
00089
00090         if (supp == NULL) {
00091             list->head = list->tail = e;
00092         } else {
00093             e->next->prev = e;
00094             list->head->prev = e;
00095             list->head = e;
00096         }
00097     } else {
00098         if (supp == NULL) {
00099             e = build_list_elem(element, NULL, list->tail);
00100             list->tail->next = e;
00101         } else {
00102             e = build_list_elem(element, supp, supp->prev);
00103             e->next->prev = e;
00104             e->prev->next = e;
00105         }
00106     }

```

```

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

```

```

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

```

## 9.15 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_← \_structures/b\_and\_b\_data.h File Reference

The data structures used in the Branch and Bound algorithm.

```
#include "mst.h"
```

### Classes

- struct [SubProblem](#)  
The struct used to represent a [SubProblem](#) or node of the Branch and Bound tree.
- struct [Problem](#)  
The struct used to represent the overall problem.
- struct [SubProblemElem](#)  
The element of the list of SubProblems.
- struct [SubProblemsList](#)  
The list of open SubProblems.
- struct [SubProblemsListIterator](#)  
The iterator of the list of SubProblems.

### Typedefs

- typedef enum [BBNodeType](#) [BBNodeType](#)  
The labels used to identify the type of a [SubProblem](#).
- typedef enum [ConstraintType](#) [ConstraintType](#)  
The enum used to identify the type of [Edge](#) constraints.
- typedef struct [SubProblem](#) [SubProblem](#)  
The struct used to represent a [SubProblem](#) or node of the Branch and Bound tree.
- typedef struct [Problem](#) [Problem](#)  
The struct used to represent the overall problem.
- typedef struct [SubProblemElem](#) [SubProblemElem](#)  
The element of the list of SubProblems.
- typedef struct [SubProblemsList](#) [SubProblemsList](#)  
The list of open SubProblems.

### Enumerations

- enum [BBNodeType](#) {  
    [OPEN](#) , [CLOSED\\_NN](#) , [CLOSED\\_NN\\_HYBRID](#) , [CLOSED\\_BOUND](#) ,  
    [CLOSED\\_UNFEASIBLE](#) , [CLOSED\\_1TREE](#) , [CLOSED\\_SUBGRADIENT](#) }  
The labels used to identify the type of a [SubProblem](#).
- enum [ConstraintType](#) { [NOTHING](#) , [MANDATORY](#) , [FORBIDDEN](#) }  
The enum used to identify the type of [Edge](#) constraints.

## Functions

- void `new_SubProblemList` (SubProblemsList \*list)  
*Create a new SubProblem List.*
- void `delete_SubProblemList` (SubProblemsList \*list)  
*Delete a SubProblem List.*
- bool `is_SubProblemList_empty` (SubProblemsList \*list)  
*Check if a SubProblem List is empty.*
- size\_t `get_SubProblemList_size` (SubProblemsList \*list)  
*Get the size of a SubProblem List.*
- void `add_elem_SubProblemList_bottom` (SubProblemsList \*list, SubProblem \*element)  
*Add a SubProblem to the bottom of a SubProblem List.*
- void `add_elem_SubProblemList_index` (SubProblemsList \*list, SubProblem \*element, size\_t index)  
*Add a SubProblem at a specific index of a SubProblem List.*
- void `delete_SubProblemList_elem_index` (SubProblemsList \*list, size\_t index)  
*Remove a SubProblem from a specific index of a SubProblem List.*
- SubProblem \* `get_SubProblemList_elem_index` (SubProblemsList \*list, size\_t index)  
*Get a SubProblem from a specific index of a SubProblem List.*
- SubProblemsListIterator \* `create_SubProblemList_iterator` (SubProblemsList \*list)  
*Create a new SubProblem List iterator on a SubProblem List.*
- bool `is_SubProblemList_iterator_valid` (SubProblemsListIterator \*iterator)  
*Check if a SubProblem List iterator is valid.*
- SubProblem \* `SubProblemList_iterator_get_next` (SubProblemsListIterator \*iterator)  
*Get the next element of a SubProblem List iterator.*
- void `delete_SubProblemList_iterator` (SubProblemsListIterator \*iterator)  
*Delete a SubProblem List iterator.*

### 9.15.1 Detailed Description

The data structures used in the Branch and Bound algorithm.

#### Author

Lorenzo Sciandra

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

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file `b_and_b_data.h`.

## 9.15.2 Typedef Documentation

### 9.15.2.1 BBNodeType

```
typedef enum BBNodeType BBNodeType
```

The labels used to identify the type of a [SubProblem](#).

### 9.15.2.2 ConstraintType

```
typedef enum ConstraintType ConstraintType
```

The enum used to identify the type of [Edge](#) constraints.

### 9.15.2.3 Problem

```
typedef struct Problem Problem
```

The struct used to represent the overall problem.

### 9.15.2.4 SubProblem

```
typedef struct SubProblem SubProblem
```

The struct used to represent a [SubProblem](#) or node of the Branch and Bound tree.

### 9.15.2.5 SubProblemElem

```
typedef struct SubProblemElem SubProblemElem
```

The element of the list of SubProblems.

### 9.15.2.6 SubProblemsList

```
typedef struct SubProblemsList SubProblemsList
```

The list of open SubProblems.

## 9.15.3 Enumeration Type Documentation

### 9.15.3.1 BBNodeType

```
enum BBNodeType
```

The labels used to identify the type of a [SubProblem](#).

## Enumerator

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

Definition at line 22 of file [b\\_and\\_b\\_data.h](#).

### 9.15.3.2 ConstraintType

```
enum ConstraintType
```

The enum used to identify the type of [Edge](#) constraints.

## Enumerator

NOTHING	The <a href="#">Edge</a> has no constraints.
MANDATORY	The <a href="#">Edge</a> is mandatory.
FORBIDDEN	The <a href="#">Edge</a> is forbidden.

Definition at line 34 of file [b\\_and\\_b\\_data.h](#).

## 9.15.4 Function Documentation

### 9.15.4.1 add\_elem\_SubProblemList\_bottom()

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

Add a [SubProblem](#) to the bottom of a [SubProblem List](#).

## Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to modify.
<i>element</i>	The <a href="#">SubProblem</a> to add.

Definition at line 59 of file [b\\_and\\_b\\_data.c](#).

#### 9.15.4.2 `add_elem_SubProblemList_index()`

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

Add a [SubProblem](#) at a specific index of a [SubProblem List](#).

##### Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to modify.
<i>element</i>	The <a href="#">SubProblem</a> to add.
<i>index</i>	The index where to add the <a href="#">SubProblem</a> .

list is clearer way but it is already checked inside `get_list_size`

Definition at line 75 of file [b\\_and\\_b\\_data.c](#).

#### 9.15.4.3 `create_SubProblemList_iterator()`

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

Create a new [SubProblem List](#) iterator on a [SubProblem List](#).

##### Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to iterate.
-------------	---

##### Returns

the [SubProblem List](#) iterator.

Definition at line 159 of file [b\\_and\\_b\\_data.c](#).

#### 9.15.4.4 `delete_SubProblemList()`

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

Delete a [SubProblem List](#).



## Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to delete.
-------------	--

Definition at line 23 of file [b\\_and\\_b\\_data.c](#).

#### 9.15.4.5 delete\_SubProblemList\_elem\_index()

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

Remove a [SubProblem](#) from a specific index of a [SubProblem List](#).

## Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to modify.
<i>index</i>	The index of the <a href="#">SubProblem</a> to remove.

Definition at line 113 of file [b\\_and\\_b\\_data.c](#).

#### 9.15.4.6 delete\_SubProblemList\_iterator()

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

Delete a [SubProblem List](#) iterator.

## Parameters

<i>iterator</i>	The <a href="#">SubProblem List</a> iterator.
-----------------	---

Definition at line 198 of file [b\\_and\\_b\\_data.c](#).

#### 9.15.4.7 get\_SubProblemList\_elem\_index()

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

Get a [SubProblem](#) from a specific index of a [SubProblem List](#).

**Parameters**

<i>list</i>	The <a href="#">SubProblem List</a> to inspect.
<i>index</i>	The index of the <a href="#">SubProblem</a> to get.

**Returns**

The [SubProblem](#) at the specified index.

Definition at line 146 of file [b\\_and\\_b\\_data.c](#).

**9.15.4.8 [get\\_SubProblemList\\_size\(\)](#)**

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

Get the size of a [SubProblem List](#).

**Parameters**

<i>list</i>	The <a href="#">SubProblem List</a> to inspect.
-------------	---

**Returns**

The size of the [SubProblem List](#).

Definition at line 54 of file [b\\_and\\_b\\_data.c](#).

**9.15.4.9 [is\\_SubProblemList\\_empty\(\)](#)**

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

Check if a [SubProblem List](#) is empty.

**Parameters**

<i>list</i>	The <a href="#">SubProblem List</a> to check.
-------------	---

**Returns**

True if the [SubProblem List](#) is empty, false otherwise.

Definition at line 39 of file [b\\_and\\_b\\_data.c](#).

#### 9.15.4.10 is\_SubProblemList\_iterator\_valid()

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

Check if a [SubProblem List](#) iterator is valid.

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

##### Parameters

<i>iterator</i>	The <a href="#">SubProblem List</a> iterator to check.
-----------------	--

##### Returns

True if the [SubProblem List](#) iterator is valid, false otherwise.

Definition at line 170 of file [b\\_and\\_b\\_data.c](#).

#### 9.15.4.11 new\_SubProblemList()

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

Create a new [SubProblem List](#).

##### Parameters

<i>list</i>	The <a href="#">SubProblem List</a> to create.
-------------	--

Definition at line 17 of file [b\\_and\\_b\\_data.c](#).

#### 9.15.4.12 SubProblemList\_iterator\_get\_next()

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

Get the next element of a [SubProblem List](#) iterator.

##### Parameters

<i>iterator</i>	The <a href="#">SubProblem List</a> iterator.
-----------------	---

**Returns**

The next element of the [List](#) pointed by the iterator.

Definition at line 188 of file [b\\_and\\_b\\_data.c](#).

**9.16 b\_and\_b\_data.h**

[Go to the documentation of this file.](#)

```

00001
00016 #ifndef BRANCHANDBOUND1TREE_B_AND_B_DATA_H
00017 #define BRANCHANDBOUND1TREE_B_AND_B_DATA_H
00018
00019 #include "mst.h"
00020
00022 typedef enum BBNodeType{
00023     OPEN,
00024     CLOSED_NN,
00025     CLOSED_NN_HYBRID,
00026     CLOSED_BOUND,
00027     CLOSED_UNFEASIBLE,
00028     CLOSED_ITREE,
00029     CLOSED_SUBGRADIENT
00030 }BBNodeType;
00031
00032
00034 typedef enum ConstraintType{
00035     NOTHING,
00036     MANDATORY,
00037     FORBIDDEN
00038 }ConstraintType;
00039
00040
00042 typedef struct SubProblem{
00043     BBNodeType type;
00044     unsigned int id;
00045     unsigned int fatherId;
00046     double value;
00047     unsigned short treeLevel;
00048     float timeToReach;
00049     MST oneTree;
00050     unsigned short num_edges_in_cycle;
00051     double prob;
00052     ConstrainedEdge cycleEdges [MAX_VERTEX_NUM];
00053     unsigned short num_forbidden_edges;
00054     unsigned short num_mandatory_edges;
00055     int edge_to_branch;
00056     ConstrainedEdge mandatoryEdges [MAX_VERTEX_NUM];
00057     ConstraintType constraints [MAX_VERTEX_NUM] [MAX_VERTEX_NUM];
00058 }SubProblem;
00059
00060
00062 typedef struct Problem{
00063     Graph graph;
00064     Graph reformulationGraph;
00065     unsigned short candidateNodeId;
00066     unsigned short totTreeLevels;
00067     SubProblem bestSolution;
00068     double bestValue;
00069     unsigned int generatedBBNodes;
00070     unsigned int exploredBBNodes;
00071     unsigned int num_fixed_edges;
00072     bool interrupted;
00073     clock_t start;
00074     clock_t end;
00075 }Problem;
00076
00077
00079 typedef struct SubProblemElem{
00080     SubProblem subProblem;
00081     struct SubProblemElem * next;
00082     struct SubProblemElem * prev;
00083 }SubProblemElem;
00084
00085
00087 typedef struct SubProblemsList{
00088     SubProblemElem * head;
00089     SubProblemElem * tail;
00090     size_t size;

```

```

00091 }SubProblemsList;
00092
00093
00095 typedef struct {
00096     SubProblemsList * list;
00097     SubProblemElem * curr;
00098     size_t index;
00099 } SubProblemsListIterator;
00100
00101
00106 void new_SubProblemList (SubProblemsList * list);
00107
00108
00113 void delete_SubProblemList (SubProblemsList * list);
00114
00115
00121 bool is_SubProblemList_empty (SubProblemsList *list);
00122
00123
00129 size_t get_SubProblemList_size (SubProblemsList *list);
00130
00131
00137 void add_elem_SubProblemList_bottom (SubProblemsList *list, SubProblem *element);
00138
00139
00146 void add_elem_SubProblemList_index (SubProblemsList *list, SubProblem *element, size_t index);
00147
00148
00154 void delete_SubProblemList_elem_index (SubProblemsList *list, size_t index);
00155
00156
00163 SubProblem *get_SubProblemList_elem_index (SubProblemsList *list, size_t index);
00164
00165
00171 SubProblemsListIterator *create_SubProblemList_iterator (SubProblemsList *list);
00172
00173
00175
00180 bool is_SubProblemList_iterator_valid (SubProblemsListIterator *iterator);
00181
00182
00188 SubProblem *SubProblemList_iterator_get_next (SubProblemsListIterator *iterator);
00189
00190
00195 void delete_SubProblemList_iterator (SubProblemsListIterator *iterator);
00196
00197 #endif //BRANCHANDBOUND1TREE_B_AND_B_DATA_H

```

## 9.17 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/doubly\_linked\_list/linked\_list.h File Reference

```

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

```

### Classes

- struct [DlIElem](#)  
The double linked [List](#) element.
- struct [List](#)  
The double linked list.
- struct [ListIterator](#)  
The iterator for the [List](#).

## Macros

- `#define` [BRANCHANDBOUND1TREE\\_LINKED\\_LIST\\_H](#)

## Typedefs

- `typedef struct` [DllElem](#) [DllElem](#)  
The double linked [List](#) element.

### 9.17.1 Macro Definition Documentation

#### 9.17.1.1 BRANCHANDBOUND1TREE\_LINKED\_LIST\_H

`#define` [BRANCHANDBOUND1TREE\\_LINKED\\_LIST\\_H](#)

Definition at line 23 of file [linked\\_list.h](#).

### 9.17.2 Typedef Documentation

#### 9.17.2.1 DllElem

`typedef struct` [DllElem](#) [DllElem](#)

The double linked [List](#) element.

## 9.18 [linked\\_list.h](#)

[Go to the documentation of this file.](#)

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

## 9.19 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/doubly\_linked\_list/list\_functions.c File Reference

The definition of the functions to manipulate the [List](#).

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

### Functions

- [List](#) \* [new\\_list](#) (void)  
*Create a new instance of a [List](#).*
- void [del\\_list](#) ([List](#) \*list)  
*Delete an instance of a [List](#).*
- [DIIElem](#) \* [build\\_dll\\_elem](#) (void \*value, [DIIElem](#) \*next, [DIIElem](#) \*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 [DIIElem](#) to the bottom of the [List](#).*
- void [add\\_elem\\_list\\_index](#) ([List](#) \*list, void \*element, size\_t index)  
*Adds an [DIIElem](#) at the index indicated of the [List](#).*
- void [delete\\_list\\_elem\\_bottom](#) ([List](#) \*list)  
*Deletes the [DIIElem](#) at the bottom of the [List](#).*
- void [delete\\_list\\_elem\\_index](#) ([List](#) \*list, size\_t index)  
*Deletes the [DIIElem](#) at the indicated index of the [List](#).*
- void \* [get\\_list\\_elem\\_index](#) ([List](#) \*list, size\_t index)  
*Retrieves a pointer to an [DIIElem](#) from the [List](#).*

### 9.19.1 Detailed Description

The definition of the functions to manipulate the [List](#).

#### Authors

Lorenzo Sciandra, Stefano Vittorio Porta and Ivan Spada

This is a double linked [List](#) implementation that we have realized for an university project.

#### Version

1.0.0

#### Date

2019-07-9

#### Copyright

Copyright (c) 2024, license MIT

Repo: <https://gitlab.com/Stefal68/laboratorio-algoritmi-2018-19/>

Definition in file [list\\_functions.c](#).

## 9.19.2 Function Documentation

### 9.19.2.1 add\_elem\_list\_bottom()

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

Adds an [DIIElem](#) to the bottom of the [List](#).

#### Parameters

<i>list</i>	The <a href="#">List</a> to add the <a href="#">DIIElem</a> to.
<i>element</i>	The <a href="#">DIIElem</a> to add.

Definition at line 66 of file [list\\_functions.c](#).

### 9.19.2.2 add\_elem\_list\_index()

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

Adds an [DIIElem](#) at the index indicated of the [List](#).

#### Parameters

<i>array</i>	The <a href="#">List</a> to add the <a href="#">DIIElem</a> to.
<i>element</i>	The <a href="#">DIIElem</a> to add.
<i>index</i>	At what index to add the <a href="#">DIIElem</a> to the <a href="#">List</a> .

list is clearer way but it is already checked inside get\_list\_size

Definition at line 86 of file [list\\_functions.c](#).

### 9.19.2.3 build\_dll\_elem()

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

Definition at line 46 of file [list\\_functions.c](#).



### 9.19.2.4 del\_list()

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

Delete an instance of a [List](#).

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

#### Parameters

<i>list</i>	The <a href="#">List</a> to delete.
-------------	-------------------------------------

Definition at line [28](#) of file [list\\_functions.c](#).

### 9.19.2.5 delete\_list\_elem\_bottom()

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

Deletes the [DIIElem](#) at the bottom of the [List](#).

#### Parameters

<i>list</i>	The <a href="#">List</a> to remove the <a href="#">DIIElem</a> from.
-------------	--

Definition at line [127](#) of file [list\\_functions.c](#).

### 9.19.2.6 delete\_list\_elem\_index()

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

Deletes the [DIIElem](#) at the indicated index of the [List](#).

#### Parameters

<i>list</i>	The <a href="#">List</a> to remove the <a href="#">DIIElem</a> from.
<i>index</i>	The index of the <a href="#">DIIElem</a> to remove from the <a href="#">List</a> .

list is clearer but it is already checked inside get\_list\_size

Definition at line [147](#) of file [list\\_functions.c](#).

### 9.19.2.7 `get_list_elem_index()`

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

Retrieves a pointer to an [DIIElem](#) from the [List](#).

#### Parameters

<i>list</i>	The <a href="#">List</a> to retrieve the <a href="#">DIIElem</a> from.
<i>index</i>	The index of the <a href="#">DIIElem</a> to retrieve.

#### Returns

A pointer to the retrieved [DIIElem](#).

Definition at line 185 of file [list\\_functions.c](#).

### 9.19.2.8 `get_list_size()`

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

Gets the size of the [List](#).

#### Parameters

<i>list</i>	Pointer to the <a href="#">List</a> to check.
-------------	---

#### Returns

Size of the [List](#) l.

Definition at line 61 of file [list\\_functions.c](#).

### 9.19.2.9 `is_list_empty()`

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

Check if the [List](#) is empty.

#### Parameters

<i>list</i>	Pointer to the <a href="#">List</a> to check.
-------------	---

**Returns**

true if empty, false otherwise.

Definition at line 56 of file [list\\_functions.c](#).

**9.19.2.10 new\_list()**

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

Create a new instance of a [List](#).

**Returns**

The newly created [List](#).

Definition at line 18 of file [list\\_functions.c](#).

**9.20 list\_functions.c**

[Go to the documentation of this file.](#)

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

```

00058 }
00059
00060
00061 size_t get_list_size(List *list) {
00062     return (list != NULL) ? list->size : 0;
00063 }
00064
00065
00066 void add_elem_list_bottom(List *list, void *element) {
00067     if (list == NULL) {
00068         return;
00069     }
00070
00071     DllElem *e = build_dll_elem(element, NULL, list->tail);
00072
00073     if (is_list_empty(list))
00074         list->head = e;
00075     else
00076         list->tail->next = e;
00077     list->tail = e;
00078     list->size++;
00079 }
00080
00081
00082 /*
00083  * This method deletes the element at the indicated index.
00084  * If the index is greater than the size of the List, no element is removed.
00085  */
00086 void add_elem_list_index(List *list, void *element, size_t index) {
00087     if (!list || index > get_list_size(list)) {
00088         return;
00089     }
00090 }
00091
00092 // support element is a temporary pointer which avoids losing data
00093 DllElem *e;
00094 DllElem *supp = list->head;
00095
00096 for (size_t i = 0; i < index; ++i)
00097     supp = supp->next;
00098
00099 if (supp == list->head) {
00100     e = build_dll_elem(element, supp, NULL);
00101
00102     if (supp == NULL) {
00103         list->head = list->tail = e;
00104     } else {
00105         e->next->prev = e;
00106         list->head->prev = e;
00107         list->head = e;
00108     }
00109 } else {
00110     if (supp == NULL) {
00111         e = build_dll_elem(element, NULL, list->tail);
00112         list->tail->next = e;
00113     } else {
00114         e = build_dll_elem(element, supp, supp->prev);
00115         e->next->prev = e;
00116         e->prev->next = e;
00117     }
00118 }
00119
00120 list->size++;
00121 }
00122
00123
00124 /*
00125  * This method deletes the element at the bottom of the List.
00126  */
00127 void delete_list_elem_bottom(List *list) {
00128
00129     if (list == NULL || is_list_empty(list)) {
00130         return;
00131     }
00132
00133     DllElem *oldTail = list->tail;
00134
00135     list->tail = oldTail->prev;
00136     list->tail->next = NULL;
00137
00138     free(oldTail);
00139     list->size--;
00140 }
00141
00142
00143 /*
00144  * This method iteratively finds and deletes the element at the specified index, but only if it
    doesn't exceed

```

```

00145  * the size of the List. In this case, instead, no reference gets deleted.
00146  */
00147 void delete_list_elem_index(List *list, size_t index) {
00149     if (list == NULL || is_list_empty(list) || index >= get_list_size(list)) {
00150         return;
00151     }
00152
00153     DllElem *oldElem;
00154     oldElem = list->head;
00155
00156     for (size_t i = 0; i < index; ++i)
00157         oldElem = oldElem->next;
00158
00159     // Found index to remove!!
00160     if (oldElem != list->head) {
00161         oldElem->prev->next = oldElem->next;
00162         if (oldElem->next != NULL) {
00163             oldElem->next->prev = oldElem->prev;
00164         } else {
00165             list->tail = oldElem->prev;
00166         }
00167     } else {
00168         if (list->head == list->tail) {
00169             list->head = list->tail = NULL;
00170         } else {
00171             list->head = list->head->next;
00172             list->head->prev = NULL;
00173         }
00174     }
00175
00176     free(oldElem);
00177     list->size--;
00178 }
00179
00180
00181 /*
00182  * This method iteratively runs through the dllist elements and returns the one at the requested
00183  * index.
00184  * If the index exceeds the size of the List, we instead return no element.
00185  */
00186 void *get_list_elem_index(List *list, size_t index) {
00187     if (list == NULL || index >= get_list_size(list)) {
00188         return NULL;
00189     }
00190
00191     DllElem *supp; // iteration support element
00192     supp = list->head;
00193
00194     for (size_t i = 0; i < index; ++i)
00195         supp = supp->next;
00196     return supp->value;
00197 }

```

## 9.21 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/doubly\_linked\_list/list\_functions.h File Reference

The declaration of the functions to manipulate the [List](#).

