TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

CSR-24-03

Comparison of maximum weight clique algorithms

Sebastian Pettke · Wolfram Hardt · Ariane Heller

August 2024

Chemnitzer Informatik-Berichte

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Comparison of maximum weight clique algorithms

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Sebastian Pettke
Student ID: 404145
Date: 11.03.2024

Supervising tutor: Prof. Dr. rer. nat. Dr. h. c. Wolfram Hardt
Internal supervisor: Dr.-Ing. Ariane Heller
External supervisor: Dr. rer. nat. Holger Langenau

Acknowledgments

I would like to profoundly thank my supervisors Prof. Dr. rer. nat. Dr. h. c. Wolfram Hardt, Dr.-Ing. Ariane Heller and Dr. rer. nat. Holger Langenau for all their valuable support and input that made this thesis possible.

My thanks also go to Paul Stöcker, who initially drew my attention to the topic of clique search.

Furthermore, I am grateful for all the scientists who worked on the maximum weight clique problem, as my thesis is built upon their work. I would especially like to thank Prof. Chu-Min Li, Prof. Hua Jiang, Prof. Yiyuan Wang and Prof. Dr. Christian Schulz for their help with obtaining the reference implementations of their algorithms.

I am thankful to Mario Haustein and Dr.-Ing. Sebastian Heil for their help with my LaTeX-related questions.

In addition I would like to thank everyone else who accompanied me during my time at the TU Chemnitz, be it fellow students, professors, tutors or cafeteria staff.

Last but not least, I owe a lot of gratitude to my family for their support all the way from elementary school to the end of my master studies.

Abstract

The maximum weight clique problem has a wide area of possible applications and various heuristic and exact algorithms for solving it have been proposed. Multiple algorithms, including a new heuristic approach called unexplored weight (UEW), are implemented and tested on systematically ordered graphs. An approach for optimizing consecutive calculations on similar graphs is also discussed. The focus of the experiments lies on applications that require fast computations, so comparatively small graphs are tested within a short time limit. The results are evaluated in detail and recommendations about which algorithm to chose for which graphs are provided. The experiments reveal that the new UEW algorithm runs significantly faster than its competitors on smaller graphs, while still providing an acceptable solution quality.

Keywords:
Maximum weight clique problem
Heuristic algorithm
Exact algorithm
Unexplored weight

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

BMS Best from Multiple Selection

BnB Branch-and-Bound

CC Configuration Checking

IS Independent Set

MC Maximum Clique

MCP Maximum Clique Problem

MWC Maximum Weight Clique

MWCP Maximum Weight Clique Problem

SCC Strong Configuration Checking

UEW Unexplored Weight

UEW-R Unexplored Weight Reduction

1 Introduction

1.1 Motivation

Vertex-weighted graphs are a versatile tool to model a multitude of problems. The vertex weights can be set to represent a certain benefit value, while the edges in between them represent some sort of pairwise compatibility. If the aim is now to maximize the benefit, it is of interest to find the largest-weight set of vertices that are compatible with each other. Such a set is known as the maximum weight clique of the graph. The potential applications for maximum weight clique problems to practical problems cover a wide range from computer vision [1] over social network analysis [2] and communication networks [3] to biology [4], [5]. Since the maximum weight clique problem is a generalized version of the maximum clique problem [6], algorithms for the former are also capable of finding maximum cliques, which broadens the range of applications even further.

Various state of the art algorithms to solve the maximum weight clique problem have been proposed, like for example FastWClq [7], SCCWalk [6] or TSM-MWC [8]. When it comes to comprehensive comparisons of the algorithms as a guide for which algorithm to apply for which types of graphs, there seems to be untapped potential in the literature. McCreesh et al. particularly also criticize the nature of the weights used in many experiments in [9]. Usually the papers presenting the algorithms include comparisons with few algorithms and put more emphasis on certain graph instances instead of a larger number of systematically ordered test graphs. Furthermore the literature usually focuses more on large graphs with many thousands to millions of vertices. The existing algorithms are designed and tested for processing different graph instances independently from each other. Even when operating consecutively on very similar graphs, previous computation results are not reused. Together, the mentioned factors create a gap that this thesis aims to fill.