```
#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 [DIIElem](#) to the bottom of the [List](#).*

- void [add\\_elem\\_list\\_index](#) ([List](#) \*array, void \*element, size\_t index)

*Adds an [DIIElem](#) at the index indicated of the [List](#).*

- void [delete\\_list\\_elem\\_bottom](#) ([List](#) \*list)

*Deletes the [DIIElem](#) at the bottom of the [List](#).*

- void [delete\\_list\\_elem\\_index](#) ([List](#) \*list, size\_t index)

*Deletes the [DIIElem](#) at the indicated index of the [List](#).*

- void \* [get\\_list\\_elem\\_index](#) ([List](#) \*list, size\_t index)

*Retrieves a pointer to an [DIIElem](#) from the [List](#).*

## 9.21.1 Detailed Description

The declaration of the functions to manipulate the [List](#).

### Authors

Lorenzo Sciandra, Stefano Vittorio Porta and Ivan Spada

This is a double linked [List](#) implementation that we have realized for an university project.

### Version

1.0.0

### Date

2019-07-9

### Copyright

Copyright (c) 2024, license MIT

Repo: <https://gitlab.com/Stefal68/laboratorio-algoritmi-2018-19/>

Definition in file [list\\_functions.h](#).

## 9.21.2 Function Documentation

### 9.21.2.1 [add\\_elem\\_list\\_bottom\(\)](#)

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

Adds an [DIIElem](#) to the bottom of the [List](#).

#### Parameters

<i>list</i>	The <a href="#">List</a> to add the <a href="#">DIIElem</a> to.
<i>element</i>	The <a href="#">DIIElem</a> to add.

Definition at line 66 of file [list\\_functions.c](#).

#### 9.21.2.2 add\_elem\_list\_index()

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

Adds an [DIIElem](#) at the index indicated of the [List](#).

#### Parameters

<i>array</i>	The <a href="#">List</a> to add the <a href="#">DIIElem</a> to.
<i>element</i>	The <a href="#">DIIElem</a> to add.
<i>index</i>	At what index to add the <a href="#">DIIElem</a> to the <a href="#">List</a> .

list is clearer way but it is already checked inside get\_list\_size

Definition at line 86 of file [list\\_functions.c](#).

#### 9.21.2.3 del\_list()

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

Delete an instance of a [List](#).

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

#### Parameters

<i>list</i>	The <a href="#">List</a> to delete.
-------------	-------------------------------------

Definition at line 28 of file [list\\_functions.c](#).

#### 9.21.2.4 delete\_list\_elem\_bottom()

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

Deletes the [DIIElem](#) at the bottom of the [List](#).

#### Parameters

<i>list</i>	The <a href="#">List</a> to remove the <a href="#">DIIElem</a> from.
-------------	--

Definition at line 127 of file [list\\_functions.c](#).

### 9.21.2.5 delete\_list\_elem\_index()

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

Deletes the [DIIElem](#) at the indicated index of the [List](#).

#### Parameters

<i>list</i>	The <a href="#">List</a> to remove the <a href="#">DIIElem</a> from.
<i>index</i>	The index of the <a href="#">DIIElem</a> to remove from the <a href="#">List</a> .

*list* is clearer but it is already checked inside `get_list_size`

Definition at line 147 of file [list\\_functions.c](#).

### 9.21.2.6 get\_list\_elem\_index()

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

Retrieves a pointer to an [DIIElem](#) from the [List](#).

#### Parameters

<i>list</i>	The <a href="#">List</a> to retrieve the <a href="#">DIIElem</a> from.
<i>index</i>	The index of the <a href="#">DIIElem</a> to retrieve.

#### Returns

A pointer to the retrieved [DIIElem](#).

Definition at line 185 of file [list\\_functions.c](#).



### 9.21.2.7 `get_list_size()`

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

Gets the size of the [List](#).

#### Parameters

<i>list</i>	Pointer to the <a href="#">List</a> to check.
-------------	---

#### Returns

Size of the [List](#) l.

Definition at line 61 of file [list\\_functions.c](#).

### 9.21.2.8 `is_list_empty()`

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

Check if the [List](#) is empty.

#### Parameters

<i>list</i>	Pointer to the <a href="#">List</a> to check.
-------------	---

#### Returns

true if empty, false otherwise.

Definition at line 56 of file [list\\_functions.c](#).

### 9.21.2.9 `new_list()`

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

Create a new instance of a [List](#).

#### Returns

The newly created [List](#).

Definition at line 18 of file [list\\_functions.c](#).

## 9.22 list\_functions.h

[Go to the documentation of this file.](#)

```

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

```

## 9.23 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_← \_structures/doubly\_linked\_list/list\_iterator.c File Reference

The definition of the functions to manipulate the [ListIterator](#).

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

### Functions

- [ListIterator](#) \* [create\\_list\\_iterator](#) ([List](#) \*list)  
*Used for the creation of a new [ListIterator](#).*
- void \* [get\\_current\\_list\\_iterator\\_element](#) ([ListIterator](#) \*iterator)  
*Method used to get the current [DIIElem](#) 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 [DIIElem](#) of an [ListIterator](#) and moves the pointer to the next object.*

### 9.23.1 Detailed Description

The definition of the functions to manipulate the [ListIterator](#).

#### Authors

Lorenzo Sciandra, Stefano Vittorio Porta and Ivan Spada

This is a double linked [List](#) implementation that we have realized for an university project.

#### Version

1.0.0

#### Date

2019-07-9

#### Copyright

Copyright (c) 2024, license MIT

Repo: <https://gitlab.com/Stefal68/laboratorio-algoritmi-2018-19/>

Definition in file [list\\_iterator.c](#).

### 9.23.2 Function Documentation

#### 9.23.2.1 create\_list\_iterator()

```
ListIterator * create_list_iterator (  
    List * list )
```

Used for the creation of a new [ListIterator](#).

#### Parameters

<i>The</i>	<a href="#">List</a> that the new <a href="#">ListIterator</a> will point.
------------	--

#### Returns

A new [ListIterator](#).

Definition at line 18 of file [list\\_iterator.c](#).

### 9.23.2.2 delete\_list\_iterator()

```
void delete_list_iterator (
    ListIterator * iterator )
```

Delete the [ListIterator](#) given.

#### Parameters

An	<a href="#">ListIterator</a> .
----	--------------------------------

Definition at line 51 of file [list\\_iterator.c](#).

### 9.23.2.3 get\_current\_list\_iterator\_element()

```
void * get_current_list_iterator_element (
    ListIterator * iterator )
```

Method used to get the current [DIIElem](#) of an [ListIterator](#).

#### Parameters

An	<a href="#">ListIterator</a> .
----	--------------------------------

#### Returns

A pointer to the current [DIIElem](#).

Definition at line 30 of file [list\\_iterator.c](#).

### 9.23.2.4 is\_list\_iterator\_valid()

```
bool is_list_iterator_valid (
    ListIterator * iterator )
```

Used to check if the [ListIterator](#) is valid.

#### Parameters

The	Iterator we want to analyze.
-----	------------------------------

#### Returns

true if it's valid, false otherwise.

Definition at line 35 of file [list\\_iterator.c](#).

### 9.23.2.5 list\_iterator\_get\_next()

```
void * list_iterator_get_next (
    ListIterator * iterator )
```

Method that retrieves the current [DlElem](#) of an [ListIterator](#) and moves the pointer to the next object.

#### Parameters

<i>iterator</i>	The <a href="#">ListIterator</a> to use.
-----------------	--

#### Returns

The currently pointed object.

Definition at line 56 of file [list\\_iterator.c](#).

### 9.23.2.6 list\_iterator\_next()

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

Used to move the [ListIterator](#) to the next value of the object.

#### Parameters

/	The <a href="#">ListIterator</a> considered.
---	--

Definition at line 40 of file [list\\_iterator.c](#).

## 9.24 list\_iterator.c

[Go to the documentation of this file.](#)

```
00001
00015 #include "list_functions.h"
00016
00017
00018 ListIterator *create_list_iterator(List *list) {
00019     if (!list)
00020         return NULL;
00021
00022     ListIterator *new_iterator = malloc(sizeof(ListIterator));
00023     new_iterator->list = list;
00024     new_iterator->curr = new_iterator->list->head;
00025     new_iterator->index = 0;
00026     return new_iterator;
00027 }
```

```

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

```

## 9.25 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/doubly\_linked\_list/list\_iterator.h File Reference

The declaration of the functions to manipulate the [ListIterator](#).

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

### Functions

- [ListIterator](#) \* [create\\_list\\_iterator](#) ([List](#) \*list)  
*Used for the creation of a new [ListIterator](#).*
- bool [is\\_list\\_iterator\\_valid](#) ([ListIterator](#) \*iterator)  
*Used to check if the [ListIterator](#) is valid.*
- void \* [get\\_current\\_list\\_iterator\\_element](#) ([ListIterator](#) \*iterator)  
*Method used to get the current [DlElem](#) 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 [DlElem](#) of an [ListIterator](#) and moves the pointer to the next object.*
- void [delete\\_list\\_iterator](#) ([ListIterator](#) \*iterator)  
*Delete the [ListIterator](#) given.*

## 9.25.1 Detailed Description

The declaration of the functions to manipulate the [ListIterator](#).

### Authors

Lorenzo Sciandra, Stefano Vittorio Porta and Ivan Spada

This is a double linked [List](#) implementation that we have realized for an university project.

### Version

1.0.0

### Date

2019-07-9

### Copyright

Copyright (c) 2024, license MIT

Repo: <https://gitlab.com/Stefal68/laboratorio-algoritmi-2018-19/>

Definition in file [list\\_iterator.h](#).

## 9.25.2 Function Documentation

### 9.25.2.1 create\_list\_iterator()

```
ListIterator * create_list_iterator (  
    List * list )
```

Used for the creation of a new [ListIterator](#).

#### Parameters

The	<a href="#">List</a> that the new <a href="#">ListIterator</a> will point.
-----	--

#### Returns

A new [ListIterator](#).

Definition at line 18 of file [list\\_iterator.c](#).

### 9.25.2.2 delete\_list\_iterator()

```
void delete_list_iterator (
    ListIterator * iterator )
```

Delete the [ListIterator](#) given.

#### Parameters

<i>An</i>	<a href="#">ListIterator</a> .
-----------	--------------------------------

Definition at line 51 of file [list\\_iterator.c](#).

### 9.25.2.3 get\_current\_list\_iterator\_element()

```
void * get_current_list_iterator_element (
    ListIterator * iterator )
```

Method used to get the current [DIIElem](#) of an [ListIterator](#).

#### Parameters

<i>An</i>	<a href="#">ListIterator</a> .
-----------	--------------------------------

#### Returns

A pointer to the current [DIIElem](#).

Definition at line 30 of file [list\\_iterator.c](#).

### 9.25.2.4 is\_list\_iterator\_valid()

```
bool is_list_iterator_valid (
    ListIterator * iterator )
```

Used to check if the [ListIterator](#) is valid.

#### Parameters

<i>The</i>	Iterator we want to analyze.
------------	------------------------------

#### Returns

true if it's valid, false otherwise.



Definition at line 35 of file [list\\_iterator.c](#).

### 9.25.2.5 list\_iterator\_get\_next()

```
void * list_iterator_get_next (
    ListIterator * iterator )
```

Method that retrieves the current [DlElem](#) of an [ListIterator](#) and moves the pointer to the next object.

#### Parameters

<i>iterator</i>	The <a href="#">ListIterator</a> to use.
-----------------	--

#### Returns

The currently pointed object.

Definition at line 56 of file [list\\_iterator.c](#).

### 9.25.2.6 list\_iterator\_next()

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

Used to move the [ListIterator](#) to the next value of the object.

#### Parameters

/	The <a href="#">ListIterator</a> considered.
---	--

Definition at line 40 of file [list\\_iterator.c](#).

## 9.26 list\_iterator.h

[Go to the documentation of this file.](#)

```
00001
00015 #ifndef BRANCHANDBOUNDITREE_LIST_ITERATOR_H
00016 #define BRANCHANDBOUNDITREE_LIST_ITERATOR_H
00017 #include "linked_list.h"
00018
00024 ListIterator *create_list_iterator(List *list);
00025
00026
00032 bool is_list_iterator_valid(ListIterator *iterator);
00033
00034
00040 void *get_current_list_iterator_element(ListIterator *iterator);
00041
00042
```

```

00047 void list_iterator_next(ListIterator *iterator);
00048
00049
00055 void *list_iterator_get_next(ListIterator *iterator);
00056
00057
00062 void delete_list_iterator(ListIterator *iterator);
00063
00064
00065 #endif //BRANCHANDBOUNDTREE_LIST_ITERATOR_H

```

## 9.27 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/fibonacci\_heap.c File Reference

This file contains the implementation of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem.

```
#include "fibonacci_heap.h"
```

### Functions

- void `create_fibonacci_heap` (FibonacciHeap \*heap)  
*Create an empty Fibonacci Heap.*
- void `create_node` (OrdTreeNode \*node, unsigned short key, double value)  
*Create a Node with a given key and value.*
- void `insert_node` (FibonacciHeap \*heap, OrdTreeNode \*node)  
*Insert a Node in the Fibonacci Heap.*
- void `create_insert_node` (FibonacciHeap \*heap, OrdTreeNode \*node, unsigned short key, double value)  
*A wrapper function to create a Node and insert it in the Fibonacci Heap.*
- void `link_trees` (FibonacciHeap \*heap, OrdTreeNode \*child, OrdTreeNode \*father)
- void `swap_roots` (FibonacciHeap \*heap, OrdTreeNode \*node1, OrdTreeNode \*node2)
- void `consolidate` (FibonacciHeap \*heap)
- int `extract_min` (FibonacciHeap \*heap)  
*Extract the minimum Node from the Fibonacci Heap.*
- static void `cut` (FibonacciHeap \*heap, OrdTreeNode \*node, OrdTreeNode \*parent)
- static void `cascading_cut` (FibonacciHeap \*heap, OrdTreeNode \*node)
- void `decrease_value` (FibonacciHeap \*heap, OrdTreeNode \*node, double new\_value)  
*Decrease the value of a Node in the Fibonacci Heap.*
- void `delete_node` (FibonacciHeap \*heap, OrdTreeNode \*node)

### 9.27.1 Detailed Description

This file contains the implementation of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem.

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file `fibonacci_heap.c`.

## 9.27.2 Function Documentation

### 9.27.2.1 cascading\_cut()

```
static void cascading_cut (
    FibonacciHeap * heap,
    OrdTreeNode * node ) [static]
```

Definition at line 280 of file [fibonacci\\_heap.c](#).

### 9.27.2.2 consolidate()

```
void consolidate (
    FibonacciHeap * heap )
```

Definition at line 114 of file [fibonacci\\_heap.c](#).

### 9.27.2.3 create\_fibonacci\_heap()

```
void create_fibonacci_heap (
    FibonacciHeap * heap )
```

Create an empty Fibonacci Heap.

#### Parameters

<i>heap</i>	The Fibonacci Heap to be created.
-------------	-----------------------------------

Definition at line 16 of file [fibonacci\\_heap.c](#).

### 9.27.2.4 create\_insert\_node()

```
void create_insert_node (
    FibonacciHeap * heap,
    OrdTreeNode * node,
    unsigned short key,
    double value )
```

A wrapper function to create a [Node](#) and insert it in the Fibonacci Heap.

## Parameters

<i>heap</i>	The Fibonacci Heap where the <a href="#">Node</a> will be inserted.
<i>node</i>	The <a href="#">Node</a> to be created and inserted.
<i>key</i>	The key of the <a href="#">Node</a> .
<i>value</i>	The value of the <a href="#">Node</a> .

Definition at line 63 of file [fibonacci\\_heap.c](#).

### 9.27.2.5 create\_node()

```
void create_node (
    OrdTreeNode * node,
    unsigned short key,
    double value )
```

Create a [Node](#) with a given key and value.

## Parameters

<i>node</i>	The <a href="#">Node</a> to be created.
<i>key</i>	The key of the <a href="#">Node</a> .
<i>value</i>	The value of the <a href="#">Node</a> .

Definition at line 25 of file [fibonacci\\_heap.c](#).

### 9.27.2.6 cut()

```
static void cut (
    FibonacciHeap * heap,
    OrdTreeNode * node,
    OrdTreeNode * parent ) [static]
```

Definition at line 251 of file [fibonacci\\_heap.c](#).

### 9.27.2.7 decrease\_value()

```
void decrease_value (
    FibonacciHeap * heap,
    OrdTreeNode * node,
    double new_value )
```

Decrease the value of a [Node](#) in the Fibonacci Heap.

If the new value is still greater than the parent's value, nothing happens. Otherwise, the [Node](#) becomes a root. If the parent is marked, and is not a root, it becomes a root. This process is repeated until a parent is not marked or is a root.

## Parameters

<i>heap</i>	The Fibonacci Heap where the <a href="#">Node</a> is.
<i>node</i>	The <a href="#">Node</a> whose value has to be decreased.
<i>new_value</i>	The new value of the <a href="#">Node</a> .

Definition at line 294 of file [fibonacci\\_heap.c](#).

### 9.27.2.8 delete\_node()

```
void delete_node (  
    FibonacciHeap * heap,  
    OrdTreeNode * node )
```

Definition at line 312 of file [fibonacci\\_heap.c](#).

### 9.27.2.9 extract\_min()

```
int extract_min (  
    FibonacciHeap * heap )
```

Extract the minimum [Node](#) from the Fibonacci Heap.

All the children of the minimum [Node](#) become new roots. The new minimum has to be found and, by doing so, the Heap is re-ordered to maintain the Heap property and minimize the height of the Heap-ordered Trees.

## Parameters

<i>heap</i>	The Fibonacci Heap where the <a href="#">Node</a> will be extracted.
-------------	--

## Returns

The key of the minimum [Node](#) if the Heap is not empty, -1 otherwise.

Definition at line 175 of file [fibonacci\\_heap.c](#).

### 9.27.2.10 insert\_node()

```
void insert_node (  
    FibonacciHeap * heap,  
    OrdTreeNode * node )
```

Insert a [Node](#) in the Fibonacci Heap.