1.2 Objectives

Deriving from that, the thesis will have two main objectives. The first main objective is to research and select state of the art maximum weight clique algorithms and to test them on many different graphs. The set of graphs used should be systematically defined the possible combinations of certain vertex counts, edge densities and vertex

weight distributions. The focus should be on rather small graphs with at most 4000 vertices, aiming for application domains where fast runtimes are crucial.

The second objective is about exploring new algorithmic approaches. Besides of developing an own simple heuristic approach, the optimization potential of the existing algorithms when being iteratively applied on similar graphs should be evaluated. Instead of trying to find a maximum weight clique from scratch in every iteration, adapted algorithms should try to reuse the information from the clique found in the previous iteration. It should be evaluated whether such approaches can bring benefits in regard to the needed computation time or the quality of the solutions.

1.3 Related work at the professorship Computer Engineering

As already hinted, communication networks are a domain where maximum weight clique algorithms can be applied, since graphs in general are an important instrument for modeling networks. At the professorship Computer Engineering, there has been various research on communication systems, in particular with a focus on optimizing energy efficiency [10]–[13]. Other research activities at the professorship concern hardware-/software partitioning [14], [15] and task assignment [16]. What all those topics have in common, is their relation to assignment problems. Those are naturally also a great fit for being modeled as a maximum weight clique problem. A vertex can be representative for a particular assignment between two entities, and the vertex weight is chosen to represent the benefit of that assignment. The edges would then be chosen so that there exists an edge between each pair of vertices that represent assignments that are not in conflict, e.g. by not assigning the same resource twice. Calculating the maximum weight clique on such a graph should then deliver an optimal solution of the assignment problem.

1.4 Structure of the thesis

At first, chapter 2 of the thesis will introduce the reader to some fundamentals needed to understand the maximum weight clique problem and the related algorithms. Building upon that chapter 3 presents several algorithms for solving it. After that, in chapter 4 new algorithmic approaches, especially for working iteratively on similar graphs are described. Together with that an experimental setup gets explained. It will be used to evaluate the algorithms in different situations. Finally chapter 5 will then contain the analysis of the results obtained with the experimental setup and based on that try to formulate recommendations regarding the suitability of the various algorithms for different graphs and problems.

2 Fundamentals

The following sections will briefly introduce basic terms, definitions and conventions that are used in this thesis.

2.1 Graphs

A **graph** $G = (V, E)$ is defined by a set of n vertices $V = \{v_1, v_2, ..., v_n\}$ (also called nodes) and a set of m edges E. The existence of the edge $(v_i, v_j) \in E$ denotes that the two vertices v_i and v_j are connected, i.e., adjacent. It is assumed that $i \neq j$, so a vertex cannot be connected to itself with an edge.

An **undirected graph** is a special case of a graph where the order of the vertices in an edge is irrelevant, meaning that there is no difference between $(v_i, v_j) \in E$ and $(v_j, v_i) \in E$. In this thesis, only undirected graphs are considered. When a "graph" is mentioned, it is implied that this is an undirected graph.

If two vertices are adjacent, they are called **neighbors**. When referring to the neighbors or neighborhood $N(v_i)$ of a vertex v_i, the set of all its adjacent vertices, not including the vertex itself, is meant. The neighborhood (also called exclusive neighborhood or open neighborhood) is defined as $N(v_i) = \{v_j \in V \mid (v_i, v_j) \in E\}$. In contrast, the **inclusive neighborhood** (or closed neighborhood) $N[v_i]$ of a vertex also includes the vertex itself $N[v_i] = N(v_i) \cup \{v_i\}$. The number of neighbors that a vertex has is the **degree** of the vertex $\deg(v_i) = |N(v_i)|$ [17].

To classify a graph with regard to the relative amount of edges, the measure of **density**

$$d = \frac{2m}{n(n-1)} = \frac{m}{\binom{n}{2}}$$

is used [18]. As it can be seen, it is the ratio between the number of edges the graph has and the number of edges a graph with n vertices and all possible edges between them has.