## Parameters

<i>heap</i>	The Fibonacci Heap where the <a href="#">Node</a> will be inserted.
<i>node</i>	The <a href="#">Node</a> to be inserted.

Definition at line 38 of file [fibonacci\\_heap.c](#).

### 9.27.2.11 link\_trees()

```
void link_trees (
    FibonacciHeap * heap,
    OrdTreeNode * child,
    OrdTreeNode * father )
```

Definition at line 69 of file [fibonacci\\_heap.c](#).

### 9.27.2.12 swap\_roots()

```
void swap_roots (
    FibonacciHeap * heap,
    OrdTreeNode * node1,
    OrdTreeNode * node2 )
```

Definition at line 95 of file [fibonacci\\_heap.c](#).

## 9.28 fibonacci\_heap.c

[Go to the documentation of this file.](#)

```
00001
00013 #include "fibonacci_heap.h"
00014
00015
00016 void create_fibonacci_heap(FibonacciHeap * heap){
00017     heap->min_root = NULL;
00018     heap->head_tree_list = NULL;
00019     heap->tail_tree_list = NULL;
00020     heap->num_nodes = 0;
00021     heap->num_trees = 0;
00022 }
00023
00024
00025 void create_node(OrdTreeNode * node, unsigned short key, double value){
00026     node->value = value;
00027     node->key = key;
00028     node->parent = NULL;
00029     node->left_sibling = node;
00030     node->right_sibling = node;
00031
00032     node->head_child_list = NULL;
00033     node->tail_child_list = NULL;
00034     node->num_children = 0;
00035 }
00036
00037
00038 void insert_node(FibonacciHeap * heap, OrdTreeNode * node){
00039
00040     node->marked = false;
```

```

00041     node->is_root = true;
00042
00043     if(heap->num_trees == 0){
00044         heap->min_root = node;
00045         heap->head_tree_list = node;
00046         heap->tail_tree_list = node;
00047     }
00048     else{
00049         if(node->value < heap->min_root->value){
00050             heap->min_root = node;
00051         }
00052         heap->tail_tree_list->right_sibling = node;
00053         heap->head_tree_list->left_sibling = node;
00054         node->left_sibling = heap->tail_tree_list;
00055         node->right_sibling = heap->head_tree_list;
00056         heap->tail_tree_list = node;
00057     }
00058     heap->num_nodes++;
00059     heap->num_trees++;
00060 }
00061
00062
00063 void create_insert_node(FibonacciHeap * heap, OrdTreeNode * node, unsigned short key, double value){
00064     create_node(node, key, value);
00065     insert_node(heap, node);
00066 }
00067
00068
00069 void link_trees(FibonacciHeap * heap, OrdTreeNode * child, OrdTreeNode * father){
00070     child->marked = false;
00071     child->right_sibling->left_sibling = child->left_sibling;
00072     child->left_sibling->right_sibling = child->right_sibling;
00073     child->is_root = false;
00074     heap->num_trees--;
00075
00076     child->parent = father;
00077     father->num_children++;
00078     if(father->num_children == 1){
00079         father->head_child_list = child;
00080         father->tail_child_list = child;
00081         child->left_sibling = child;
00082         child->right_sibling = child;
00083     }
00084     else{
00085         father->tail_child_list->right_sibling = child;
00086         father->head_child_list->left_sibling = child;
00087         child->left_sibling = father->tail_child_list;
00088         child->right_sibling = father->head_child_list;
00089         father->tail_child_list = child;
00090     }
00091 }
00092 }
00093
00094
00095 void swap_roots(FibonacciHeap * heap, OrdTreeNode * node1, OrdTreeNode * node2){
00096     if(node1->key == node2->key){
00097         return;
00098     }
00099     else{
00100
00101         OrdTreeNode * node1_left = node1->left_sibling;
00102         OrdTreeNode * node1_right = node1->right_sibling;
00103         OrdTreeNode * node2_left = node2->left_sibling;
00104         OrdTreeNode * node2_right = node2->right_sibling;
00105
00106         node1->left_sibling = node2_left;
00107         node1->right_sibling = node2_right;
00108         node2->left_sibling = node1_left;
00109         node2->right_sibling = node1_right;
00110     }
00111 }
00112
00113
00114 void consolidate(FibonacciHeap * heap){
00115
00116     int dimension = ((int) ceil(log(heap->num_nodes) / log(2))) + 2;
00117     OrdTreeNode * degree_list [dimension] ;
00118
00119     for (int i = 0; i < dimension; i++) {
00120         degree_list[i] = NULL;
00121     }
00122
00123     OrdTreeNode * iter = heap->head_tree_list;
00124     int num_it = heap->num_trees;
00125
00126     for (int i = 0; i < num_it; i++) {
00127         OrdTreeNode * deg_i_node = iter;

```

```

00128     unsigned short deg = deg_i_node->num_children;
00129     iter = iter->right_sibling;
00130
00131     while (degree_list[deg] != NULL) {
00132
00133         OrdTreeNode * same_deg_node = degree_list[deg];
00134
00135         if (deg_i_node->value > same_deg_node->value) {
00136             OrdTreeNode * temp = deg_i_node;
00137             deg_i_node = same_deg_node;
00138             same_deg_node = temp;
00139             //iter = deg_i_node;
00140         }
00141
00142         link_trees(heap, same_deg_node, deg_i_node);
00143         // since same_deg_node is now a child of deg_i_node, it is no longer a root
00144         degree_list[deg] = NULL;
00145         deg++;
00146     }
00147     degree_list[deg] = deg_i_node;
00148 }
00149
00150 heap->min_root = NULL;
00151
00152 for (int i = 0; i < dimension; i++) {
00153     if (degree_list[i] != NULL) {
00154         if (heap->min_root == NULL) {
00155             heap->min_root = degree_list[i];
00156             heap->head_tree_list = degree_list[i];
00157             heap->tail_tree_list = degree_list[i];
00158         }
00159         else {
00160             if (heap->min_root->value > degree_list[i]->value) {
00161                 heap->min_root = degree_list[i];
00162             }
00163             heap->tail_tree_list->right_sibling = degree_list[i];
00164             heap->head_tree_list->left_sibling = degree_list[i];
00165             degree_list[i]->left_sibling = heap->tail_tree_list;
00166             degree_list[i]->right_sibling = heap->head_tree_list;
00167             heap->tail_tree_list = degree_list[i];
00168         }
00169     }
00170 }
00171
00172 }
00173
00174
00175 int extract_min(FibonacciHeap * heap) {
00176
00177     int min_pos;
00178
00179     if (heap->min_root != NULL) {
00180
00181         min_pos = heap->min_root->key;
00182
00183         OrdTreeNode *child = heap->min_root->head_child_list;
00184         unsigned short num_children = heap->min_root->num_children;
00185
00186         for (unsigned short i = 0; i < num_children; i++) {
00187             child->parent = NULL;
00188             child->is_root = true;
00189             child->marked = false;
00190
00191             if (num_children - i == 1) {
00192                 heap->min_root->head_child_list = NULL;
00193                 heap->min_root->tail_child_list = NULL;
00194             }
00195             else {
00196                 if (heap->min_root->head_child_list->key == child->key) {
00197                     heap->min_root->head_child_list = child->right_sibling;
00198                 }
00199                 if (heap->min_root->tail_child_list->key == child->key) {
00200                     heap->min_root->tail_child_list = child->left_sibling;
00201                 }
00202                 child->left_sibling->right_sibling = child->right_sibling;
00203                 child->right_sibling->left_sibling = child->left_sibling;
00204             }
00205
00206             heap->tail_tree_list->right_sibling = child;
00207             heap->head_tree_list->left_sibling = child;
00208             child->left_sibling = heap->tail_tree_list;
00209             child->right_sibling = heap->head_tree_list;
00210             heap->tail_tree_list = child;
00211
00212             heap->num_trees++;
00213
00214             child = heap->min_root->head_child_list;

```



```

00215     heap->min_root->num_children--;
00216 }
00217
00218     heap->num_trees--;
00219
00220     if(heap->head_tree_list->key == heap->min_root->key){
00221         heap->head_tree_list = heap->head_tree_list->right_sibling;
00222     }
00223     if(heap->tail_tree_list->key == heap->min_root->key){
00224         heap->tail_tree_list = heap->tail_tree_list->left_sibling;
00225     }
00226
00227     heap->min_root->left_sibling->right_sibling = heap->min_root->right_sibling;
00228     heap->min_root->right_sibling->left_sibling = heap->min_root->left_sibling;
00229
00230     if (heap->min_root->key == heap->min_root->right_sibling->key) {
00231         // the min root is the only tree in the heap, and it has no children, so the heap is now
empty
00232         heap->min_root = NULL;
00233         heap->tail_tree_list = NULL;
00234         heap->head_tree_list = NULL;
00235         heap->num_trees = 0;
00236     }
00237     else{
00238         // choose a new min root, arbitrarily
00239         heap->min_root = heap->min_root->right_sibling;
00240         // re-consolidate the heap
00241         consolidate(heap);
00242     }
00243     heap->num_nodes--;
00244 } else{
00245     min_pos = -1;
00246 }
00247 return min_pos;
00248 }
00249
00250
00251 static void cut(FibonacciHeap * heap, OrdTreeNode * node, OrdTreeNode * parent){
00252     node->parent = NULL;
00253     node->is_root = true;
00254     node->marked = false;
00255     parent->num_children--;
00256     if(parent->num_children == 0){
00257         parent->head_child_list = NULL;
00258         parent->tail_child_list = NULL;
00259     }
00260     else{
00261         if(parent->head_child_list->key == node->key){
00262             parent->head_child_list = node->right_sibling;
00263         }
00264         if(parent->tail_child_list->key == node->key){
00265             parent->tail_child_list = node->left_sibling;
00266         }
00267         node->left_sibling->right_sibling = node->right_sibling;
00268         node->right_sibling->left_sibling = node->left_sibling;
00269     }
00270
00271     heap->tail_tree_list->right_sibling = node;
00272     heap->head_tree_list->left_sibling = node;
00273     node->left_sibling = heap->tail_tree_list;
00274     node->right_sibling = heap->head_tree_list;
00275     heap->tail_tree_list = node;
00276     heap->num_trees++;
00277 }
00278
00279
00280 static void cascading_cut(FibonacciHeap * heap, OrdTreeNode * node){
00281     if(node->parent != NULL){
00282         if(!node->marked){
00283             node->marked = true;
00284         }
00285         else{
00286             OrdTreeNode * parent = node->parent;
00287             cut(heap, node, parent);
00288             cascading_cut(heap, parent);
00289         }
00290     }
00291 }
00292
00293
00294 void decrease_value(FibonacciHeap * heap, OrdTreeNode * node, double new_value){
00295     if(new_value > node->value){
00296         fprintf(stderr, "Error: new value is greater than current value\n");
00297         exit(EXIT_FAILURE);
00298     }
00299     else{
00300         node->value = new_value;

```

```

00301         if (node->parent != NULL && node->value < node->parent->value) {
00302             OrdTreeNode * parent = node->parent;
00303             cut(heap, node, node->parent);
00304             cascading_cut(heap, parent);
00305         }
00306         if (node->value < heap->min_root->value) {
00307             heap->min_root = node;
00308         }
00309     }
00310 }
00311
00312 void delete_node(FibonacciHeap * heap, OrdTreeNode * node) {
00313     decrease_value(heap, node, -FLT_MAX);
00314     extract_min(heap);
00315 }

```

## 9.29 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/fibonacci\_heap.h File Reference

This file contains the declaration of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem.

```

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

```

### Classes

- struct [OrdTreeNode](#)  
A Heap-ordered Tree [Node](#) where the key of the parent is  $\leq$  the key of its children.
- struct [FibonacciHeap](#)  
The Fibonacci Heap datastructure as collection of Heap-ordered Trees.

### Macros

- #define [BRANCHANDBOUND1TREE\\_FIBONACCI\\_HEAP\\_H](#)

### Typedefs

- typedef struct [OrdTreeNode](#) [OrdTreeNode](#)  
A Heap-ordered Tree [Node](#) where the key of the parent is  $\leq$  the key of its children.
- typedef struct [FibonacciHeap](#) [FibonacciHeap](#)  
The Fibonacci Heap datastructure as collection of Heap-ordered Trees.

## Functions

- void `create_fibonacci_heap` (`FibonacciHeap` \*heap)  
*Create an empty Fibonacci Heap.*
- void `create_node` (`OrdTreeNode` \*node, unsigned short key, double value)  
*Create a `Node` with a given key and value.*
- void `insert_node` (`FibonacciHeap` \*heap, `OrdTreeNode` \*node)  
*Insert a `Node` in the Fibonacci Heap.*
- void `create_insert_node` (`FibonacciHeap` \*heap, `OrdTreeNode` \*node, unsigned short key, double value)  
*A wrapper function to create a `Node` and insert it in the Fibonacci Heap.*
- int `extract_min` (`FibonacciHeap` \*heap)  
*Extract the minimum `Node` from the Fibonacci Heap.*
- void `decrease_value` (`FibonacciHeap` \*heap, `OrdTreeNode` \*node, double new\_value)  
*Decrease the value of a `Node` in the Fibonacci Heap.*

### 9.29.1 Detailed Description

This file contains the declaration of the Fibonacci Heap datastructure for the Minimum Spanning Tree problem.

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file `fibonacci_heap.h`.

### 9.29.2 Macro Definition Documentation

#### 9.29.2.1 BRANCHANDBOUND1TREE\_FIBONACCI\_HEAP\_H

```
#define BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H
```

Definition at line 22 of file `fibonacci_heap.h`.

### 9.29.3 Typedef Documentation

### 9.29.3.1 FibonacciHeap

```
typedef struct FibonacciHeap FibonacciHeap
```

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

### 9.29.3.2 OrdTreeNode

```
typedef struct OrdTreeNode OrdTreeNode
```

A Heap-ordered Tree [Node](#) where the key of the parent is  $\leq$  the key of its children.

## 9.29.4 Function Documentation

### 9.29.4.1 create\_fibonacci\_heap()

```
void create_fibonacci_heap (
    FibonacciHeap * heap )
```

Create an empty Fibonacci Heap.

#### Parameters

<i>heap</i>	The Fibonacci Heap to be created.
-------------	-----------------------------------

Definition at line 16 of file [fibonacci\\_heap.c](#).

### 9.29.4.2 create\_insert\_node()

```
void create_insert_node (
    FibonacciHeap * heap,
    OrdTreeNode * node,
    unsigned short key,
    double value )
```

A wrapper function to create a [Node](#) and insert it in the Fibonacci Heap.

#### Parameters

<i>heap</i>	The Fibonacci Heap where the <a href="#">Node</a> will be inserted.
<i>node</i>	The <a href="#">Node</a> to be created and inserted.
<i>key</i>	The key of the <a href="#">Node</a> .
<i>value</i>	The value of the <a href="#">Node</a> .

Definition at line 63 of file [fibonacci\\_heap.c](#).

#### 9.29.4.3 create\_node()

```
void create_node (
    OrdTreeNode * node,
    unsigned short key,
    double value )
```

Create a [Node](#) with a given key and value.

##### Parameters

<i>node</i>	The <a href="#">Node</a> to be created.
<i>key</i>	The key of the <a href="#">Node</a> .
<i>value</i>	The value of the <a href="#">Node</a> .

Definition at line 25 of file [fibonacci\\_heap.c](#).

#### 9.29.4.4 decrease\_value()

```
void decrease_value (
    FibonacciHeap * heap,
    OrdTreeNode * node,
    double new_value )
```

Decrease the value of a [Node](#) in the Fibonacci Heap.

If the new value is still greater than the parent's value, nothing happens. Otherwise, the [Node](#) becomes a root. If the parent is marked, and is not a root, it becomes a root. This process is repeated until a parent is not marked or is a root.

##### Parameters

<i>heap</i>	The Fibonacci Heap where the <a href="#">Node</a> is.
<i>node</i>	The <a href="#">Node</a> whose value has to be decreased.
<i>new_value</i>	The new value of the <a href="#">Node</a> .

Definition at line 294 of file [fibonacci\\_heap.c](#).

#### 9.29.4.5 extract\_min()

```
int extract_min (
    FibonacciHeap * heap )
```

Extract the minimum [Node](#) from the Fibonacci Heap.

All the children of the minimum [Node](#) become new roots. The new minimum has to be found and, by doing so, the Heap is re-ordered to maintain the Heap property and minimize the height of the Heap-ordered Trees.

#### Parameters

<i>heap</i>	The Fibonacci Heap where the <a href="#">Node</a> will be extracted.
-------------	--

#### Returns

The key of the minimum [Node](#) if the Heap is not empty, -1 otherwise.

Definition at line 175 of file [fibonacci\\_heap.c](#).

#### 9.29.4.6 insert\_node()

```
void insert_node (
    FibonacciHeap * heap,
    OrdTreeNode * node )
```

Insert a [Node](#) in the Fibonacci Heap.

#### Parameters

<i>heap</i>	The Fibonacci Heap where the <a href="#">Node</a> will be inserted.
<i>node</i>	The <a href="#">Node</a> to be inserted.

Definition at line 38 of file [fibonacci\\_heap.c](#).

## 9.30 fibonacci\_heap.h

[Go to the documentation of this file.](#)

```
00001
00012 #pragma once
00013 #include <stdlib.h>
00014 #include <stdio.h>
00015 #include <stddef.h>
00016 #include <string.h>
00017 #include <assert.h>
00018 #include <stdbool.h>
00019 #include <float.h>
00020 #include <math.h>
00021 #ifndef BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H
00022 #define BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H
00023
00024
00026 typedef struct OrdTreeNode {
00027     unsigned short key;
00028     double value;
00029     struct OrdTreeNode *parent;
00030
00031     struct OrdTreeNode *left_sibling;
00032     struct OrdTreeNode *right_sibling;
00033 }
```

```

00034     struct OrdTreeNode *head_child_list;
00035     struct OrdTreeNode *tail_child_list;
00036     unsigned short num_children;
00037     bool marked;
00038     bool is_root;
00039 }OrdTreeNode;
00040
00041
00043 typedef struct FibonacciHeap{
00044     OrdTreeNode * min_root;
00045     OrdTreeNode * head_tree_list;
00046     OrdTreeNode * tail_tree_list;
00047     unsigned short num_nodes;
00048     unsigned short num_trees;
00049 }FibonacciHeap;
00050
00051
00053
00056 void create_fibonacci_heap(FibonacciHeap * heap);
00057
00058
00060
00065 void create_node(OrdTreeNode * node, unsigned short key, double value);
00066
00067
00069
00073 void insert_node(FibonacciHeap * heap, OrdTreeNode * node);
00074
00075
00077
00083 void create_insert_node(FibonacciHeap * heap, OrdTreeNode * node, unsigned short key, double value);
00084
00085
00087
00093 int extract_min(FibonacciHeap * heap);
00094
00095
00097
00105 void decrease_value(FibonacciHeap * heap, OrdTreeNode * node, double new_value);
00106
00107
00108 #endif //BRANCHANDBOUND1TREE_FIBONACCI_HEAP_H

```

## 9.31 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/graph.c File Reference

The implementation of the graph data structure.

```

#include "graph.h"
#include "doubly_linked_list/list_iterator.h"

```

### Functions

- void `create_graph` (`Graph` \*graph, `List` \*nodes\_list, `List` \*edges\_list, `GraphKind` kind)  
*Create a new instance of a `Graph` with all the needed parameters.*
- void `create_euclidean_graph` (`Graph` \*graph, `List` \*nodes)  
*Create a new instance of an euclidean graphs only the Nodes are necessary.*
- void `print_graph` (const `Graph` \*G)  
*Print Nodes, Edges and other information of the `Graph`.*

### 9.31.1 Detailed Description

The implementation of the graph data structure.

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [graph.c](#).

### 9.31.2 Function Documentation

#### 9.31.2.1 create\_euclidean\_graph()

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

Create a new instance of an euclidean graphs only the Nodes are necessary.

#### Parameters

<i>nodes</i>	Pointer to the <a href="#">List</a> of Nodes.
<i>graph</i>	Pointer to the <a href="#">Graph</a> to be initialized.

Definition at line [71](#) of file [graph.c](#).

#### 9.31.2.2 create\_graph()

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

Create a new instance of a [Graph](#) with all the needed parameters.



## Parameters

<i>nodes</i>	Pointer to the <a href="#">List</a> of Nodes.
<i>edges</i>	Pointer to the <a href="#">List</a> of Edges.
<i>kind</i>	Type of the <a href="#">Graph</a> .
<i>graph</i>	Pointer to the <a href="#">Graph</a> to be initialized.

Definition at line 18 of file [graph.c](#).

## 9.31.2.3 print\_graph()

```
void print_graph (
    const Graph * graph )
```

Print Nodes, Edges and other information of the [Graph](#).

## Parameters

<i>graph</i>	Pointer to the <a href="#">Graph</a> to be printed.
--------------	---

Definition at line 101 of file [graph.c](#).