If weights are assigned to each vertex, the graph is a **vertex-weighted graph**. Such a weighted graph is then defined as $G = (V, E, w)$, where w is a function that assigns a weight to each vertex. The weight of a vertex v_i is thus referred to as $w(v_i)$. Furthermore the notation $v_i^{w_i}$ represents a vertex with the index i and a weight of $w_i = w(v_i)$. v_i and $v_i^{w_i}$ denominate the same vertex. Which of the two notations is

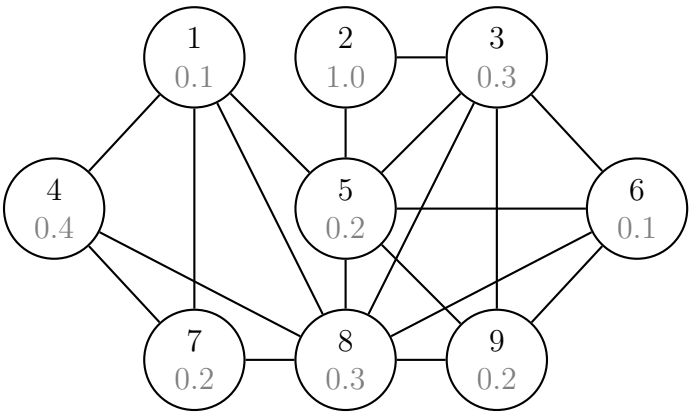

Figure 2.1: Example of a vertex-weighted graph

used depends on the relevance of the vertex weight in the respective context. It is assumed that $\forall v_i \in V : w(v_i) > 0$, weights should not be zero or negative. If in the practical use case an entity represented by a vertex provides no benefit and would therefore have a weight of exactly zero, it should not be included in the graph. Some algorithms furthermore require integer weights. The vertex weights of the graphs that the algorithms are tested on are not necessarily integers, but rather floating point values in a range of $0 < w(v) \leq 1$. In such cases, all vertex weights are premultiplied with a constant factor before being passed to those algorithms.

An example of how graphs are visualized within the thesis is shown in figure 2.1. A vertex is depicted by a circle. The black number at the top inside the circle is the index of the vertex, while the gray number at the bottom represents the vertex-weight. If unweighted graphs are visualized, the circles contain only one number, which is the index. If two vertices are connected by an edge, a line is drawn between the respective circles. In the given example graph, v_1 (the vertex with index 1) has a weight of 0.1 and is connected to v_4, v_5, v_7 and v_8. $v_2^{1.0}$ is connected to v_3 and v_5, etc. Sometimes only a part of a graph is relevant for an explanation and the rest of the graph is not visualized. Such left out parts of a graph are indicated by dotted lines that go downwards from a vertex into nowhere. This is often done in figures for graph reductions.

2.2 Cliques

A **clique** C is a subset of the vertices V that satisfies the condition

$$\forall v_i, v_j \in C, i \neq j : (v_i, v_j) \in E,$$

i.e., every vertex in a clique is connected to each other vertex in that clique. In the previously given example graph (figure 2.1), one clique is for instance formed by v_1,

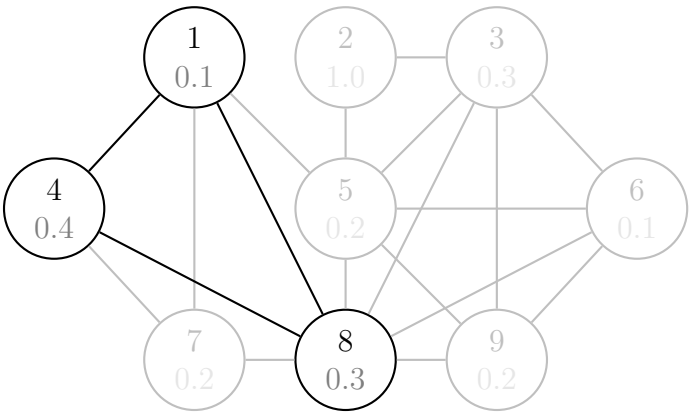

Figure 2.2: Example of a clique

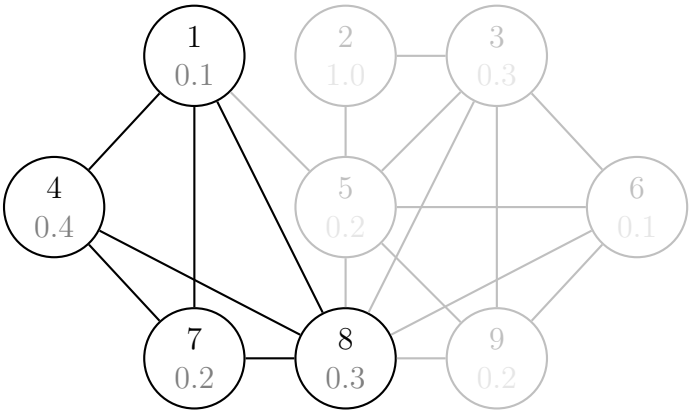

Figure 2.3: Example of a maximal clique

v_4 and v_8. This clique is highlighted in figure 2.2 and as it can easily be seen, all three vertices are connected to each other. The weight of a clique C is calculated by adding together the weights of all its vertices:

$$w(C) = \sum_{v \in C} w(v).$$

A **maximal clique** is a clique that can not be extended by any other vertex while still remaining a clique. The clique highlighted in figure 2.2 is not a maximal clique. v_7 is connected to v_1, v_4 and v_8, so it can be be used to extend the clique to the one highlighted in figure 2.3. Since there is no other vertex in the graph that is connected to all of v_1, v_4, v_7 and v_8, this is now a maximal clique.

If there exists no clique in a graph with more vertices than a certain clique, that clique is called a **maximum clique** (MC). A maximum clique is always also a maximal clique. It is possible that there are multiple maximum cliques within a graph. Those multiple maximum cliques then need to have the same number of

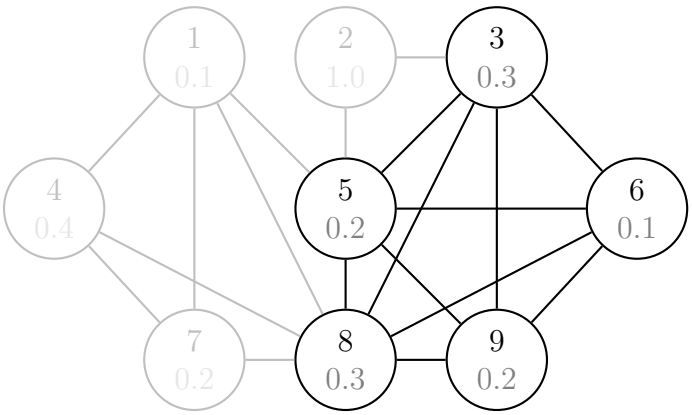

Figure 2.4: The maximum clique in the example graph

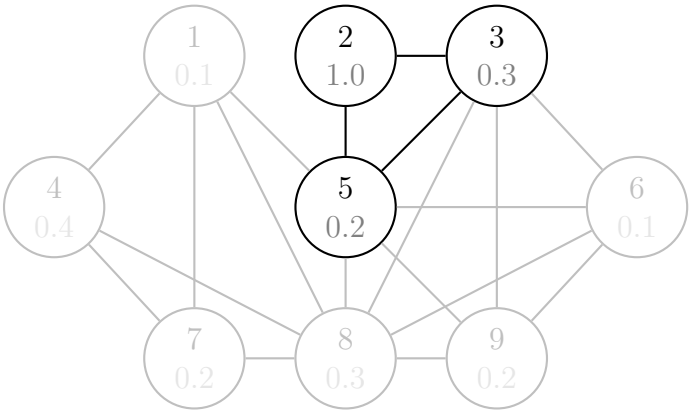

Figure 2.5: The maximum weight clique in the example graph

vertices by definition. The problem of finding the maximum clique within a graph is known as the **maximum clique problem** (MCP). In the example graph, there is one maximum clique which consists of v_3, v_5, v_6, v_8 and v_9. This maximum clique with 5 vertices is highlighted in figure 2.4. The maximal clique presented in figure 2.3 is not a maximum clique, as it only consists of 4 vertices.

The clique with the greatest sum of vertex weights is called the **maximum weight clique** (MWC). The problem of finding the maximum weight clique of a graph is called the **maximum weight clique problem** (MWCP). Given that the weights are greater than zero, the maximum weight clique will always be a maximal clique as well. It is however not necessary that the clique containing the most vertices, the maximum clique, is also the maximum weight clique. This is also the case in the example graph. The maximum clique highlighted in figure 2.4 has a weight of 1.1, but the maximum weight clique with a weight of 1.5 is formed out of $v_2^{1.0}$, $v_3^{0.3}$ and $v_5^{0.2}$, as shown in figure 2.5. Other maximal cliques in the graph with less weight than the maximum weight clique are the one shown in figure 2.3 with a weight of 1.0, as well as $v_1^{0.1}$, $v_5^{0.2}$ and $v_8^{0.3}$ with a weight of 0.6.