## 9.32 graph.c

[Go to the documentation of this file.](#)

```
00001
00014 #include "graph.h"
00015 #include "doubly_linked_list/list_iterator.h"
00016
00017
00018 void create_graph(Graph * graph, List *nodes_list, List *edges_list, GraphKind kind) {
00019     graph->kind = kind;
00020     graph->num_edges = 0;
00021     graph->num_nodes = 0;
00022     graph->orderedEdges = false;
00023     graph->cost = 0;
00024
00025     ListIterator *nodes_iterator = create_list_iterator(nodes_list);
00026     unsigned short numNodes = 0;
00027     for (size_t j = 0; j < nodes_list->size; j++) {
00028         Node *curr = list_iterator_get_next(nodes_iterator);
00029         graph->nodes[numNodes].positionInGraph = numNodes;
00030         graph->nodes[numNodes].num_neighbours = 0;
00031         graph->nodes[numNodes].y = curr->y;
00032         graph->nodes[numNodes].x = curr->x;
00033         graph->num_nodes++;
00034         numNodes++;
00035     }
00036     delete_list_iterator(nodes_iterator);
00037
00038     unsigned short numEdges = 0;
00039     ListIterator *edges_iterator = create_list_iterator(edges_list);
00040     for (size_t i = 0; i < edges_list->size; i++) {
00041         //add the source vertex to the data_structures
00042         Edge * current_edge = list_iterator_get_next(edges_iterator);
00043         unsigned short src = current_edge->src;
00044         unsigned short dest = current_edge->dest;
00045
00046         graph->edges[numEdges].dest = dest;
00047         graph->edges[numEdges].src = src;
```

```

00048     graph->edges[numEdges].prob = current_edge->prob;
00049     graph->edges[numEdges].weight = current_edge->weight;
00050     graph->edges[numEdges].symbol = current_edge->symbol;
00051     graph->edges[numEdges].positionInGraph = numEdges;
00052
00053     graph->nodes[src].neighbours[graph->nodes[src].num_neighbours] = dest;
00054     graph->nodes[src].num_neighbours++;
00055     graph->edges_matrix[src][dest] = graph->edges[numEdges];
00056     graph->cost += current_edge->weight;
00057
00058     graph->edges_matrix[dest][src] = graph->edges_matrix[src][dest];
00059     graph->nodes[dest].neighbours[graph->nodes[dest].num_neighbours] = src;
00060     graph->nodes[dest].num_neighbours++;
00061
00062     numEdges++;
00063     graph->num_edges++;
00064 }
00065 delete_list_iterator(edges_iterator);
00066 del_list(edges_list);
00067 del_list(nodes_list);
00068 }
00069
00070
00071 void create_euclidean_graph(Graph * graph, List *nodes) {
00072     List *edges_list = new_list();
00073
00074     unsigned short z = 0;
00075     Edge edges [MAX_EDGES_NUM];
00076     ListIterator *i_nodes_iterator = create_list_iterator(nodes);
00077     for (size_t i = 0; i < nodes->size; i++) {
00078         Node *node_src = list_iterator_get_next(i_nodes_iterator);
00079         for (size_t j = i + 1; j < nodes->size; j++) {
00080             Node *node_dest = get_list_elem_index(nodes, j);
00081
00082             edges[z].src = node_src->positionInGraph;
00083             edges[z].dest = node_dest->positionInGraph;
00084             edges[z].symbol = z + 1;
00085             edges[z].positionInGraph = z;
00086             edges[z].prob = 0;
00087             edges[z].weight = (float) sqrt(pow(fabs(node_src->x - node_dest->x), 2) +
00088                                     pow(fabs(node_src->y - node_dest->y), 2));
00089             add_elem_list_bottom(edges_list, &edges[z]);
00090             z++;
00091         }
00092     }
00093
00094     delete_list_iterator(i_nodes_iterator);
00095
00096     create_graph(graph, nodes, edges_list, WEIGHTED_GRAPH);
00097 }
00098
00099
00100
00101 void print_graph(const Graph *G) {
00102     printf("Nodes: %i\n", G->num_nodes);
00103     for (int i = 0; i < G->num_nodes; i++) {
00104         Node curr = G->nodes[i];
00105         printf("Node%i:\t(%.1f, %.1f)\t%i neighbours: ", curr.positionInGraph, DBL_DIG-1, curr.x,
00106             DBL_DIG-1, curr.y, curr.num_neighbours);
00107
00108         for (int z = 0; z < curr.num_neighbours; z++) {
00109             printf("%i ", G->nodes[curr.neighbours[z]].positionInGraph);
00110         }
00111         printf("\n");
00112     }
00113
00114     printf("\nCost: %lf\n", G->cost);
00115     printf("\nEdges: %i\n", G->num_edges);
00116
00117     double dim = (log(G->num_nodes) / log(10) + 1) * 2 + 7;
00118     for (unsigned short j = 0; j < G->num_edges; j++) {
00119         char edge_print [(int) dim];
00120         char edge_print_dest [(int) (dim-7)/2];
00121         Edge curr = G->edges[j];
00122         sprintf(edge_print, "%i", curr.src);
00123         strcat(edge_print, " <--> ");
00124         sprintf(edge_print_dest, "%i", curr.dest);
00125         strcat(edge_print, edge_print_dest);
00126
00127         printf("Edge%i:\t%s\tweight = %.1f\tprob = %.1f\n",
00128             curr.symbol,
00129             edge_print,
00130             DBL_DIG-1,
00131             curr.weight,
00132             DBL_DIG-1,
00133             curr.prob);
00134     }

```

```
00134
00135 }
```

## 9.33 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/graph.h File Reference

The data structures to model the [Graph](#).

```
#include "../doubly_linked_list/linked_list.h"
#include "../doubly_linked_list/list_iterator.h"
#include "../doubly_linked_list/list_functions.h"
#include "../problem_settings.h"
```

### Classes

- struct [Node](#)  
*Structure of a [Node](#).*
- struct [Edge](#)  
*Structure of an [Edge](#).*
- struct [Graph](#)  
*Structure of a [Graph](#).*

### Typedefs

- typedef enum [GraphKind](#) [GraphKind](#)  
*Enum to specify the kind of the [Graph](#).*
- typedef struct [Node](#) [Node](#)  
*Structure of a [Node](#).*
- typedef struct [Edge](#) [Edge](#)  
*Structure of an [Edge](#).*
- typedef struct [Graph](#) [Graph](#)  
*Structure of a [Graph](#).*

### Enumerations

- enum [GraphKind](#) { [WEIGHTED\\_GRAPH](#) , [UNWEIGHTED\\_GRAPH](#) }  
*Enum to specify the kind of the [Graph](#).*

### Functions

- void [create\\_graph](#) ([Graph](#) \*graph, [List](#) \*nodes, [List](#) \*edges, [GraphKind](#) kind)  
*Create a new instance of a [Graph](#) with all the needed parameters.*
- void [create\\_euclidean\\_graph](#) ([Graph](#) \*graph, [List](#) \*nodes)  
*Create a new instance of an euclidean graphs only the Nodes are necessary.*
- void [print\\_graph](#) (const [Graph](#) \*graph)  
*Print Nodes, Edges and other information of the [Graph](#).*

### 9.33.1 Detailed Description

The data structures to model the [Graph](#).

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [graph.h](#).

### 9.33.2 Typedef Documentation

#### 9.33.2.1 Edge

```
typedef struct Edge Edge
```

Structure of an [Edge](#).

#### 9.33.2.2 Graph

```
typedef struct Graph Graph
```

Structure of a [Graph](#).

#### 9.33.2.3 GraphKind

```
typedef enum GraphKind GraphKind
```

Enum to specify the kind of the [Graph](#).

#### 9.33.2.4 Node

```
typedef struct Node Node
```

Structure of a [Node](#).

### 9.33.3 Enumeration Type Documentation

#### 9.33.3.1 GraphKind

```
enum GraphKind
```

Enum to specify the kind of the [Graph](#).

## Enumerator

WEIGHTED_GRAPH	The <a href="#">Graph</a> is weighted.
UNWEIGHTED_GRAPH	The <a href="#">Graph</a> is unweighted.

Definition at line 23 of file [graph.h](#).

## 9.33.4 Function Documentation

### 9.33.4.1 `create_euclidean_graph()`

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

Create a new instance of an euclidean graphs only the Nodes are necessary.

## Parameters

<i>nodes</i>	Pointer to the <a href="#">List</a> of Nodes.
<i>graph</i>	Pointer to the <a href="#">Graph</a> to be initialized.

Definition at line 71 of file [graph.c](#).

### 9.33.4.2 `create_graph()`

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

Create a new instance of a [Graph](#) with all the needed parameters.

## Parameters

<i>nodes</i>	Pointer to the <a href="#">List</a> of Nodes.
<i>edges</i>	Pointer to the <a href="#">List</a> of Edges.
<i>kind</i>	Type of the <a href="#">Graph</a> .
<i>graph</i>	Pointer to the <a href="#">Graph</a> to be initialized.

Definition at line 18 of file [graph.c](#).

## 9.33.4.3 print\_graph()

```
void print_graph (
    const Graph * graph )
```

Print Nodes, Edges and other information of the [Graph](#).

## Parameters

<i>graph</i>	Pointer to the <a href="#">Graph</a> to be printed.
--------------	---

Definition at line 101 of file [graph.c](#).

## 9.34 graph.h

[Go to the documentation of this file.](#)

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

## 9.35 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/mfset.c File Reference

This file contains the implementation of the Merge-Find [Set](#) datastructure for the Minimum Spanning Tree problem.

```
#include "mfset.h"
```

### Functions

- void [create\\_forest\\_constrained](#) ([Forest](#) \*forest, const [Node](#) \*nodes, unsigned short num\_nodes, unsigned short candidateId)  
*Create a new [Forest](#) with n [Sets](#), each [Set](#) containing a [Node](#), with constraints.*
- void [create\\_forest](#) ([Forest](#) \*forest, const [Node](#) \*nodes, unsigned short num\_nodes)  
*Create a new [Forest](#) with n [Sets](#), each [Set](#) containing a [Node](#), without constraints.*
- [Set](#) \* [find](#) ([Set](#) \*set)  
*Find the root of a [Set](#).*
- void [merge](#) ([Set](#) \*set1, [Set](#) \*set2)  
*Merge two [Sets](#) in the [Forest](#) if they are not already in the same [Set](#).*
- void [print\\_forest](#) (const [Forest](#) \*forest)  
*Print all the [Forest](#).*

### 9.35.1 Detailed Description

This file contains the implementation of the Merge-Find [Set](#) datastructure for the Minimum Spanning Tree problem.

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [mfset.c](#).

### 9.35.2 Function Documentation

#### 9.35.2.1 create\_forest()

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

Create a new [Forest](#) with n [Sets](#), each [Set](#) containing a [Node](#), without constraints.



## Parameters

<i>nodes</i>	Pointer to the <a href="#">List</a> of Nodes.
<i>num_nodes</i>	Number of Nodes in the <a href="#">List</a> .
<i>forest</i>	Pointer to the <a href="#">Forest</a> to be initialized.

Definition at line 31 of file [mfset.c](#).

### 9.35.2.2 create\_forest\_constrained()

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

Create a new [Forest](#) with n Sets, each [Set](#) containing a [Node](#), with constraints.

The candidateId [Node](#) is not added to the [Forest](#) because for the 1-tree I need a [MST](#) on the remaining Nodes.

## Parameters

<i>nodes</i>	Pointer to the <a href="#">List</a> of Nodes.
<i>num_nodes</i>	Number of Nodes in the <a href="#">List</a> .
<i>candidateId</i>	Id of the <a href="#">Node</a> in the <a href="#">List</a> to be excluded from the <a href="#">Forest</a> .
<i>forest</i>	Pointer to the <a href="#">Forest</a> to be initialized.

Definition at line 17 of file [mfset.c](#).

### 9.35.2.3 find()

```
Set * find (
    Set * set )
```

Find the root of a [Set](#).

Complexity:  $O(\log n)$ , only a path in the tree is traversed. The parent [Set](#) of all the Nodes in the path are updated to point to the root, to reduce the complexity of the next find operations.

## Parameters

<i>set</i>	Pointer to the <a href="#">Set</a> .
------------	--------------------------------------

### Returns

Pointer to the root of the [Set](#).

Definition at line 44 of file [mfset.c](#).

#### 9.35.2.4 merge()

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

Merge two Sets in the [Forest](#) if they are not already in the same [Set](#).

The [Set](#) with the highest rank is the parent of the other. This is done to let the find operation run in  $O(\log n)$  time.  
Complexity:  $O(\log n_1 + \log n_2)$

### Parameters

<i>set1</i>	Pointer to the first <a href="#">Set</a> .
<i>set2</i>	Pointer to the second <a href="#">Set</a> .

Definition at line 53 of file [mfset.c](#).

#### 9.35.2.5 print\_forest()

```
void print_forest (
    const Forest * forest )
```

Print all the [Forest](#).

Used for debugging purposes.

### Parameters

<i>forest</i>	Pointer to the <a href="#">Forest</a> .
---------------	---

Definition at line 72 of file [mfset.c](#).

## 9.36 mfset.c

[Go to the documentation of this file.](#)

```
00001
00014 #include "mfset.h"
00015
00016
```

```

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 ,%2d\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 }

```

## 9.37 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/mfset.h File Reference

This file contains the declaration of the Merge-Find [Set](#) datastructure for the Minimum Spanning Tree problem.

```
#include "graph.h"
```

## Classes

- struct [Set](#)  
*A [Set](#) is a node in the [Forest](#).*
- struct [Forest](#)  
*A [Forest](#) is a list of [Sets](#).*

## Typedefs

- typedef struct [Set](#) [Set](#)  
*A [Set](#) is a node in the [Forest](#).*
- typedef struct [Forest](#) [Forest](#)  
*A [Forest](#) is a list of [Sets](#).*

## Functions

- void [create\\_forest](#) ([Forest](#) \*forest, const [Node](#) \*nodes, unsigned short num\_nodes)  
*Create a new [Forest](#) with  $n$  [Sets](#), each [Set](#) containing a [Node](#), without constraints.*
- void [create\\_forest\\_constrained](#) ([Forest](#) \*forest, const [Node](#) \*nodes, unsigned short num\_nodes, unsigned short candidateId)  
*Create a new [Forest](#) with  $n$  [Sets](#), each [Set](#) containing a [Node](#), with constraints.*
- void [merge](#) ([Set](#) \*set1, [Set](#) \*set2)  
*Merge two [Sets](#) in the [Forest](#) if they are not already in the same [Set](#).*
- [Set](#) \* [find](#) ([Set](#) \*set)  
*Find the root of a [Set](#).*
- void [print\\_forest](#) (const [Forest](#) \*forest)  
*Print all the [Forest](#).*

### 9.37.1 Detailed Description

This file contains the declaration of the Merge-Find [Set](#) datastructure for the Minimum Spanning Tree problem.

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [mfset.h](#).

## 9.37.2 Typedef Documentation

### 9.37.2.1 Forest

```
typedef struct Forest Forest
```

A [Forest](#) is a list of Sets.

### 9.37.2.2 Set

```
typedef struct Set Set
```

A [Set](#) is a node in the [Forest](#).

## 9.37.3 Function Documentation

### 9.37.3.1 create\_forest()

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

Create a new [Forest](#) with n Sets, each [Set](#) containing a [Node](#), without constraints.

#### Parameters

<i>nodes</i>	Pointer to the <a href="#">List</a> of Nodes.
<i>num_nodes</i>	Number of Nodes in the <a href="#">List</a> .
<i>forest</i>	Pointer to the <a href="#">Forest</a> to be initialized.

Definition at line 31 of file [mfset.c](#).

### 9.37.3.2 create\_forest\_constrained()

```
void create_forest_constrained (
    Forest * forest,
    const Node * nodes,
```

```

    unsigned short num_nodes,
    unsigned short candidateId )

```

Create a new [Forest](#) with n Sets, each [Set](#) containing a [Node](#), with constraints.

The candidateId [Node](#) is not added to the [Forest](#) because for the 1-tree I need a [MST](#) on the remaining Nodes.

#### Parameters

<i>nodes</i>	Pointer to the <a href="#">List</a> of Nodes.
<i>num_nodes</i>	Number of Nodes in the <a href="#">List</a> .
<i>candidateId</i>	Id of the <a href="#">Node</a> in the <a href="#">List</a> to be excluded from the <a href="#">Forest</a> .
<i>forest</i>	Pointer to the <a href="#">Forest</a> to be initialized.

Definition at line 17 of file [mfset.c](#).

#### 9.37.3.3 find()

```

Set * find (
    Set * set )

```

Find the root of a [Set](#).

Complexity:  $O(\log n)$ , only a path in the tree is traversed. The parent [Set](#) of all the Nodes in the path are updated to point to the root, to reduce the complexity of the next find operations.

#### Parameters

<i>set</i>	Pointer to the <a href="#">Set</a> .
------------	--------------------------------------

#### Returns

Pointer to the root of the [Set](#).

Definition at line 44 of file [mfset.c](#).

#### 9.37.3.4 merge()

```

void merge (
    Set * set1,
    Set * set2 )

```

Merge two Sets in the [Forest](#) if they are not already in the same [Set](#).

The [Set](#) with the highest rank is the parent of the other. This is done to let the find operation run in  $O(\log n)$  time. Complexity:  $O(\log n_1 + \log n_2)$

## Parameters

<i>set1</i>	Pointer to the first <a href="#">Set</a> .
<i>set2</i>	Pointer to the second <a href="#">Set</a> .

Definition at line 53 of file [mfset.c](#).

9.37.3.5 `print_forest()`

```
void print_forest (
    const Forest * forest )
```

Print all the [Forest](#).

Used for debugging purposes.

## Parameters

<i>forest</i>	Pointer to the <a href="#">Forest</a> .
---------------	---

Definition at line 72 of file [mfset.c](#).

9.38 `mfset.h`

[Go to the documentation of this file.](#)

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

## 9.39 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/mst.c File Reference

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

```
#include "mst.h"
```

### Functions

- void `create_mst` (`MST *mst`, const `Node *nodes`, unsigned short `num_nodes`)  
*Create a Minimum Spanning Tree from a set of Nodes.*
- void `add_edge` (`MST *tree`, const `Edge *edge`)  
*Add an `Edge` to the `MST`.*
- void `print_mst` (const `MST *tree`)  
*Print the `MST`, printing all the information it contains.*
- void `print_mst_original_weight` (const `MST *tree`, const `Graph *graph`)  
*Print the `MST`, printing all the information it contains.*

### 9.39.1 Detailed Description

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

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file `mst.c`.

### 9.39.2 Function Documentation

#### 9.39.2.1 `add_edge()`

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

Add an `Edge` to the `MST`.



## Parameters

<i>tree</i>	The Minimum Spanning Tree.
<i>edge</i>	The <a href="#">Edge</a> to add.

Definition at line [38](#) of file [mst.c](#).

### 9.39.2.2 create\_mst()

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

Create a Minimum Spanning Tree from a set of Nodes.

## Parameters

<i>mst</i>	The Minimum Spanning Tree to be initialized.
<i>nodes</i>	The set of Nodes.
<i>num_nodes</i>	The number of Nodes.

Definition at line [17](#) of file [mst.c](#).

### 9.39.2.3 print\_mst()

```
void print_mst (
    const MST * mst )
```

Print the [MST](#), printing all the information it contains.

## Parameters

<i>tree</i>	The Minimum Spanning Tree.
-------------	----------------------------

Definition at line [70](#) of file [mst.c](#).

### 9.39.2.4 print\_mst\_original\_weight()

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

Print the [MST](#), printing all the information it contains.

This method is used to print a [Tree](#) with original. [Edge](#) weights, since in the branch and bound algorithm, with the dual procedure the [Edge](#) weights are changed.

#### Parameters

<i>tree</i>	The Minimum Spanning Tree.
<i>graph</i>	The <a href="#">Graph</a> from which the <a href="#">MST</a> was created.

Definition at line 92 of file [mst.c](#).

## 9.40 mst.c

[Go to the documentation of this file.](#)

```

00001
00014 #include "mst.h"
00015
00016
00017 void create_mst(MST * mst, const Node * nodes, unsigned short num_nodes) {
00018     mst->isValid = false;
00019     mst->cost = 0;
00020     mst->num_nodes = num_nodes;
00021     mst->num_edges = 0;
00022     mst->prob = 0;
00023
00024     for (unsigned short i = 0; i < num_nodes; i++) {
00025         mst->nodes[i].positionInGraph = nodes[i].positionInGraph;
00026         mst->nodes[i].x = nodes[i].x;
00027         mst->nodes[i].y = nodes[i].y;
00028         mst->nodes[i].num_neighbours = 0;
00029
00030         for (unsigned short j = i; j < num_nodes; j++) {
00031             mst->edges_matrix[i][j] = -1;
00032             mst->edges_matrix[j][i] = -1;
00033         }
00034     }
00035 }
00036
00037
00038 void add_edge(MST * tree, const Edge * edge){
00039
00040     unsigned short src = edge->src;
00041     unsigned short dest = edge->dest;
00042
00043     tree->edges[tree->num_edges].src = src;
00044     tree->edges[tree->num_edges].dest = dest;
00045     tree->edges[tree->num_edges].weight = edge->weight;
00046     tree->edges[tree->num_edges].symbol = edge->symbol;
00047     tree->edges[tree->num_edges].prob = edge->prob;
00048     tree->edges[tree->num_edges].positionInGraph = tree->num_edges;
00049     tree->nodes[src].neighbours[tree->nodes[src].num_neighbours] = dest;
00050     tree->nodes[src].num_neighbours++;
00051     tree->nodes[dest].neighbours[tree->nodes[dest].num_neighbours] = src;
00052     tree->nodes[dest].num_neighbours++;
00053     tree->edges_matrix[src][dest] = (short) tree->num_edges;
00054     tree->edges_matrix[dest][src] = (short) tree->num_edges;
00055
00056     tree->num_edges++;
00057     tree->cost += edge->weight;
00058
00059     if (HYBRID) {
00060         if (tree->num_edges == 1) {
00061             tree->prob = edge->prob;
00062         }
00063         else {
00064             tree->prob = ((tree->prob * ((float) tree->num_edges - 1)) + edge->prob) / ((float)
tree->num_edges);
00065         }
00066     }
00067 }
00068
00069
00070 void print_mst(const MST * tree){

```

```

00071     printf("\nMST or 1-Tree with cost: %lf and validity = %s\n", tree->cost, tree->isValid ? "TRUE" :
"FALSE");
00072
00073     double dim = (log(tree->num_nodes) / log(10) + 1) * 2 + 7;
00074     for (unsigned short i = 0; i < tree->num_edges; i++) {
00075         char edge_print [(int) dim] ;
00076         char edge_print_dest [(int) (dim-7)/2] ;
00077         const Edge * curr = &tree->edges[i];
00078         sprintf(edge_print, "%i", curr->src);
00079         strcat(edge_print, " <--> ");
00080         sprintf(edge_print_dest, "%i", curr->dest);
00081         strcat(edge_print, edge_print_dest);
00082         printf("Edge%i:\t%s\tweight = %lf\tprob = %lf\n",
00083             curr->symbol,
00084             edge_print,
00085             curr->weight,
00086             curr->prob);
00087     }
00088 }
00089 }
00090
00091
00092 void print_mst_original_weight(const MST * tree, const Graph * graph){
00093     printf("\nMST or 1-Tree with cost: %f and validity = %s\n", tree->cost, tree->isValid ? "TRUE" :
"FALSE");
00094
00095     double dim = (log(tree->num_nodes) / log(10) + 1) * 2 + 7;
00096     for (unsigned short i = 0; i < tree->num_edges; i++) {
00097         char edge_print [(int) dim] ;
00098         char edge_print_dest [(int) (dim-7)/2] ;
00099         const Edge * curr = &tree->edges[i];
00100         sprintf(edge_print, "%i", curr->src);
00101         strcat(edge_print, " <--> ");
00102         sprintf(edge_print_dest, "%i", curr->dest);
00103         strcat(edge_print, edge_print_dest);
00104         printf("Edge%i: %s weight = %f prob = %f\n",
00105             curr->symbol,
00106             edge_print,
00107             graph->edges_matrix[curr->src][curr->dest].weight,
00108             curr->prob);
00109     }
00110 }

```

## 9.41 GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structures/mst.h File Reference

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

```

#include "mfset.h"
#include "fibonacci_heap.h"

```

### Classes

- struct [ConstrainedEdge](#)  
A reduced form of an [Edge](#) in the [Graph](#), with only the source and destination Nodes.
- struct [MST](#)  
Minimum Spanning Tree, or [MST](#), and also a 1-Tree.

### Macros

- #define [BRANCHANDBOUND1TREE\\_MST\\_H](#)

## Typedefs

- typedef struct [ConstrainedEdge](#) [ConstrainedEdge](#)  
*A reduced form of an [Edge](#) in the [Graph](#), with only the source and destination Nodes.*
- typedef struct [MST](#) [MST](#)  
*Minimum Spanning Tree, or [MST](#), and also a 1-Tree.*

## Functions

- void [create\\_mst](#) ([MST](#) \*mst, const [Node](#) \*nodes, unsigned short num\_nodes)  
*Create a Minimum Spanning Tree from a set of Nodes.*
- void [add\\_edge](#) ([MST](#) \*tree, const [Edge](#) \*edge)  
*Add an [Edge](#) to the [MST](#).*
- void [print\\_mst](#) (const [MST](#) \*mst)  
*Print the [MST](#), printing all the information it contains.*
- void [print\\_mst\\_original\\_weight](#) (const [MST](#) \*mst, const [Graph](#) \*graph)  
*Print the [MST](#), printing all the information it contains.*

### 9.41.1 Detailed Description

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

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [mst.h](#).

### 9.41.2 Macro Definition Documentation

#### 9.41.2.1 BRANCHANDBOUND1TREE\_MST\_H

```
#define BRANCHANDBOUND1TREE_MST_H
```

Definition at line 15 of file [mst.h](#).

### 9.41.3 Typedef Documentation

#### 9.41.3.1 ConstrainedEdge

```
typedef struct ConstrainedEdge ConstrainedEdge
```

A reduced form of an [Edge](#) in the [Graph](#), with only the source and destination Nodes.

#### 9.41.3.2 MST

```
typedef struct MST MST
```

Minimum Spanning Tree, or [MST](#), and also a 1-Tree.

### 9.41.4 Function Documentation

#### 9.41.4.1 add\_edge()

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

Add an [Edge](#) to the [MST](#).

##### Parameters

<i>tree</i>	The Minimum Spanning Tree.
<i>edge</i>	The <a href="#">Edge</a> to add.

Definition at line 38 of file [mst.c](#).

#### 9.41.4.2 create\_mst()

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

Create a Minimum Spanning Tree from a set of Nodes.

## Parameters

<i>mst</i>	The Minimum Spanning Tree to be initialized.
<i>nodes</i>	The set of Nodes.
<i>num_nodes</i>	The number of Nodes.

Definition at line 17 of file [mst.c](#).

#### 9.41.4.3 print\_mst()

```
void print_mst (
    const MST * mst )
```

Print the [MST](#), printing all the information it contains.

## Parameters

<i>tree</i>	The Minimum Spanning Tree.
-------------	----------------------------

Definition at line 70 of file [mst.c](#).

#### 9.41.4.4 print\_mst\_original\_weight()

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

Print the [MST](#), printing all the information it contains.

This method is used to print a 1Tree with original. [Edge](#) weights, since in the branch and bound algorithm, with the dual procedure the [Edge](#) weights are changed.

## Parameters

<i>tree</i>	The Minimum Spanning Tree.
<i>graph</i>	The <a href="#">Graph</a> from which the <a href="#">MST</a> was created.

Definition at line 92 of file [mst.c](#).

## 9.42 mst.h

[Go to the documentation of this file.](#)

00001

```

00013 #pragma once
00014 #ifndef BRANCHANDBOUND1TREE_MST_H
00015 #define BRANCHANDBOUND1TREE_MST_H
00016 #include "mfset.h"
00017 #include "fibonacci_heap.h"
00018
00019
00021 typedef struct ConstrainedEdge{
00022     unsigned short src;
00023     unsigned short dest;
00024 }ConstrainedEdge;
00025
00026
00028 typedef struct MST{
00029     bool isValid;
00030     double cost;
00031     double prob;
00032     unsigned short num_nodes;
00033     unsigned short num_edges;
00034     Node nodes [MAX_VERTEX_NUM];
00035     Edge edges [MAX_VERTEX_NUM];
00036     short edges_matrix [MAX_VERTEX_NUM][MAX_VERTEX_NUM];
00037 }MST;
00038
00039
00046 void create_mst(MST* mst, const Node * nodes, unsigned short num_nodes);
00047
00048
00054 void add_edge(MST * tree, const Edge * edge);
00055
00056
00061 void print_mst(const MST * mst);
00062
00063
00070 void print_mst_original_weight(const MST * mst, const Graph * graph);
00071
00072
00073 #endif //BRANCHANDBOUND1TREE_MST_H

```

## 9.43 GraphConvolutionalBranchandBound/src/HybridSolver/main/graph-convnet-tsp/main.py File Reference

### Namespaces

- namespace [main](#)

### Functions

- def [main.compute\\_prob](#) (net, config, dtypeLong, dtypeFloat)
- def [main.write\\_adjacency\\_matrix](#) (graph, y\_probs, x\_edges\_values, nodes\_coord, filepath, num\_nodes, kmedoids\_labels=None)
- def [main.add\\_dummy\\_cities](#) (num\_nodes, model\_size)
- def [main.create\\_temp\\_file](#) (num\_nodes, str\_grap)
- def [main.cluster\\_nodes](#) (graph, k)
- def [main.fix\\_instance\\_size](#) (graph, num\_nodes, model\_size=100)
- def [main.get\\_instance](#) (num\_nodes)
- def [main.main](#) (filepath, num\_nodes, model\_size)

### Variables

- [main.category](#)
- sys [main.filepath](#) = sys.argv[1]
- int [main.num\\_nodes](#) = int(sys.argv[2])
- int [main.model\\_size](#) = int(sys.argv[3])

## 9.44 main.py

[Go to the documentation of this file.](#)

```

00001 """
00002     @file main.py
00003     @author Lorenzo Sciandra
00004     @brief A recombination of code take from: https://github.com/chaitjo/graph-convnet-tsp.
00005     Some functions were created for the purpose of the paper.
00006     @version 1.0.0
00007     @data 2024-05-1
00008     @copyright Copyright (c) 2024, license MIT
00009     Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound
00010 """
00011 import errno
00012 import os
00013 import sys
00014 import time
00015 import numpy as np
00016 import torch
00017 from torch.autograd import Variable
00018 import torch.nn.functional as F
00019 import torch.nn as nn
00020 from sklearn.utils.class_weight import compute_class_weight
00021 # Remove warning
00022 import warnings
00023
00024 warnings.filterwarnings("ignore", category=UserWarning)
00025 from scipy.sparse import SparseEfficiencyWarning
00026
00027 warnings.simplefilter('ignore', SparseEfficiencyWarning)
00028 from config import *
00029 from utils.graph_utils import *
00030 from utils.google_tsp_reader import GoogleTSPReader
00031 from utils.plot_utils import *
00032 from models.gcn_model import ResidualGatedGCNModel
00033 from sklearn_extra.cluster import KMedoids
00034 from utils.model_utils import *
00035
00036
00037 def compute_prob(net, config, dtypeLong, dtypeFloat):
00038     """
00039     This function computes the probability of the edges being in the optimal tour, by running the GCN.
00040     Args:
00041         net: The Graph Convolutional Network.
00042         config: The configuration file, from which the parameters are taken.
00043         dtypeLong: The data type for the long tensors.
00044         dtypeFloat: The data type for the float tensors.
00045     Returns:
00046         y_probs: The probability of the edges being in the optimal tour.
00047         x_edges_values: The distance between the nodes.
00048     """
00049     # Set evaluation mode
00050     net.eval()
00051
00052     # Assign parameters
00053     num_nodes = config.num_nodes
00054     num_neighbors = config.num_neighbors
00055     batch_size = config.batch_size
00056     test_filepath = config.test_filepath
00057
00058     # Load TSP data
00059     dataset = GoogleTSPReader(num_nodes, num_neighbors, batch_size=batch_size, filepath=test_filepath)
00060
00061     # Convert dataset to iterable
00062     dataset = iter(dataset)
00063
00064     # Initially set loss class weights as None
00065     edge_cw = None
00066
00067     y_probs = []
00068
00069     # read the instance number line from the test_filepath
00070     instance = None
00071     lines = []
00072     with open(test_filepath, 'r') as f:
00073         lines = f.readlines()
00074
00075     if lines is None:
00076         raise Exception("The input file is empty.")
00077
00078     instance = lines[0]
00079
00080     if instance is None:
00081         raise Exception("The instance does not exist.")
00082

```



```

00083     instance = instance.split(" output")[0]
00084     instance = [float(x) for x in instance.split(" ")]
00085     # print(config)
00086
00087     with torch.no_grad():
00088
00089         batch = next(dataset)
00090
00091         while batch.nodes_coord.flatten().tolist() != instance:
00092             batch = next(dataset)
00093
00094         x_edges = Variable(torch.LongTensor(batch.edges).type(dtypeLong), requires_grad=False)
00095         x_edges_values = Variable(torch.FloatTensor(batch.edges_values).type(dtypeFloat),
requires_grad=False)
00096         x_nodes = Variable(torch.LongTensor(batch.nodes).type(dtypeLong), requires_grad=False)
00097         x_nodes_coord = Variable(torch.FloatTensor(batch.nodes_coord).type(dtypeFloat),
requires_grad=False)
00098         y_edges = Variable(torch.LongTensor(batch.edges_target).type(dtypeLong), requires_grad=False)
00099
00100         # Compute class weights (if uncomputed)
00101         if type(edge_cw) != torch.Tensor:
00102             edge_labels = y_edges.cpu().numpy().flatten()
00103             edge_cw = compute_class_weight("balanced", classes=np.unique(edge_labels), y=edge_labels)
00104
00105         y_preds, _ = net.forward(x_edges, x_edges_values, x_nodes, x_nodes_coord, y_edges, edge_cw)
00106         y = F.softmax(y_preds, dim=3)
00107         # y_bins = y.argmax(dim=3)
00108         y_probs = y[:, :, :, 1]
00109
00110         nodes_coord = batch.nodes_coord.flatten().tolist()
00111
00112         return y_probs, x_edges_values, nodes_coord
00113
00114
00115 def write_adjacency_matrix(graph, y_probs, x_edges_values, nodes_coord, filepath, num_nodes,
kmedoids_labels=None):
00116     """
00117     This function writes the adjacency matrix to a file.
00118     The file is in the format:
00119         cities: (x1, y1);(x2, y2);...;(xn, yn)
00120         adjacency matrix:
00121         (0.0, 0.0);(0.23, 0.9);...;(0.15, 0.56)
00122         ...
00123         (0.23, 0.9);(0.3, 0.59);...;(0.0, 0.0)
00124         where each entry is (distance, probability)
00125     If needed adjusts the size of the graph when the model size is different from the number of nodes
in the instance.
00126     Args:
00127         graph: The set of nodes in the graph.
00128         y_probs: The probability of the edges being in the optimal tour.
00129         x_edges_values: The weight of the edges.
00130         nodes_coord: The nodes coordinates used in the GCN.
00131         filepath: The path to the file where the adjacency matrix will be written.
00132         num_nodes: The number of nodes in the TSP instance.
00133         kmedoids_labels: The labels of the k-medoids clustering.
00134     """
00135
00136     model_size = y_probs.shape[1]
00137     y_probs = y_probs.flatten().numpy()
00138     x_edges_values = x_edges_values.flatten().numpy()
00139
00140     # stack the arrays horizontally and convert to string data type
00141     arr_combined = np.stack((x_edges_values, y_probs), axis=1).astype('U')
00142
00143     if num_nodes < model_size:
00144         nodes_coord = nodes_coord[:num_nodes*2]
00145         final_arr = []
00146         for i in range(num_nodes):
00147             for j in range(num_nodes):
00148                 final_arr.append(arr_combined[i * model_size + j])
00149
00150             for j in range(num_nodes, model_size):
00151                 if i != j-num_nodes:
00152                     if arr_combined[i * model_size + j][1] > final_arr[i * num_nodes + j -
num_nodes][1]:
00153                         final_arr[i * num_nodes + j - num_nodes][1] = arr_combined[i * model_size +
j][1]
00154
00155         arr_combined = np.array(final_arr)
00156
00157     elif num_nodes > model_size:
00158         nodes_coord = [[nodes_coord[i], nodes_coord[i + 1]] for i in range(0, len(nodes_coord), 2)]
00159         arr_combined = arr_combined.flatten()
00160         arr_combined = arr_combined.reshape(model_size, model_size, 2).tolist()
00161         j = 0
00162         for node in graph:
00163             if node not in nodes_coord:

```

```

00164         new_row = []
00165         label = kmedoids_labels[j]
00166
00167         for i in range(len(nodes_coord)):
00168             distance = np.linalg.norm(np.array(node) - np.array(nodes_coord[i]))
00169             prob = 0.0 #arr_combined[label][i][1]
00170             arr_combined[i].append([distance, prob])
00171             new_row.append([distance, prob])
00172
00173         new_row.append([0.0, 0.0])
00174         arr_combined.append(new_row)
00175         nodes_coord.append(node)
00176
00177         j += 1
00178
00179         arr_combined = np.array(arr_combined).flatten().tolist()
00180         arr_combined = [[arr_combined[i], arr_combined[i + 1]] for i in range(0, len(arr_combined),
2)]
00181         nodes_coord = np.array(nodes_coord).flatten().tolist()
00182
00183         nodes_coord = ";".join(["({nodes_coord[i]}, {nodes_coord[i + 1]})" for i in range(0,
len(nodes_coord), 2)])
00184         arr_strings = np.array(['({}, {});'.format(x[0], x[1]) for x in arr_combined])
00185
00186         with open(filepath, 'w') as f:
00187             f.write("%s\n" % nodes_coord)
00188             edge = 0
00189             for item in arr_strings:
00190                 if (edge + 1) % num_nodes == 0:
00191                     f.write("%s\n" % item)
00192                 else:
00193                     f.write("%s" % item)
00194                 edge += 1
00195             f.flush()
00196             os.fsync(f.fileno())
00197
00198
00199 def add_dummy_cities(num_nodes, model_size):
00200     """
00201     This function adds dummy cities to the graph instance. The dummy cities are randomly generated and
00202     added to the graph instance and the new instance is saved in a temporary file.
00203     Args:
00204         num_nodes: The number of nodes of the graph instance.
00205         model_size: The size of the Graph Convolutional Network to use.
00206     """
00207
00208     num_dummy_cities = model_size - num_nodes
00209     filepath = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"
00210     graph_str = None
00211
00212     with open(filepath, "r") as f:
00213         lines = f.readlines()
00214         graph_str = lines[0]
00215
00216     graph_str = graph_str.split(" output")[0]
00217     if graph_str is None:
00218         raise Exception("The input file is empty.")
00219
00220     nodes = graph_str.split(" ")
00221     graph = [[float(nodes[i]), float(nodes[i + 1])] for i in range(0, len(nodes), 2)]
00222     rr_index = 0
00223     for i in range(num_dummy_cities):
00224         find = False
00225         x = None
00226         y = None
00227         while not find:
00228             values = np.random.randint(1, 9, 2)
00229             signs = np.random.choice([-1, 1], 2)
00230             x = graph[rr_index][0] + signs[0] * values[0] * 0.000000000000001
00231             y = graph[rr_index][1] + signs[1] * values[1] * 0.000000000000001
00232             if [x, y] not in graph:
00233                 find = True
00234
00235         graph.append([x, y])
00236         graph_str += " " + str(x) + " " + str(y)
00237         rr_index = (rr_index + 1) % num_nodes
00238
00239     seq = np.linspace(1, model_size, model_size, dtype=int)
00240     seq_str = ""
00241     for s in seq:
00242         seq_str += str(s) + " "
00243
00244     seq_str += "1"
00245     graph_str += " output " + seq_str
00246
00247     with open(filepath, 'w+') as file:
00248         file.writelines(graph_str)

```

```

00249         file.flush()
00250         os.fsync(file.fileno())
00251
00252
00253 def create_temp_file(num_nodes, str_grap):
00254     """
00255     Creates a temporary file with the graph instance.
00256     Args:
00257         num_nodes: The number of nodes of the graph instance.
00258         str_grap: The graph instance.
00259     """
00260
00261     filepath = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"
00262
00263     with open(filepath, 'w+') as file:
00264         file.writelines(str_grap)
00265         file.flush()
00266         os.fsync(file.fileno())
00267
00268
00269 def cluster_nodes(graph, k):
00270     """
00271     Applies the k-medoids clustering to the graph.
00272     Args:
00273         graph: The graph to cluster.
00274         k: The number of clusters to create.
00275     Returns:
00276         medoids_str: The medoids of the clusters.
00277         kmedoids.labels_: The labels of the clusters.
00278     """
00279     graph = np.array(graph)
00280     kmedoids = KMedoids(n_clusters=k, method='pam', random_state=42).fit(graph)
00281     medoids = kmedoids.cluster_centers_
00282     medoids_str = " ".join(f"{x} {y}" for x, y in medoids)
00283
00284     return medoids_str, kmedoids.labels_
00285
00286
00287 def fix_instance_size(graph, num_nodes, model_size=100):
00288     """
00289     The function that fixes the instance size with clustering.
00290     It applies the k-medoids clustering to the graph and creates a new instance with the medoids as
00291     the new nodes.
00292     Args:
00293         graph: The graph to fix.
00294         num_nodes: The number of nodes of the graph instance.
00295         model_size: The size of the Graph Convolutional Network to use.
00296     """
00297
00298     new_graph_str = ""
00299     end_str = " output "
00300
00301     print("Need to fix the instance size with clustering")
00302     new_graph_str, kmedoids_labels = cluster_nodes(graph, model_size)
00303
00304     for i in range(1, model_size + 1):
00305         end_str += str(i) + " "
00306
00307     end_str += "1"
00308
00309     create_temp_file(num_nodes, new_graph_str + end_str)
00310     return kmedoids_labels
00311
00312 def get_instance(num_nodes):
00313     """
00314     The function that reads the current instance from the file.
00315     Args:
00316         num_nodes: The number of nodes of the graph instance.
00317     Returns:
00318         graph: The graph instance.
00319     """
00320
00321     lines = None
00322     file_path = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"
00323
00324     with open(file_path, "r") as f:
00325         lines = f.readlines()
00326
00327     if lines is None or len(lines) == 0:
00328         raise Exception(
00329             "The current instance for the number of nodes " + str(num_nodes) + " does not exist.")
00330
00331     str_graph = lines[0]
00332
00333     if "output" in str_graph:
00334         str_graph = str_graph.split(" output")[0]