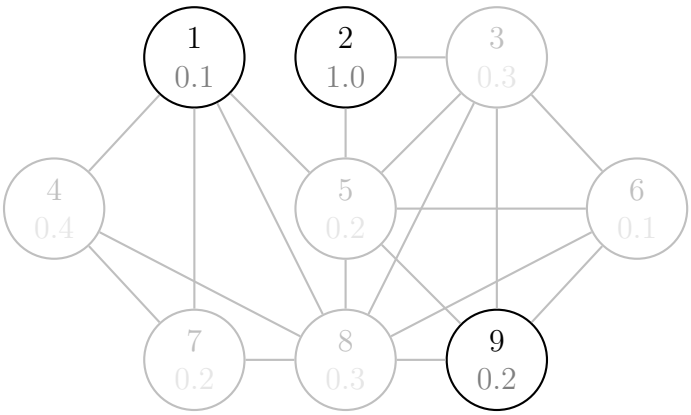

Figure 2.6: Example of an independent set

If it is mentioned that a clique C_1 is better than another clique C_2, this means that C_1 has a higher weight than C_2, i.e., $w(C_1) > w(C_2)$. Within algorithms the notation $\hat{C}$ is used for the best clique (i.e., with the greatest weight) that has been found so far. Similarly $\hat{w}(D)$ refers to the most weighted vertex within a set of vertices D. Note that D does not necessarily have to be a clique.

2.3 Independent sets

Contrary to a clique, an **independent set** D is a subset of the vertices V where every vertex is *not* connected to any other vertex in that independent set. Formally speaking, the condition for an independent set is:

$$\forall v_i, v_j \in D, i \neq j : (v_i, v_j) \notin E.$$

A commonly used abbreviation for "independent set" is IS. An example for an IS is provided in figure 2.6. It can be seen that the set $D_1 = \{v_1, v_2, v_9\}$ is an independent set, since v_1 is not adjacent to v_2 and v_9, while v_2 and v_9 are not adjacent either.

3 Algorithms

The subsequent sections will present the different classic and recent algorithms from the literature that are later compared in the experimental part. The algorithms are presented in the order of their publication. Some notations and namings will deviate from those used in the respective original papers, mostly for consistency reasons. Sometimes also slight restructurings were conducted to make matters more clear, parts of long procedures were moved to new subprocedures. Subprocedures of algorithms are suffixed by the name of the main algorithm to make their affiliation clear. Sometimes there are letters or other markers on the left side of an algorithms pseudo-code. They tag the start of a section of the algorithm that can be referenced within the explaining text. Those sections that structure the algorithm do not have any direct impact on the code itself and are just an orientation guide for the reader.

In general there are two types of algorithms: **heuristic** and **exact** ones. Exact algorithms always search for the optimal solution, i.e., the actual maximum weight clique. In the general case, finding the maximum weight clique is a NP-hard problem [19]. So depending on the input graphs, exact approaches can require prohibitively long computation times. Heuristic algorithms on the other hand try to find a clique of great weight in rather short time using some kinds of estimates. They can find a solution that is optimal or close to optimal, but may also be completely off in some cases. While in general it can be said that exact algorithms provide higher-quality results but are slower, there can also be cases where exact algorithms are faster than heuristic ones. Then again an exact algorithm may not find a solution in a reasonable time while a heuristic one finds the maximum weight clique quickly, thus delivering the higher quality result. Table 3.1 provides an overview of the heuristic and exact MWC algorithms that will be introduced here.

Heuristic algorithms	Exact algorithms
FastWClq	WLMC
SCCWalk	WC-WMC
SCCWalk4L	TCM-MWC
MWCRedu	MWCPeel

Table 3.1: Algorithms overview

3.1 Exact algorithms

3.1.1 WLMC

The exact algorithm WLMC [20] was presented in 2017 by Jiang, C.M. Li and Manyà. Main features introduced with it are its special preprocessing and vertex splitting strategies. It requires integer vertex weights. In general it makes use of a strategy called **branch-and-bound** (BnB). This encompasses exploring subproblems in a search tree. Branches of the tree that do not meet a certain bound, which is required for potential solutions, are discarded. The first mention of the BnB-scheme can be found at [21].