```

```

00335
00336     str_graph = str_graph.replace("\n", "").strip()
00337     nodes = str_graph.split(" ")
00338     graph = [float(x) for x in nodes]
00339     graph = [[graph[i], graph[i + 1]] for i in range(0, len(graph), 2)]
00340
00341     return graph
00342
00343
00344
00345 def main(filepath, num_nodes, model_size):
00346     """
00347     The function that runs the Graph Convolutional Network and writes the adjacency matrix to a file
00348     for the given input instance.
00349     Args:
00350         filepath: The path to the file where the adjacency matrix will be written.
00351         num_nodes: The number of nodes in the TSP instance.
00352         model_size: The size of the Graph Convolutional Network to use.
00353     """
00354
00355     graph = None
00356     kmedoids_labels = None
00357
00358     if num_nodes < model_size:
00359         add_dummy_cities(num_nodes, model_size)
00360     elif num_nodes > model_size:
00361         graph = get_instance(num_nodes)
00362         kmedoids_labels = fix_instance_size(graph, num_nodes)
00363
00364     config_path = "./logs/tsp" + str(model_size) + "/config.json"
00365     config = get_config(config_path)
00366
00367     config.gpu_id = "0"
00368     config.accumulation_steps = 1
00369
00370     config.val_filepath = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"
00371     config.test_filepath = "data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"
00372
00373     os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
00374     os.environ["CUDA_VISIBLE_DEVICES"] = str(config.gpu_id)
00375
00376     if torch.cuda.is_available():
00377         # print("CUDA available, using GPU ID {}".format(config.gpu_id))
00378         dtypeFloat = torch.cuda.FloatTensor
00379         dtypeLong = torch.cuda.LongTensor
00380         torch.cuda.manual_seed(1)
00381     else:
00382         # print("CUDA not available")
00383         dtypeFloat = torch.FloatTensor
00384         dtypeLong = torch.LongTensor
00385         torch.manual_seed(1)
00386
00387     net = nn.DataParallel(ResidualGatedGCNModel(config, dtypeFloat, dtypeLong))
00388     if torch.cuda.is_available():
00389         net.cuda()
00390
00391     log_dir = f"./logs/{config.expt_name}/"
00392     if torch.cuda.is_available():
00393         checkpoint = torch.load(log_dir + "best_val_checkpoint.tar")
00394     else:
00395         checkpoint = torch.load(log_dir + "best_val_checkpoint.tar", map_location='cpu')
00396     # Load network state
00397     net.load_state_dict(checkpoint['model_state_dict'])
00398     config.batch_size = 1
00399     probs, edges_value, nodes_coord = compute_prob(net, config, dtypeLong, dtypeFloat)
00400     write_adjacency_matrix(graph, probs, edges_value, nodes_coord, filepath, num_nodes,
00401                             kmedoids_labels)
00402
00403 if __name__ == "__main__":
00404     """
00405     Args:
00406         sys.argv[1]: The path to the file where the adjacency matrix will be written.
00407         sys.argv[2]: The number of nodes in the TSP instance.
00408         sys.argv[3]: The dimension of the model.
00409     """
00410
00411     if len(sys.argv) != 4:
00412         print("\nPlease provide the path to the output file to write in, the number of nodes in the
00413 tsp and the "
00414             "instance number to analyze. The format is: "
00415             "<filepath> <number of nodes> <model size>\n")
00416         sys.exit(1)
00417     if not isinstance(sys.argv[1], str) or not isinstance(sys.argv[2], str) or not
00418         isinstance(sys.argv[3], str):
00419         print("Error: The arguments must be strings.")

```

```

00419         sys.exit(1)
00420
00421     filepath = sys.argv[1]
00422     num_nodes = int(sys.argv[2])
00423     model_size = int(sys.argv[3])
00424     main(filepath, num_nodes, model_size)

```

## 9.45 GraphConvolutionalBranchandBound/README.md File Reference

### 9.46 GraphConvolutionalBranchandBound/src/Hybrid↵ Solver/main/graph-convnet-tsp/README.md File Reference

### 9.47 GraphConvolutionalBranchandBound/src/HybridSolver/main/↵ HybridSolver.py File Reference

#### Namespaces

- namespace [HybridSolver](#)

#### Functions

- def [HybridSolver.adjacency\\_matrix](#) (orig\_graph)
- def [HybridSolver.create\\_temp\\_file](#) (num\_nodes, str\_grap)
- def [HybridSolver.get\\_nodes](#) (graph)
- def [HybridSolver.get\\_instance](#) (instance, num\_nodes)
- def [HybridSolver.build\\_c\\_program](#) (build\_directory, num\_nodes, hyb\_mode)
- def [HybridSolver.hybrid\\_solver](#) (num\_instances, num\_nodes, hyb\_mode, gen\_matrix)

#### Variables

- argparse [HybridSolver.parser](#) = argparse.ArgumentParser()
- [HybridSolver.type](#)
- [HybridSolver.str](#)
- [HybridSolver.default](#)
- [HybridSolver.int](#)
- [HybridSolver.action](#)
- argparse [HybridSolver.opts](#) = parser.parse\_args()
- argparse [HybridSolver.gen\\_matrix](#) = False

## 9.48 HybridSolver.py

[Go to the documentation of this file.](#)

```

00001 """
00002     @file: HybridSolver.py
00003     @author Lorenzo Sciandra
00004     @brief First it builds the program in C, specifying the number of nodes to use and whether it is
           in hybrid mode or not.
00005     Then it runs the graph conv net on the instance, and finally it runs the Branch and Bound.
00006     It can be run on a single instance or a range of instances.
00007     The input matrix is generated by the neural network and stored in the data folder. The output is
           stored in the results folder.
00008     @version 1.0.0
00009     @data 2024-05-1
00010     @copyright Copyright (c) 2024, license MIT
00011
00012     Repo: https://github.com/LorenzoSciandra/GraphConvolutionalBranchandBound
00013 """
00014 import subprocess
00015 import argparse
00016 import pprint as pp
00017 import os
00018 import time
00019 import numpy as np
00020
00021
00022 def adjacency_matrix(orig_graph):
00023     """
00024     Calculates the adjacency matrix of the graph.
00025     Args:
00026         orig_graph: The original graph.
00027
00028     Returns:
00029         The adjacency matrix of the graph.
00030     """
00031     adj_matrix = np.zeros((len(orig_graph), len(orig_graph)))
00032
00033     for i in range(0, len(orig_graph)):
00034         for j in range(i + 1, len(orig_graph)):
00035             adj_matrix[i][j] = np.linalg.norm(np.array(orig_graph[i]) - np.array(orig_graph[j]))
00036             adj_matrix[j][i] = adj_matrix[i][j]
00037
00038     return adj_matrix
00039
00040
00041 def create_temp_file(num_nodes, str_grap):
00042     """
00043     Creates a temporary file to store the current instance of the TSP for the neural network.
00044     Args:
00045         num_nodes: The number of nodes in the TSP instance.
00046         str_grap: The string representation of the graph.
00047     """
00048     filepath = "graph-convnet-tsp/data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt"
00049
00050     if not os.path.exists(os.path.dirname(filepath)):
00051         try:
00052             os.makedirs(os.path.dirname(filepath))
00053         except OSError as exc: # Guard against race condition
00054             if exc.errno != errno.EEXIST:
00055                 raise
00056
00057     with open(filepath, 'w+') as file:
00058         file.writelines(str_grap)
00059         file.flush()
00060         os.fsync(file.fileno())
00061
00062
00063 def get_nodes(graph):
00064     """
00065     From a graph, it returns the nodes in a string format.
00066     Args:
00067         graph: The graph to get the nodes from.
00068
00069     Returns:
00070         The nodes in a string format.
00071     """
00072     nodes = ""
00073     i = 0
00074     for node in graph:
00075         nodes += "\t" + str(i) + " : " + str(node[0]) + " " + str(node[1]) + "\n"
00076         i += 1
00077
00078     return nodes
00079
00080

```

```

00081 def get_instance(instance, num_nodes):
00082     """
00083     Gets the instance of the TSP from the file.
00084     Args:
00085         instance: The instance to get.
00086         num_nodes: The number of nodes in the TSP instance.
00087
00088     Returns:
00089         The graph in a list and string format.
00090     """
00091
00092     lines = None
00093     file_path = "graph-convnet-tsp/data/hyb_tsp/test_" + str(num_nodes) + "_nodes.txt"
00094     with open(file_path, "r") as f:
00095         lines = f.readlines()
00096
00097     if lines is None or len(lines) < instance - 1:
00098         raise Exception(
00099             "The instance " + str(instance) + " for the number of nodes " + str(num_nodes) + " does
not exist.")
00100
00101     str_graph = lines[instance - 1]
00102     orig_graph = str_graph
00103
00104     if "output" in str_graph:
00105         str_graph = str_graph.split(" output")[0]
00106
00107     str_graph = str_graph.replace("\n", "").strip()
00108     nodes = str_graph.split(" ")
00109     graph = [float(x) for x in nodes]
00110     graph = [[graph[i], graph[i + 1]] for i in range(0, len(graph), 2)]
00111
00112     return graph, orig_graph
00113
00114
00115 def build_c_program(build_directory, num_nodes, hyb_mode):
00116     """
00117     Builds the C program with the specified number of nodes and whether it is in hybrid mode or not.
00118     Args:
00119         build_directory: The directory where the CMakeLists.txt file is located and where the
executable will be built.
00120         num_nodes: The number of nodes to use in the C program.
00121         hyb_mode: 1 if the program is in hybrid mode, 0 otherwise.
00122     """
00123     source_directory = "../"
00124     cmake_command = [
00125         "cmake",
00126         "-S" + source_directory,
00127         "-B" + build_directory,
00128         "-DCMAKE_BUILD_TYPE=Release",
00129         "-DMAX_VERTEX_NUM=" + str(num_nodes),
00130         "-DHYBRID=" + str(hyb_mode)
00131     ]
00132     # print(cmake_command)
00133     make_command = [
00134         "make",
00135         "-C" + build_directory,
00136         "-j"
00137     ]
00138     try:
00139         subprocess.check_call(cmake_command)
00140         subprocess.check_call(make_command)
00141     except subprocess.CalledProcessError as e:
00142         print("Build failed:")
00143         print(e.output)
00144         raise Exception("Build failed")
00145
00146
00147 def hybrid_solver(num_instances, num_nodes, hyb_mode, gen_matrix):
00148     """
00149     The Graph Convolutional Branch-and-Bound Solver.
00150     Args:
00151         num_instances: The range of instances to run on the Solver.
00152         num_nodes: The number of nodes in each TSP instance.
00153         hyb_mode: True if the program is in hybrid mode, False otherwise.
00154         gen_matrix: True if the adjacency matrix is already generated, False otherwise.
00155     """
00156
00157     model_size = 0
00158     adj_matrix = None
00159
00160     if hyb_mode:
00161         if num_nodes <= 1:
00162             raise Exception("The number of nodes must be greater than 1.")
00163         elif num_nodes <= 20:
00164             model_size = 20
00165         elif num_nodes <= 50:

```

```

00166         model_size = 50
00167     else:
00168         model_size = 100
00169     else:
00170         model_size = num_nodes
00171
00172     build_directory = "../cmake-build/CMakeFiles/BranchAndBound1Tree.dir"
00173     hybrid = 1 if hyb_mode else 0
00174     build_c_program(build_directory, num_nodes, hybrid)
00175
00176     if "-" in num_instances:
00177         instances = num_instances.split("-")
00178         start_instance = 1 if int(instances[0]) == 0 else int(instances[0])
00179         end_instance = int(instances[1])
00180     else:
00181         start_instance = 1
00182         end_instance = int(num_instances)
00183
00184     print("Starting instance: " + str(start_instance))
00185     print("Ending instance: " + str(end_instance))
00186
00187     for i in range(start_instance, end_instance + 1):
00188         start_time = time.time()
00189         graph, str_graph = get_instance(i, num_nodes)
00190         input_file = "../data/AdjacencyMatrix/tsp_" + str(num_nodes) + "_nodes/tsp_test_" + str(i) +
00191             ".csv"
00192         absolute_input_path = os.path.abspath(input_file)
00193
00194         if not os.path.exists(os.path.dirname(absolute_input_path)):
00195             try:
00196                 os.makedirs(os.path.dirname(absolute_input_path))
00197             except OSError as exc: # Guard against race condition
00198                 if exc.errno != errno.EEXIST:
00199                     raise
00200
00201         result_mode = "hybrid" if hyb_mode else "classic"
00202         output_file = "../results/AdjacencyMatrix/tsp_" + str(num_nodes) + "_nodes_" + result_mode + \
00203             + "/tsp_result_" + str(i) + ".txt"
00204
00205         if hyb_mode:
00206             create_temp_file(num_nodes, str_graph)
00207             absolute_python_path = os.path.abspath("../graph-convnet-tsp/main.py")
00208             result = subprocess.run(
00209                 ['python3', absolute_python_path, absolute_input_path, str(num_nodes),
00210                  str(model_size)],
00211                 cwd="../graph-convnet-tsp", check=True)
00212             if result.returncode == 0:
00213                 print('Neural Network completed successfully on instance ' + str(i) + ' / ' +
00214                     str(end_instance))
00215             else:
00216                 print('Neural Network failed on instance ' + str(i) + ' / ' + str(end_instance))
00217
00218         elif gen_matrix:
00219             adj_matrix = adjacency_matrix(graph)
00220             with open(absolute_input_path, "w") as f:
00221                 nodes_coord = ";".join([f"({graph[i][0]}, {graph[i][1]})" for i in range(len(graph))])
00222                 f.write(nodes_coord + "\n")
00223                 for k in range(len(adj_matrix)):
00224                     for j in range(len(adj_matrix[k])):
00225                         f.write(f"({adj_matrix[k][j]}, 0);")
00226                 f.write("\n")
00227
00228         absolute_output_path = os.path.abspath(output_file)
00229         if not os.path.exists(os.path.dirname(absolute_output_path)):
00230             try:
00231                 os.makedirs(os.path.dirname(absolute_output_path))
00232             except OSError as exc: # Guard against race condition
00233                 if exc.errno != errno.EEXIST:
00234                     raise
00235
00236         cmd = [build_directory + "/BranchAndBound1Tree", absolute_input_path, absolute_output_path]
00237         result = subprocess.run(cmd)
00238         if result.returncode == 0:
00239             print('Branch-and-Bound completed successfully on instance ' + str(i) + ' / ' +
00240                 str(end_instance))
00241             else:
00242                 print('Branch-and-Bound failed on instance ' + str(i) + ' / ' + str(end_instance))
00243
00244         end_time = time.time()
00245         cities = get_nodes(graph)
00246
00247         if hyb_mode:
00248             os.remove("graph-convnet-tsp/data/hyb_tsp/test_" + str(num_nodes) + "_nodes_temp.txt")
00249
00250         with open(output_file, "a") as f:
00251             f.write("\nNodes: \n" + cities)
00252             f.write("\nTime taken: " + str(end_time - start_time) + "s\n")
00253             f.flush()

```



```

00249         os.fsync(f.fileno())
00250
00251
00252 if __name__ == "__main__":
00253     """
00254     Args:
00255         --range_instances: The range of instances to run on the Solver.
00256         --num_nodes: The number of nodes in each TSP instance.
00257         --hybrid_mode: If present, the program is in hybrid mode, otherwise it is in classic mode.
00258     """
00259
00260     parser = argparse.ArgumentParser()
00261     parser.add_argument("--range_instances", type=str, default="1-1")
00262     parser.add_argument("--num_nodes", type=int, default=20)
00263     parser.add_argument("--hybrid", action="store_true")
00264     opts = parser.parse_args()
00265
00266     pp.pprint(vars(opts))
00267
00268     gen_matrix = opts.hybrid == False
00269
00270     hybrid_solver(opts.range_instances, opts.num_nodes, opts.hybrid, gen_matrix)

```

## 9.49 GraphConvolutionalBranchandBound/src/Hybrid Solver/main/main.c File Reference

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

```
#include "../test/main_test.h"
```

### Functions

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

*Main function, where you start the program, read the input file and print/write the results.*

### 9.49.1 Detailed Description

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

#### Author

Lorenzo Sciandra

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [main.c](#).

## 9.49.2 Function Documentation

### 9.49.2.1 main()

```
int main (
    int argc,
    char * argv[] )
```

Main function, where you start the program, read the input file and print/write the results.

#### Parameters

<i>argc</i>	The number of arguments passed to the program.
<i>argv</i>	The arguments passed to the program.

#### Returns

0 if the program ends correctly, 1 otherwise.

Definition at line 23 of file [main.c](#).

## 9.50 main.c

[Go to the documentation of this file.](#)

```
00001
00014 #include "../test/main_test.h"
00015
00016
00023 int main(int argc, char *argv[]) {
00024
00025     if (argc != 3) {
00026         perror("Wrong number of arguments");
00027         printf("\nYou need to pass 2 arguments: <input file> <output file>\n");
00028         exit(1);
00029     }
00030
00031
00032     char *input_file = argv[1];
00033     char *output_file = argv[2];
00034
00035     //printf("\nNodes:%d \t\tMode: %s\t\t\n", MAX_VERTEX_NUM, HYBRID ? "Hybrid" : "Classic");
00036
00037     //printf("\nReading from file '%s'\n", input_file);
00038     //printf("\nWriting to file '%s'\n", output_file);
00039
00040     freopen(output_file, "w+", stdout);
00041
00042     //run_all_tests();
00043
00044     static Problem new_problem;
00045
00046     //read_tsp_lib_file(&new_problem.graph, input_file);
00047     read_tsp_csv_file(&new_problem.graph, input_file);
00048
00049     //print_graph(&new_problem.graph);
00050
00051     branch_and_bound(&new_problem);
00052
00053     print_problem();
00054
00055     fclose(stdout);
00056
00057     return 0;
00058 }
```

## 9.51 GraphConvolutionalBranchandBound/src/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** DBL\_MAX  
The maximum number to set the initial value of *Problem* and *SubProblem*.
- #define **PRIM** 1  
The maximum number of *Node* in the *Graph*.
- #define **MAX\_EDGES\_NUM** (MAX\_VERTEX\_NUM \* (MAX\_VERTEX\_NUM - 1) / 2)  
The maximum number of edges in the *Graph*.
- #define **GHOSH\_UB** (sqrt(MAX\_VERTEX\_NUM) \* 1.27f)  
The first upper bound for the problem, see: <https://www.semanticscholar.org/paper/Expected-Travel-Among-Random-Points-in-a-Region-Ghosh/4c395ab42054f4312ad24cb500fb8ca6f7ad>
- #define **TRACE()** fprintf(stderr, "%s (%d): %s\n", \_\_FILE\_\_, \_\_LINE\_\_, \_\_func\_\_)  
Used to debug the code, to check if the execution reaches a certain point.
- #define **EPSILON** (GHOSH\_UB / 1000)  
The first constant used to compare two *SubProblem* in the branch and bound algorithm.
- #define **EPSILON2** (0.1f \* EPSILON)  
The second constant used to compare two *SubProblem* in the branch and bound algorithm.
- #define **BETTER\_PROB** 0.05f  
The third constant used to compare two *SubProblem* in the branch and bound algorithm.
- #define **PROB\_BRANCH** 0.5f  
The way with generate the children of a *SubProblem*.
- #define **TIME\_LIMIT\_SECONDS** 600  
The maximum time to run the algorithm. Default: 10 minutes.
- #define **NUM\_HK\_INITIAL\_ITERATIONS** (((float) MAX\_VERTEX\_NUM \* MAX\_VERTEX\_NUM)/50) + 0.5f + MAX\_VERTEX\_NUM + 15)  
The maximum number of dual iterations for the root of the branch and bound tree.
- #define **NUM\_HK\_ITERATIONS** (((float) MAX\_VERTEX\_NUM / 4) + 5)  
The maximum number of dual iterations for nodes of the branch and bound tree that are not the root.