Main procedure

Algorithm 1 outlines the main procedure of WLMC. At first an initial clique C_0, an initial vertex ordering O_0 and a reduced Graph are determined by the helper function $InitializeWLMC$. O_0 is a degeneracy ordering of the vertices so vertices at the beginning of the ordering tend to have fewer neighbors than the ones at the end of the ordering. G' is a version of the original input graph G where all vertices u where $UB_0(u) \leq w(C_0)$ have been removed (cf. chapter 3). The best found clique $\hat{C}$ is initialized with C_0. The vertices of G' are prepared in V' in an ordered manner according to O_0 so that they can be used for looping through them in the following main loop.

In the for-loop of this main procedure of WLMC, searches for the maximum weight clique are conducted from each individual vertex v_i of the reduced graph as a respective starting point. It should be noted that the loop traverses V' in reverse, which means that vertices with many neighbors are processed earlier as starting vertices for a clique. Vertices with more neighbors are more likely to be in a large and thus high weight clique. This means the better cliques are often explored first. When $\hat{C}$ is always updated accordingly, the less good cliques that occur later can be detected as such and discarded ahead of further exploration.

At the start of each loop iteration a set $Candidates$ is formed which contains all the vertices that are allowed to be explored as potential clique members alongside v_i. If the $Candidates$ and v_i can potentially form a higher weigh clique than $\hat{C}$, the clique search in the subgraph $G[Candidates]$ is performed. Again the initialization procedure is used to get another initial clique C'_0, ordering O'_0 and reduced graph G'' specifically for the subgraph. As every member of the subgraph is a neighbor of v_i and v_i is not included in the subgraph itself, every clique in $G[Candidates]$ can be extended with v_i in the original graph. Therefore if C'_0 and v_i can together form a better clique than $\hat{C}$, the latter already gets updated.

Following that, in the most important part of the main loop, the highest weight clique C' within the reduced subgraph G'' (plus v_i) is searched for recursively by the function $SearchMaxWCliqueWLMC$. If a new best clique was found within the subgraph, $\hat{C}$ can be updated as well. When the loop is finished, all viable branches of the search space for a maximum weight clique have been explored and $\hat{C}$ is returned.

Algorithm 1: WLMC

```
(C_0, O_0, G') := InitializeWLMC(G, 0);
Ĉ := C_0;
V' := vertices of G';
order V' w.r.t. O_0;
for i := |V'| to 1 do
    Candidates := N(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V'|}};
    if w(Candidates) + w(v_i) > w(Ĉ) then
        (C'_0, O'_0, G'') := InitializeWLMC(G[Candidates], w(Ĉ) − w(v_i));
        if w(C'_0) + w(v_i) > w(Ĉ) then
            Ĉ := C'_0 ∪ {v_i};
        C' := SearchMaxWCliqueWLMC(G'', Ĉ, {v_i}, O'_0);
        if w(C') > w(Ĉ) then
            Ĉ := C';
return Ĉ;
```

Initialization

The initialization function is given in algorithm 2. From a graph G and a lower bound for the clique weight lb it derives an initial clique C_0 an ordering O_0 of the vertices according to their degrees and a reduced graph G'.

Section A determines a vertex degree ordering and the initial clique. The vertex set U is initialized as a copy of all vertices V in G. The degrees $deg(v)$ are computed by counting the number of neighbors of each vertex. In the following loop over all vertices, the vertex of minimal degree is chosen and stored as v_i where i is the loop counter variable. When the condition $\deg(v_i) = |U| - 1$ is satisfied, that means that v_i is adjacent to all other vertices in U. Since v_i has a minimal degree, the other vertices can not have fewer neighbors, nor can they have more as the maximum possible degree in U is $|U| - 1$. This means that in this situation every vertex in U is connected to every other one, which means that U is a clique. Therefore the initial clique C_0 is set to U. The elements of U are stored in the remaining $v_i, v_{i+1}, \ldots, v_{|V|}$. For the latter assignments the order does not matter anymore since all vertices in U

should have the same degree. After the ordered vertices and the clique are stored, the loop is terminated. In the cases where U is not yet a clique, v_i is removed from U. Subsequently the degree of every neighbor of v_i that is still in U is decreased by 1 to reflect the new degrees within the changed set U.

Following that in section B the lower bound for the clique weight lb is adapted to match the weight of the initial Clique in the case that it is smaller. Then the reduced graph G' is assembled. It is initialized as a copy of G. The actual reduction is performed by removing all vertices where the weight of the inclusive neighborhood $w(N[v])$ is not greater than lb. The ordering O_0 can be simply obtained by stringing together the v_i's that were collected before. Finally C_0, O_0 and G' are returned.

Algorithm 2: InitializeWLMC(G, lb)

```
A  U := V;
   deg(v) := |N(v)| ∀v ∈ U;
   for i := 1 to |V| do
       v_i := arg min_{v∈U} deg(v);
       if deg(v_i) = |U| − 1 then
           {v_i, v_{i+1}, ..., v_{|V|}} := elements of U in arbitrary order;
           C_0 := U;
           break;
       U := U \ {v_i};
       for v ∈ N(v_i) ∩ U do
           deg(v) := deg(v) − 1;
B  if w(C_0) > lb then
       lb := w(C_0);
   G' := G;
   for v ∈ V do
       if w(N[v]) ≤ lb then
           remove v and its incident edges from G';
   O_0 := {v_1, v_2, ..., v_{|V|}};
   return (C_0, O_0, G');
```

Clique search

Algorithm listing 3 shows the recursive function that WLMC uses to find the maximum weight clique within a subgraph. As parameters it takes a graph G in which the best clique is searched, the best clique $\hat{C}$ found so far, the clique C that is currently being built and an ordering O of the vertices in G. The return value is the best clique found in G or the $\hat{C}$ passed to the function, whichever is greater. As the

recursion level advances, the smaller the subgraph of the original input graph that is being passed to the clique search procedure will be.

The main task of section 1 is distributing all vertices V of the graph of into two disjunct sets A and B. A is chosen so that the weight of the best clique within it is not greater than $w(\hat{C}) - w(C)$. The remaining vertices B are called the branching vertices. The actual process of building the two sets is accomplished by the function $GetBranchesWLMC$. That function only returns B, but A can be trivially constructed as $A = V \setminus B$ has to hold. Furthermore possible early return scenarios are handled. If the subgraph passed to $SearchMaxCliqueWLMC$ does not contain any vertices, the search in the current branch is finished and the created clique C is returned. When no branching vertices can be found, the search can also be aborted. $A = V$ will hold when $B = \emptyset$. Given the properties of A that were mentioned earlier, this means that the current subgraph cannot contain a clique with a weight greater than $w(\hat{C}) - w(C)$. Due to this, a better clique than $\hat{C}$ will not be found in the current branch of the search, so $\hat{C}$ is returned.

After having the branching vertices in place, they are used in section 2 to explore different paths of the search tree and recursively build up the clique in different variants. Within their set $|B| = \{b_1, \ldots, b_{|B|}\}$ the branching vertices are arranged according to the Ordering O. This means that b_1 has the smallest degree while $b_{|V|}$ was one of the last remaining vertices in the initialization procedure when the vertex of the least degree was removed in every step. The ordering of B should already be produced like this by the function $GetBranchesWLMC$. In a loop that reversely runs through B from the last to the first element, each branching vertex is evaluated regarding its potential as a new member of the clique C. Similar to the loop in the main $WLMC$ procedure, B is iterated through backwards because vertices that have a higher position in the Ordering O should be more likely to be in a large clique. This again allows for an early discard of unnecessary exploration.

Within the loop, first a set $Candidates$ is defined that contains the vertices that the algorithm allows to be possibly added to C after adding b_i. These are all neighbors of b_i that are part of A or $\{b_j \in B \mid j > i\}$. Now the best case weight of a completed maximal clique with b_i added is determined, i.e., a developed version of C where b_i and all $Candidates$ could be added. If this best case weight is higher than the weight of $\hat{C}$, the recursion step is started. Now the actual best clique with b_i added is determined and stored as C'. For this, the clique search procedure $SearchMaxWCLiqueWLMC$ is called by itself, with the subgraph of G containing only the $Candidates$, the unchanged $\hat{C}$, a current clique consisting of C and $\{b_i\}$, as well as the also unchanged ordering O. If C' is better than $\hat{C}$, $\hat{C}$ is updated accordingly.

When the loop is finished, $\hat{C}$ is returned, which has been potentially updated by suitable cliques induced by "good" branching vertices.