### 9.51.1 Detailed Description

Contains all the execution settings.

#### Author

Lorenzo Sciandra

Not only MACROs for branch-and-bound, but also for testing and debugging. The two MACROs MAX\_VERTEX\_↔NUM and HYBRID that are used to set the maximum number of [Node](#) in the [Graph](#) and to choose the algorithm to use are now in the CMakeLists.txt file, so that they can be changed from the command line.

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [problem\\_settings.h](#).

### 9.51.2 Macro Definition Documentation

#### 9.51.2.1 BETTER\_PROB

```
#define BETTER_PROB 0.05f
```

The third constant used to compare two [SubProblem](#) in the branch and bound algorithm.

If two [SubProblem](#) are within EPSILON2 (and therefore equal), the one that has a greater probability than the other of at least BETTER\_PROB is considered better.

#### See also

`branch_and_bound.c::compare_subproblems()`

Definition at line 82 of file [problem\\_settings.h](#).

### 9.51.2.2 EPSILON

```
#define EPSILON (GHOSH_UB / 1000)
```

The first constant used to compare two [SubProblem](#) in the branch and bound algorithm.

Two [SubProblem](#) are considered equal if their lower bound is within EPSILON of each other.

See also

`branch_and_bound.c::compare_subproblems()`

Definition at line 65 of file [problem\\_settings.h](#).

### 9.51.2.3 EPSILON2

```
#define EPSILON2 (0.1f * EPSILON)
```

The second constant used to compare two [SubProblem](#) in the branch and bound algorithm.

If two [SubProblem](#) are equal and their lower bound is within EPSILON2 of each other, their probability is compared.

See also

`branch_and_bound.c::compare_subproblems()`

Definition at line 73 of file [problem\\_settings.h](#).

### 9.51.2.4 GHOSH\_UB

```
#define GHOSH_UB (sqrt(MAX_VERTEX_NUM) * 1.27f)
```

The first upper bound for the problem, see: <https://www.semanticscholar.org/paper/Expected-Travel-Among-Random-Points-in-a-Region-Ghosh/4c395ab42054f4312ad24cb500fb8ca6f7>

Definition at line 53 of file [problem\\_settings.h](#).

### 9.51.2.5 INFINITE

```
#define INFINITE DBL_MAX
```

The maximum number to set the initial value of [Problem](#) and [SubProblem](#).

Definition at line 33 of file [problem\\_settings.h](#).

### 9.51.2.6 MAX\_EDGES\_NUM

```
#define MAX_EDGES_NUM (MAX_VERTEX_NUM * (MAX_VERTEX_NUM - 1) / 2)
```

The maximum number of edges in the [Graph](#).

Definition at line 49 of file [problem\\_settings.h](#).

### 9.51.2.7 NUM\_HK\_INITIAL\_ITERATIONS

```
#define NUM_HK_INITIAL_ITERATIONS (((float) MAX_VERTEX_NUM * MAX_VERTEX_NUM) / 50) + 0.5f) +  
MAX_VERTEX_NUM + 15)
```

The maximum number of dual iterations for the root of the branch and bound tree.

Definition at line 100 of file [problem\\_settings.h](#).

### 9.51.2.8 NUM\_HK\_ITERATIONS

```
#define NUM_HK_ITERATIONS ((float) MAX_VERTEX_NUM / 4) + 5)
```

The maximum number of dual iterations for nodes of the branch and bound tree that are not the root.

Definition at line 104 of file [problem\\_settings.h](#).

### 9.51.2.9 PRIM

```
#define PRIM 1
```

The maximum number of [Node](#) in the [Graph](#).

The parameter to choose the algorithm to use, 0 for Classic B&B, 1 for Hybrid B&B. 1 if the [MST](#) is solved with the Prim algorithm using Fibonacci Heap, 0 if it is solved with the Kruskal algorithm using MFSets.

Definition at line 45 of file [problem\\_settings.h](#).

### 9.51.2.10 PROB\_BRANCH

```
#define PROB_BRANCH 0.5f
```

The way with generate the children of a [SubProblem](#).

If the probability of a [SubProblem](#) is greater than PROB\_BRANCH, the children are generated by first imposing the constraint that the edge (i, j) is in the solution and then the constraint that it is not. Otherwise, the children are generated by imposing the constraint that the edge (i, j) is not in the solution and then the constraint that it is.

See also

`branch_and_bound.c::branch()`

Definition at line 92 of file [problem\\_settings.h](#).

### 9.51.2.11 TIME\_LIMIT\_SECONDS

```
#define TIME_LIMIT_SECONDS 600
```

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

Definition at line 96 of file [problem\\_settings.h](#).

### 9.51.2.12 TRACE

```
#define TRACE( ) fprintf(stderr, "%s (%d): %s\n", __FILE__, __LINE__, __func__)
```

Used to debug the code, to check if the execution reaches a certain point.

Definition at line 57 of file [problem\\_settings.h](#).

## 9.52 problem\_settings.h

[Go to the documentation of this file.](#)

```

00001
00017 #ifndef BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
00018 #define BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H
00019 #include <stdio.h>
00020 #include <limits.h>
00021 #include <float.h>
00022 #include <string.h>
00023 #include <stdarg.h>
00024 #include <stdbool.h>
00025 #include <stdlib.h>
00026 #include <math.h>
00027 #include <time.h>
00028 #include <errno.h>
00029 #include <pthread.h>
00030
00031
00033 #define INFINITE DBL_MAX
00034
00035
00037 //#define MAX_VERTEX_NUM 20/-- no longer in this file, but in the CMakeLists.txt to be able to change
it from the command line.
00038
00039
00041 //#define HYBRID 0/-- no longer in this file, but in the CMakeLists.txt to be able to change it from
the command line.
00042
00043
00045 #define PRIM 1
00046
00047
00049 #define MAX_EDGES_NUM (MAX_VERTEX_NUM * (MAX_VERTEX_NUM - 1) / 2)
00050
00051
00053 #define GHOSH_UB (sqrt(MAX_VERTEX_NUM) * 1.27f)
00054
00055
00057 #define TRACE() fprintf(stderr, "%s (%d): %s\n", __FILE__, __LINE__, __func__)
00058
00059
00061
00065 #define EPSILON (GHOSH_UB / 1000)
00066
00067
00069
00073 #define EPSILON2 (0.1f * EPSILON)
00074
00075
00077
00082 #define BETTER_PROB 0.05f
00083
00084
00086
00092 #define PROB_BRANCH 0.5f
00093
00094
00096 #define TIME_LIMIT_SECONDS 600
00097
00098
00100 #define NUM_HK_INITIAL_ITERATIONS (((float) MAX_VERTEX_NUM * MAX_VERTEX_NUM) / 50) + 0.5f) +
MAX_VERTEX_NUM + 15)
00101
00102
00104 #define NUM_HK_ITERATIONS ((float) MAX_VERTEX_NUM / 4) + 5)
00105
00106
00107 #endif //BRANCHANDBOUND1TREE_PROBLEM_SETTINGS_H

```

## 9.53 GraphConvolutionalBranchandBound/src/HybridSolver/main/↵ ReadME.md File Reference

## 9.54 GraphConvolutionalBranchandBound/src/HybridSolver/main/tsp\_↵ instance\_reader.c File Reference

The definition of the function to read input files.



```
#include "tsp_instance_reader.h"
```

## Functions

- void [read\\_tsp\\_lib\\_file](#) ([Graph](#) \*graph, char \*filename)  
*Reads a .tsp file and stores the data in the [Graph](#).*
- void [read\\_tsp\\_csv\\_file](#) ([Graph](#) \*graph, char \*filename)  
*Reads a .csv file and stores the data in the [Graph](#).*

### 9.54.1 Detailed Description

The definition of the function to read input files.

#### Author

Lorenzo Sciandra

There are two functions to read the input files, one for the .tsp format and one for the .csv format.

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [tsp\\_instance\\_reader.c](#).

### 9.54.2 Function Documentation

#### 9.54.2.1 read\_tsp\_csv\_file()

```
void read_tsp_csv_file (  
    Graph * graph,  
    char * filename )
```

Reads a .csv file and stores the data in the [Graph](#).

#### Parameters

<i>graph</i>	The <a href="#">Graph</a> where the data will be stored.
<i>filename</i>	The name of the file to read.

Definition at line 79 of file [tsp\\_instance\\_reader.c](#).

### 9.54.2.2 read\_tsp\_lib\_file()

```
void read_tsp_lib_file (
    Graph * graph,
    char * filename )
```

Reads a .tsp file and stores the data in the [Graph](#).

#### Parameters

<i>graph</i>	The <a href="#">Graph</a> where the data will be stored.
<i>filename</i>	The name of the file to read.

Definition at line 18 of file [tsp\\_instance\\_reader.c](#).

## 9.55 tsp\_instance\_reader.c

[Go to the documentation of this file.](#)

```
00001
00015 #include "tsp_instance_reader.h"
00016
00017
00018 void read_tsp_lib_file(Graph *graph, char *filename) {
00019     FILE *fp = fopen(filename, "r");
00020     if (fp == NULL) {
00021         perror("Error while opening the file.\n");
00022         printf("\nFile: %s\n", filename);
00023         exit(1);
00024     }
00025
00026     char *line = NULL;
00027     size_t len = 0;
00028     bool check_euc_2d = false;
00029     while (getline(&line, &len, fp) != -1 &&
00030            strstr(line, "NODE_COORD_SECTION") == NULL) {
00031         if (strstr(line, "EDGE_WEIGHT_TYPE : EUC_2D") == NULL) {
00032             check_euc_2d = true;
00033         }
00034     }
00035
00036     if (!check_euc_2d) {
00037         perror("The current TSP file is not an euclidean one.\n");
00038         printf("\nFile: %s\n", filename);
00039         exit(1);
00040     }
00041
00042     unsigned short i = 0;
00043     Node nodes[MAX_VERTEX_NUM];
00044     graph->kind = WEIGHTED_GRAPH;
00045     List *nodes_list = new_list();
00046     bool end_of_file = false;
00047     while (getline(&line, &len, fp) != -1 && !end_of_file) {
00048         if (strstr(line, "EOF") == NULL) {
00049             unsigned short id;
00050             float x;
00051             float y;
00052
00053             int result = sscanf(line, "%hu %f %f", &id, &x, &y);
00054             if (result != 3) {
00055                 perror("Error while reading the file.\n");
00056                 printf("\nFile: %s\n", filename);
00057                 exit(1);
00058             }
00059         }
00060     }
```

```

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

```

```

00146             graph->edges_matrix[i][j] = graph->edges[z];
00147             graph->edges_matrix[j][i] = graph->edges[z];
00148             z++;
00149         } else {
00150             graph->nodes[i].neighbours[graph->nodes[i].num_neighbours] = j;
00151             graph->nodes[i].num_neighbours++;
00152         }
00153     }
00154 }
00155     token = strtok(NULL, ";");
00156     j++;
00157 }
00158     graph->num_nodes++;
00159     i++;
00160 }
00161     free(line);
00162     if (fclose(fp) == EOF) {
00163         perror("Error while closing the file.\n");
00164         printf("\nFile: %s\n", filename);
00165         exit(1);
00166     }
00167 }

```

## 9.56 GraphConvolutionalBranchandBound/src/HybridSolver/main/tsp\_↵ instance\_reader.h File Reference

The declaration of the function to read input files.

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

### Functions

- void [read\\_tsp\\_lib\\_file](#) ([Graph](#) \*graph, char \*filename)  
*Reads a .tsp file and stores the data in the [Graph](#).*
- void [read\\_tsp\\_csv\\_file](#) ([Graph](#) \*graph, char \*filename)  
*Reads a .csv file and stores the data in the [Graph](#).*

### 9.56.1 Detailed Description

The declaration of the function to read input files.

#### Author

Lorenzo Sciandra

There are two functions to read the input files, one for the .tsp format and one for the .csv format.

#### Version

1.0.0 @data 2024-05-1

#### Copyright

Copyright (c) 2024, license MIT

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

Definition in file [tsp\\_instance\\_reader.h](#).

## 9.56.2 Function Documentation

### 9.56.2.1 read\_tsp\_csv\_file()

```
void read_tsp_csv_file (
    Graph * graph,
    char * filename )
```

Reads a .csv file and stores the data in the [Graph](#).

#### Parameters

<i>graph</i>	The <a href="#">Graph</a> where the data will be stored.
<i>filename</i>	The name of the file to read.

Definition at line 79 of file [tsp\\_instance\\_reader.c](#).

### 9.56.2.2 read\_tsp\_lib\_file()

```
void read_tsp_lib_file (
    Graph * graph,
    char * filename )
```

Reads a .tsp file and stores the data in the [Graph](#).

#### Parameters

<i>graph</i>	The <a href="#">Graph</a> where the data will be stored.
<i>filename</i>	The name of the file to read.

Definition at line 18 of file [tsp\\_instance\\_reader.c](#).

## 9.57 tsp\_instance\_reader.h

[Go to the documentation of this file.](#)

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



# Index

- action
  - HybridSolver, 18
- add\_dummy\_cities
  - main, 20
- add\_edge
  - mst.c, 168
  - mst.h, 173
- add\_elem\_list\_bottom
  - list\_functions.c, 120
  - list\_functions.h, 126
- add\_elem\_list\_index
  - list\_functions.c, 120
  - list\_functions.h, 127
- add\_elem\_SubProblemList\_bottom
  - b\_and\_b\_data.c, 100
  - b\_and\_b\_data.h, 111
- add\_elem\_SubProblemList\_index
  - b\_and\_b\_data.c, 101
  - b\_and\_b\_data.h, 112
- adjacency\_matrix
  - HybridSolver, 16
- b\_and\_b\_data.c
  - add\_elem\_SubProblemList\_bottom, 100
  - add\_elem\_SubProblemList\_index, 101
  - build\_list\_elem, 101
  - create\_SubProblemList\_iterator, 101
  - delete\_SubProblemList, 102
  - delete\_SubProblemList\_elem\_index, 102
  - delete\_SubProblemList\_iterator, 102
  - get\_current\_openSubProblemList\_iterator\_element, 103
  - get\_SubProblemList\_elem\_index, 103
  - get\_SubProblemList\_size, 103
  - is\_SubProblemList\_empty, 104
  - is\_SubProblemList\_iterator\_valid, 104
  - list\_openSubProblemList\_next, 104
  - new\_SubProblemList, 105
  - SubProblemList\_iterator\_get\_next, 105
- b\_and\_b\_data.h
  - add\_elem\_SubProblemList\_bottom, 111
  - add\_elem\_SubProblemList\_index, 112
  - BBNodeType, 110
  - CLOSED\_1TREE, 111
  - CLOSED\_BOUND, 111
  - CLOSED\_NN, 111
  - CLOSED\_NN\_HYBRID, 111
  - CLOSED\_SUBGRADIENT, 111
  - CLOSED\_UNFEASIBLE, 111
  - ConstraintType, 110, 111
  - create\_SubProblemList\_iterator, 112
  - delete\_SubProblemList, 112
  - delete\_SubProblemList\_elem\_index, 113
  - delete\_SubProblemList\_iterator, 113
  - FORBIDDEN, 111
  - get\_SubProblemList\_elem\_index, 113
  - get\_SubProblemList\_size, 114
  - is\_SubProblemList\_empty, 114
  - is\_SubProblemList\_iterator\_valid, 114
  - MANDATORY, 111
  - new\_SubProblemList, 115
  - NOTHING, 111
  - OPEN, 111
  - Problem, 110
  - SubProblem, 110
  - SubProblemElem, 110
  - SubProblemList\_iterator\_get\_next, 115
  - SubProblemsList, 110
- BBNodeType
  - b\_and\_b\_data.h, 110
- bestSolution
  - Problem, 43
- bestValue
  - Problem, 44
- BETTER\_PROB
  - problem\_settings.h, 188
- bound
  - branch\_and\_bound.c, 56
  - branch\_and\_bound.h, 80
- branch
  - branch\_and\_bound.c, 57
  - branch\_and\_bound.h, 81
- branch\_and\_bound
  - branch\_and\_bound.c, 57
  - branch\_and\_bound.h, 81
- branch\_and\_bound.c
  - bound, 56
  - branch, 57
  - branch\_and\_bound, 57
  - check\_feasibility, 57
  - check\_hamiltonian, 58
  - compare\_candidate\_node, 58
  - compare\_subproblems, 58
  - constrained\_kruskal, 60
  - constrained\_prim, 60
  - copy\_constraints, 60
  - dfs, 61
  - find\_candidate\_node, 61
  - infer\_constraints, 61

- initialize\_matrix, 62
- max\_edge\_path\_1Tree, 62
- mst\_to\_one\_tree, 62
- nearest\_prob\_neighbour, 63
- print\_problem, 63
- print\_subProblem, 63
- set\_problem, 64
- time\_limit\_reached, 64
- variable\_fixing, 64
- branch\_and\_bound.h
  - bound, 80
  - branch, 81
  - branch\_and\_bound, 81
  - check\_feasibility, 81
  - check\_hamiltonian, 82
  - clean\_matrix, 82
  - compare\_subproblems, 82
  - copy\_constraints, 83
  - dfs, 83
  - find\_candidate\_node, 84
  - infer\_constraints, 84
  - mst\_to\_one\_tree, 84
  - nearest\_prob\_neighbour, 85
  - print\_problem, 85
  - print\_subProblem, 86
  - problem, 87
  - set\_problem, 86
  - time\_limit\_reached, 86
  - variable\_fixing, 87
- BRANCHANDBOUND1TREE\_FIBONACCI\_HEAP\_H
  - fibonacci\_heap.h, 147
- BRANCHANDBOUND1TREE\_LINKED\_LIST\_H
  - linked\_list.h, 118
- BRANCHANDBOUND1TREE\_MST\_H
  - mst.h, 172
- build\_c\_program
  - HybridSolver, 16
- build\_dll\_elem
  - list\_functions.c, 120
- build\_list\_elem
  - b\_and\_b\_data.c, 101
- candidateNodeId
  - Problem, 44
- cascading\_cut
  - fibonacci\_heap.c, 139
- category
  - main, 22
- check\_feasibility
  - branch\_and\_bound.c, 57
  - branch\_and\_bound.h, 81
- check\_hamiltonian
  - branch\_and\_bound.c, 58
  - branch\_and\_bound.h, 82
- clean\_matrix
  - branch\_and\_bound.h, 82
- CLOSED\_1TREE
  - b\_and\_b\_data.h, 111
- CLOSED\_BOUND
  - b\_and\_b\_data.h, 111
- CLOSED\_NN
  - b\_and\_b\_data.h, 111
- CLOSED\_NN\_HYBRID
  - b\_and\_b\_data.h, 111
- CLOSED\_SUBGRADIENT
  - b\_and\_b\_data.h, 111
- CLOSED\_UNFEASIBLE
  - b\_and\_b\_data.h, 111
- cluster\_nodes
  - main, 20
- compare\_candidate\_node
  - branch\_and\_bound.c, 58
- compare\_subproblems
  - branch\_and\_bound.c, 58
  - branch\_and\_bound.h, 82
- compute\_prob
  - main, 20
- consolidate
  - fibonacci\_heap.c, 139
- constrained\_kruskal
  - branch\_and\_bound.c, 60
- constrained\_prim
  - branch\_and\_bound.c, 60
- ConstrainedEdge, 25
  - dest, 25
  - mst.h, 173
  - src, 25
- constraints
  - SubProblem, 48
- ConstraintType
  - b\_and\_b\_data.h, 110, 111
- copy\_constraints
  - branch\_and\_bound.c, 60
  - branch\_and\_bound.h, 83
- cost
  - Graph, 32
  - MST, 36
- create\_euclidean\_graph
  - graph.c, 152
  - graph.h, 158
- create\_fibonacci\_heap
  - fibonacci\_heap.c, 139
  - fibonacci\_heap.h, 148
- create\_forest
  - mfset.c, 160
  - mfset.h, 165
- create\_forest\_constrained
  - mfset.c, 161
  - mfset.h, 165
- create\_graph
  - graph.c, 152
  - graph.h, 158
- create\_insert\_node
  - fibonacci\_heap.c, 139
  - fibonacci\_heap.h, 148
- create\_list\_iterator
  - list\_iterator.c, 131



- list\_iterator.h, 135
- create\_mst
  - mst.c, 169
  - mst.h, 173
- create\_node
  - fibonacci\_heap.c, 140
  - fibonacci\_heap.h, 149
- create\_SubProblemList\_iterator
  - b\_and\_b\_data.c, 101
  - b\_and\_b\_data.h, 112
- create\_temp\_file
  - HybridSolver, 16
  - main, 20
- curr
  - ListIterator, 35
  - Set, 46
  - SubProblemsListIterator, 54
- cut
  - fibonacci\_heap.c, 140
- cycleEdges
  - SubProblem, 48
- decrease\_value
  - fibonacci\_heap.c, 140
  - fibonacci\_heap.h, 149
- default
  - HybridSolver, 18
- del\_list
  - list\_functions.c, 120
  - list\_functions.h, 127
- delete\_list\_elem\_bottom
  - list\_functions.c, 121
  - list\_functions.h, 127
- delete\_list\_elem\_index
  - list\_functions.c, 121
  - list\_functions.h, 128
- delete\_list\_iterator
  - list\_iterator.c, 131
  - list\_iterator.h, 135
- delete\_node
  - fibonacci\_heap.c, 141
- delete\_SubProblemList
  - b\_and\_b\_data.c, 102
  - b\_and\_b\_data.h, 112
- delete\_SubProblemList\_elem\_index
  - b\_and\_b\_data.c, 102
  - b\_and\_b\_data.h, 113
- delete\_SubProblemList\_iterator
  - b\_and\_b\_data.c, 102
  - b\_and\_b\_data.h, 113
- dest
  - ConstrainedEdge, 25
  - Edge, 27
- dfs
  - branch\_and\_bound.c, 61
  - branch\_and\_bound.h, 83
- DllElem, 26
  - linked\_list.h, 118
  - next, 26
  - prev, 26
  - value, 27
- Edge, 27
  - dest, 27
  - graph.h, 156
  - positionInGraph, 28
  - prob, 28
  - src, 28
  - symbol, 28
  - weight, 28
- edge\_to\_branch
  - SubProblem, 49
- edges
  - Graph, 32
  - MST, 37
- edges\_matrix
  - Graph, 32
  - MST, 37
- end
  - Problem, 44
- EPSILON
  - problem\_settings.h, 188
- EPSILON2
  - problem\_settings.h, 189
- exploredBBNodes
  - Problem, 44
- extract\_min
  - fibonacci\_heap.c, 141
  - fibonacci\_heap.h, 149
- fatherId
  - SubProblem, 49
- fibonacci\_heap.c
  - cascading\_cut, 139
  - consolidate, 139
  - create\_fibonacci\_heap, 139
  - create\_insert\_node, 139
  - create\_node, 140
  - cut, 140
  - decrease\_value, 140
  - delete\_node, 141
  - extract\_min, 141
  - insert\_node, 141
  - link\_trees, 142
  - swap\_roots, 142
- fibonacci\_heap.h
  - BRANCHANDBOUND1TREE\_FIBONACCI\_HEAP\_H, 147
  - create\_fibonacci\_heap, 148
  - create\_insert\_node, 148
  - create\_node, 149
  - decrease\_value, 149
  - extract\_min, 149
  - FibonacciHeap, 147
  - insert\_node, 150
  - OrdTreeNode, 148
- FibonacciHeap, 29
  - fibonacci\_heap.h, 147

- head\_tree\_list, 29
- min\_root, 29
- num\_nodes, 30
- num\_trees, 30
- tail\_tree\_list, 30
- filepath
  - main, 23
- find
  - mfset.c, 161
  - mfset.h, 166
- find\_candidate\_node
  - branch\_and\_bound.c, 61
  - branch\_and\_bound.h, 84
- fix\_instance\_size
  - main, 21
- FORBIDDEN
  - b\_and\_b\_data.h, 111
- Forest, 30
  - mfset.h, 165
  - num\_sets, 31
  - sets, 31
- gen\_matrix
  - HybridSolver, 18
- generatedBBNodes
  - Problem, 44
- get\_current\_list\_iterator\_element
  - list\_iterator.c, 132
  - list\_iterator.h, 136
- get\_current\_openSubProblemList\_iterator\_element
  - b\_and\_b\_data.c, 103
- get\_instance
  - HybridSolver, 16
  - main, 21
- get\_list\_elem\_index
  - list\_functions.c, 121
  - list\_functions.h, 128
- get\_list\_size
  - list\_functions.c, 122
  - list\_functions.h, 128
- get\_nodes
  - HybridSolver, 17
- get\_SubProblemList\_elem\_index
  - b\_and\_b\_data.c, 103
  - b\_and\_b\_data.h, 113
- get\_SubProblemList\_size
  - b\_and\_b\_data.c, 103
  - b\_and\_b\_data.h, 114
- GHOSH\_UB
  - problem\_settings.h, 189
- Graph, 31
  - cost, 32
  - edges, 32
  - edges\_matrix, 32
  - graph.h, 156
  - kind, 32
  - nodes, 32
  - num\_edges, 33
  - num\_nodes, 33

- orderedEdges, 33
- graph
  - Problem, 45
- graph.c
  - create\_euclidean\_graph, 152
  - create\_graph, 152
  - print\_graph, 153
- graph.h
  - create\_euclidean\_graph, 158
  - create\_graph, 158
  - Edge, 156
  - Graph, 156
  - GraphKind, 156, 157
  - Node, 156
  - print\_graph, 158
  - UNWEIGHTED\_GRAPH, 158
  - WEIGHTED\_GRAPH, 158
- GraphConvolutionalBranchandBound/README.md, 181
- GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/br 55, 65
- GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/br 79, 88
- GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/kr 89, 91
- GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/kr 93, 95
- GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/pr 96, 97
- GraphConvolutionalBranchandBound/src/HybridSolver/main/algorithms/pr 98, 99
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 99, 105
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 108, 116
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 117, 118
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 119, 123
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 125, 130
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 130, 133
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 134, 137
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 138, 142
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 146, 150
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 151, 153
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 155, 159
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 160, 162
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur 163, 167
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data\_structur

- 168, 170
- GraphConvolutionalBranchandBound/src/HybridSolver/main/data/branchandbound.h, 141
- 171, 174
- GraphConvolutionalBranchandBound/src/HybridSolver/main/convnet-tsp/main.py, 175, 176
- GraphConvolutionalBranchandBound/src/HybridSolver/main/convnet-tsp/README.md, 181
- GraphConvolutionalBranchandBound/src/HybridSolver/main/convnet-tsp/README.md, 181, 182
- GraphConvolutionalBranchandBound/src/HybridSolver/main/main.c, 185, 186
- GraphConvolutionalBranchandBound/src/HybridSolver/main/problem\_settings.h, 187, 192
- GraphConvolutionalBranchandBound/src/HybridSolver/main/README.md, 192
- GraphConvolutionalBranchandBound/src/HybridSolver/main/SubProblemList.c, 192, 194
- GraphConvolutionalBranchandBound/src/HybridSolver/main/tsp/branchandbound.h, 196, 197
- GraphKind
  - graph.h, 156, 157
- head
  - List, 34
  - SubProblemsList, 53
- head\_child\_list
  - OrdTreeNode, 40
- head\_tree\_list
  - FibonacciHeap, 29
- hybrid\_solver
  - HybridSolver, 17
- HybridSolver, 15
  - action, 18
  - adjacency\_matrix, 16
  - build\_c\_program, 16
  - create\_temp\_file, 16
  - default, 18
  - gen\_matrix, 18
  - get\_instance, 16
  - get\_nodes, 17
  - hybrid\_solver, 17
  - int, 18
  - opts, 18
  - parser, 18
  - str, 19
  - type, 19
- id
  - SubProblem, 49
- index
  - ListIterator, 35
  - SubProblemsListIterator, 54
- infer\_constraints
  - branch\_and\_bound.c, 61
  - branch\_and\_bound.h, 84
- INFINITE
  - problem\_settings.h, 189
- initialize\_matrix
  - branch\_and\_bound.c, 62
- insert\_node
  - fibonacci\_heap.c, 141
  - fibonacci\_heap.h, 150
- intgraph-
  - HybridSolver, 18
- interrupted
  - Problem, 45
- is\_HybridSolver.py,
  - list\_functions.c, 122
  - list\_functions.h, 129
  - is\_list\_iterator\_valid
  - list\_iterator.h, 136
- is\_ReadME.md,
  - OrdTreeNode, 41
- is\_SubProblemList\_empty
  - b\_and\_b\_data.c, 104
  - is\_SubProblemList\_iterator\_valid
  - b\_and\_b\_data.c, 104
  - b\_and\_b\_data.h, 114
- isValid
  - MST, 37
- key
  - OrdTreeNode, 41
- kind
  - Graph, 32
- kruskal
  - kruskal.c, 89
  - kruskal.h, 93
- kruskal.c
  - kruskal, 89
  - pivot\_quicksort, 90
  - quick\_sort, 90
  - swap, 90
  - wrap\_quick\_sort, 90
- kruskal.h
  - kruskal, 93
  - pivot\_quicksort, 94
  - quick\_sort, 94
  - swap, 95
  - wrap\_quick\_sort, 95
- left\_sibling
  - OrdTreeNode, 41
- link\_trees
  - fibonacci\_heap.c, 142
- linked\_list.h
  - BRANCHANDBOUND1TREE\_LINKED\_LIST\_H, 118
  - DllElem, 118
- List, 33
  - head, 34
  - size, 34
  - tail, 34
- list
  - ListIterator, 35
  - SubProblemsListIterator, 54

- list\_functions.c
  - add\_elem\_list\_bottom, [120](#)
  - add\_elem\_list\_index, [120](#)
  - build\_dll\_elem, [120](#)
  - del\_list, [120](#)
  - delete\_list\_elem\_bottom, [121](#)
  - delete\_list\_elem\_index, [121](#)
  - get\_list\_elem\_index, [121](#)
  - get\_list\_size, [122](#)
  - is\_list\_empty, [122](#)
  - new\_list, [123](#)
- list\_functions.h
  - add\_elem\_list\_bottom, [126](#)
  - add\_elem\_list\_index, [127](#)
  - del\_list, [127](#)
  - delete\_list\_elem\_bottom, [127](#)
  - delete\_list\_elem\_index, [128](#)
  - get\_list\_elem\_index, [128](#)
  - get\_list\_size, [128](#)
  - is\_list\_empty, [129](#)
  - new\_list, [129](#)
- list\_iterator.c
  - create\_list\_iterator, [131](#)
  - delete\_list\_iterator, [131](#)
  - get\_current\_list\_iterator\_element, [132](#)
  - is\_list\_iterator\_valid, [132](#)
  - list\_iterator\_get\_next, [133](#)
  - list\_iterator\_next, [133](#)
- list\_iterator.h
  - create\_list\_iterator, [135](#)
  - delete\_list\_iterator, [135](#)
  - get\_current\_list\_iterator\_element, [136](#)
  - is\_list\_iterator\_valid, [136](#)
  - list\_iterator\_get\_next, [137](#)
  - list\_iterator\_next, [137](#)
- list\_iterator\_get\_next
  - list\_iterator.c, [133](#)
  - list\_iterator.h, [137](#)
- list\_iterator\_next
  - list\_iterator.c, [133](#)
  - list\_iterator.h, [137](#)
- list\_openSubProblemList\_next
  - b\_and\_b\_data.c, [104](#)
- ListIterator, [35](#)
  - curr, [35](#)
  - index, [35](#)
  - list, [35](#)
- main, [19](#)
  - add\_dummy\_cities, [20](#)
  - category, [22](#)
  - cluster\_nodes, [20](#)
  - compute\_prob, [20](#)
  - create\_temp\_file, [20](#)
  - filepath, [23](#)
  - fix\_instance\_size, [21](#)
  - get\_instance, [21](#)
  - main, [21](#)
  - main.c, [186](#)
  - model\_size, [23](#)
  - num\_nodes, [23](#)
  - write\_adjacency\_matrix, [22](#)
- main.c
  - main, [186](#)
- MANDATORY
  - b\_and\_b\_data.h, [111](#)
- mandatoryEdges
  - SubProblem, [49](#)
- marked
  - OrdTreeNode, [41](#)
- max\_edge\_path\_1Tree
  - branch\_and\_bound.c, [62](#)
- MAX\_EDGES\_NUM
  - problem\_settings.h, [189](#)
- merge
  - mfset.c, [162](#)
  - mfset.h, [166](#)
- mfset.c
  - create\_forest, [160](#)
  - create\_forest\_constrained, [161](#)
  - find, [161](#)
  - merge, [162](#)
  - print\_forest, [162](#)
- mfset.h
  - create\_forest, [165](#)
  - create\_forest\_constrained, [165](#)
  - find, [166](#)
  - Forest, [165](#)
  - merge, [166](#)
  - print\_forest, [167](#)
  - Set, [165](#)
- min\_root
  - FibonacciHeap, [29](#)
- model\_size
  - main, [23](#)
- MST, [36](#)
  - cost, [36](#)
  - edges, [37](#)
  - edges\_matrix, [37](#)
  - isValid, [37](#)
  - mst.h, [173](#)
  - nodes, [37](#)
  - num\_edges, [37](#)
  - num\_nodes, [38](#)
  - prob, [38](#)
- mst.c
  - add\_edge, [168](#)
  - create\_mst, [169](#)
  - print\_mst, [169](#)
  - print\_mst\_original\_weight, [169](#)
- mst.h
  - add\_edge, [173](#)
  - BRANCHANDBOUND1TREE\_MST\_H, [172](#)
  - ConstrainedEdge, [173](#)
  - create\_mst, [173](#)
  - MST, [173](#)
  - print\_mst, [174](#)

- print\_mst\_original\_weight, 174
- mst\_to\_one\_tree
  - branch\_and\_bound.c, 62
  - branch\_and\_bound.h, 84
- nearest\_prob\_neighbour
  - branch\_and\_bound.c, 63
  - branch\_and\_bound.h, 85
- neighbours
  - Node, 39
- new\_list
  - list\_functions.c, 123
  - list\_functions.h, 129
- new\_SubProblemList
  - b\_and\_b\_data.c, 105
  - b\_and\_b\_data.h, 115
- next
  - DllElem, 26
  - SubProblemElem, 52
- Node, 38
  - graph.h, 156
  - neighbours, 39
  - num\_neighbours, 39
  - positionInGraph, 39
  - x, 39
  - y, 39
- nodes
  - Graph, 32
  - MST, 37
- NOTHING
  - b\_and\_b\_data.h, 111
- num\_children
  - OrdTreeNode, 41
- num\_edges
  - Graph, 33
  - MST, 37
- num\_edges\_in\_cycle
  - SubProblem, 49
- num\_fixed\_edges
  - Problem, 45
- num\_forbidden\_edges
  - SubProblem, 50
- NUM\_HK\_INITIAL\_ITERATIONS
  - problem\_settings.h, 190
- NUM\_HK\_ITERATIONS
  - problem\_settings.h, 190
- num\_in\_forest
  - Set, 47
- num\_mandatory\_edges
  - SubProblem, 50
- num\_neighbours
  - Node, 39
- num\_nodes
  - FibonacciHeap, 30
  - Graph, 33
  - main, 23
  - MST, 38
- num\_sets
  - Forest, 31
- num\_trees
  - FibonacciHeap, 30
- oneTree
  - SubProblem, 50
- OPEN
  - b\_and\_b\_data.h, 111
- opts
  - HybridSolver, 18
- orderedEdges
  - Graph, 33
- OrdTreeNode, 40
  - fibonacci\_heap.h, 148
  - head\_child\_list, 40
  - is\_root, 41
  - key, 41
  - left\_sibling, 41
  - marked, 41
  - num\_children, 41
  - parent, 42
  - right\_sibling, 42
  - tail\_child\_list, 42
  - value, 42
- parent
  - OrdTreeNode, 42
- parentSet
  - Set, 47
- parser
  - HybridSolver, 18
- pivot\_quicksort
  - kruskal.c, 90
  - kruskal.h, 94
- positionInGraph
  - Edge, 28
  - Node, 39
- prev
  - DllElem, 26
  - SubProblemElem, 52
- PRIM
  - problem\_settings.h, 190
- prim
  - prim.c, 97
  - prim.h, 99
- prim.c
  - prim, 97
- prim.h
  - prim, 99
- print\_forest
  - mfset.c, 162
  - mfset.h, 167
- print\_graph
  - graph.c, 153
  - graph.h, 158
- print\_mst
  - mst.c, 169
  - mst.h, 174
- print\_mst\_original\_weight
  - mst.c, 169

- mst.h, 174
- print\_problem
  - branch\_and\_bound.c, 63
  - branch\_and\_bound.h, 85
- print\_subProblem
  - branch\_and\_bound.c, 63
  - branch\_and\_bound.h, 86
- prob
  - Edge, 28
  - MST, 38
  - SubProblem, 50
- PROB\_BRANCH
  - problem\_settings.h, 190
- Problem, 43
  - b\_and\_b\_data.h, 110
  - bestSolution, 43
  - bestValue, 44
  - candidateNodeId, 44
  - end, 44
  - exploredBBNodes, 44
  - generatedBBNodes, 44
  - graph, 45
  - interrupted, 45
  - num\_fixed\_edges, 45
  - reformulationGraph, 45
  - start, 45
  - totTreeLevels, 46
- problem
  - branch\_and\_bound.h, 87
- problem\_settings.h
  - BETTER\_PROB, 188
  - EPSILON, 188
  - EPSILON2, 189
  - GHOSH\_UB, 189
  - INFINITE, 189
  - MAX\_EDGES\_NUM, 189
  - NUM\_HK\_INITIAL\_ITERATIONS, 190
  - NUM\_HK\_ITERATIONS, 190
  - PRIM, 190
  - PROB\_BRANCH, 190
  - TIME\_LIMIT\_SECONDS, 191
  - TRACE, 191
- quick\_sort
  - kruskal.c, 90
  - kruskal.h, 94
- rango
  - Set, 47
- read\_tsp\_csv\_file
  - tsp\_instance\_reader.c, 193
  - tsp\_instance\_reader.h, 197
- read\_tsp\_lib\_file
  - tsp\_instance\_reader.c, 194
  - tsp\_instance\_reader.h, 197
- reformulationGraph
  - Problem, 45
- right\_sibling
  - OrdTreeNode, 42
- Set, 46
  - curr, 46
  - mfset.h, 165
  - num\_in\_forest, 47
  - parentSet, 47
  - rango, 47
- set\_problem
  - branch\_and\_bound.c, 64
  - branch\_and\_bound.h, 86
- sets
  - Forest, 31
- size
  - List, 34
  - SubProblemsList, 53
- src
  - ConstrainedEdge, 25
  - Edge, 28
- start
  - Problem, 45
- str
  - HybridSolver, 19
- SubProblem, 47
  - b\_and\_b\_data.h, 110
  - constraints, 48
  - cycleEdges, 48
  - edge\_to\_branch, 49
  - fatherId, 49
  - id, 49
  - mandatoryEdges, 49
  - num\_edges\_in\_cycle, 49
  - num\_forbidden\_edges, 50
  - num\_mandatory\_edges, 50
  - oneTree, 50
  - prob, 50
  - timeToReach, 50
  - treeLevel, 51
  - type, 51
  - value, 51
- subProblem
  - SubProblemElem, 52
- SubProblemElem, 51
  - b\_and\_b\_data.h, 110
  - next, 52
  - prev, 52
  - subProblem, 52
- SubProblemList\_iterator\_get\_next
  - b\_and\_b\_data.c, 105
  - b\_and\_b\_data.h, 115
- SubProblemsList, 53
  - b\_and\_b\_data.h, 110
  - head, 53
  - size, 53
  - tail, 53
- SubProblemsListIterator, 54
  - curr, 54
  - index, 54
  - list, 54
- swap

- kruskal.c, [90](#)
- kruskal.h, [95](#)
- swap\_roots
  - fibonacci\_heap.c, [142](#)
- symbol
  - Edge, [28](#)
- tail
  - List, [34](#)
  - SubProblemsList, [53](#)
- tail\_child\_list
  - OrdTreeNode, [42](#)
- tail\_tree\_list
  - FibonacciHeap, [30](#)
- time\_limit\_reached
  - branch\_and\_bound.c, [64](#)
  - branch\_and\_bound.h, [86](#)
- TIME\_LIMIT\_SECONDS
  - problem\_settings.h, [191](#)
- timeToReach
  - SubProblem, [50](#)
- totTreeLevels
  - Problem, [46](#)
- TRACE
  - problem\_settings.h, [191](#)
- treeLevel
  - SubProblem, [51](#)
- tsp\_instance\_reader.c
  - read\_tsp\_csv\_file, [193](#)
  - read\_tsp\_lib\_file, [194](#)
- tsp\_instance\_reader.h
  - read\_tsp\_csv\_file, [197](#)
  - read\_tsp\_lib\_file, [197](#)
- type
  - HybridSolver, [19](#)
  - SubProblem, [51](#)
- UNWEIGHTED\_GRAPH
  - graph.h, [158](#)
- value
  - DllElem, [27](#)
  - OrdTreeNode, [42](#)
  - SubProblem, [51](#)
- variable\_fixing
  - branch\_and\_bound.c, [64](#)
  - branch\_and\_bound.h, [87](#)
- weight
  - Edge, [28](#)
- WEIGHTED\_GRAPH
  - graph.h, [158](#)
- wrap\_quick\_sort
  - kruskal.c, [90](#)
  - kruskal.h, [95](#)
- write\_adjacency\_matrix
  - main, [22](#)

y

Node, [39](#)Node, [39](#)

x