Algorithm 3: SearchMaxWCliqueWLMC($G, \hat{C}, C, O$)

```
1 if V = ∅ then
      return C;
  B := GetBranchesWLMC(G, w(Ĉ) − w(C), O);
  if B = ∅ then
      return Ĉ;
  A := V \ B;
2 Let B = {b_1, b_2, ..., b_|B|}, b_1 < b_2 < ... < b_|B| w.r.t. O;
  for i := |B| to 1 do
      Candidates := N(b_i) ∩ ({b_{i+1}, b_{i+2}, ..., b_|B|} ∪ A);
      if w(C ∪ {b_i}) + w(Candidates) > w(Ĉ) then
          C' := SearchMaxWCliqueWLMC(G[Candidates], Ĉ, C ∪ {b_i}, O);
          if w(C') > w(Ĉ) then
              Ĉ := C';
  return Ĉ;
```

Branching

To obtain the branching vertices mentioned before, the function that can be found as algorithm 4 is used. The branching vertices are stored in the variable B. The process of determining B requires to build up some independent sets, who are in turn stored in one parent set Π. Both variables are initialized as empty sets. The actual algorithm encompasses two sections marked as 1 and 2.

Section 1 basically iterates through all vertices of the graph G that was passed to the function and inserts it either into an independent set within Π or into B. Until V is empty, inside the loop a vertex v is chosen from V that is the greatest according to the ordering O. Then v is removed from V. Now there are three options where to put v afterwards. The first option is to insert it into an existing independent set $D \in \Pi$. This is allowed if the definition of an independent set is not violated, i.e. none of the members of D are a neighbor of v. If the latter is not the case, the second option is to insert v into its own new independent set and add this new set to Π. Both the first and the second option are only permitted if the sum of the highest weight vertices of each set in the resulting Π would not exceed the parameter t. Otherwise the third option is chosen, and v is added to the branching vertices B.

After the initial creation of the sets, section 2 initializes the upper bound ub_0 for the maximum weight of a clique in $A = D_1 \cup \ldots \cup D_{|A|}$. Using the same formula as in some comparisons in section 1 of the algorithm, a simple upper bound is given by the sum of the maximum weights of each IS in Π. This upper bound is not always tight. The most weighted vertices $\hat{w}(D_j)$ or even any vertex from every IS $D_j \in \Pi$

may not actually form a clique together. In such cases, Π is called **conflicting**. Some members of B can then be inserted into Π, while the upper bound based on the cliques that are actually possible to build still does not exceed t. The for-loop goes through each branching vertex b_i and checks whether it could be inserted into Π. For this the function $UP\&SplitWLMC$ is used. It gets passed a union of Π and $\{b_i\}$ and the upper bound of the weight of a clique containing vertices from Π and b_i. The latter equals to $ub_0 + w(b_i)$ according to our best knowledge up until now. $UP\&SplitWLMC$ now determines a tighter upper bound ub and rearranges the independent sets to Π' in the process. If ub is then below or equal to t, that means that b_i should not be considered as a branching vertex. Remember that $t = w(\hat{C}) - w(C)$ when $GetBranchesWLMC$ was called. This means that a further exploration with b_i cannot lead to new best clique. So if $ub \leq t$, the updated ub is used for ub_0, Π' replaces the previous Π, and b_i is removed from B. After the loop is finished, the refined B is returned. It should now only contain branching vertices worth exploring.

Algorithm 4: GetBranchesWLMC(G, t, O)

```
B := ∅ ;                                                    // branching vertices
Π := ∅ ;                                                    // set of ISs
1 while V ≠ ∅ do
    v := greatest vertex of V w.r.t. O;
    V := V \ {v};
    if ∃D ∈ Π | (N(v) ∩ D = ∅) ∧ (Σ_{j=1}^{|Π|} ŵ(D_j) ≤ t | after adding v into D)
     then
        D := D ∪ {v};
    else if Σ_{j=1}^{|Π|} ŵ(D_j) + w(v) ≤ t then
        D := {v};
        Π := Π ∪ {D};
    else
        B := B ∪ {v};
2 ub_0 := Σ_{j=1}^{|Π|} ŵ(D_j);
  Let B = {b_1, b_2, ..., b_{|B|}}, b_1 < b_2 < ... < b_{|B|} w.r.t. O;
  for i := |B| to 1 do
    (ub, Π') := UP&SplitWLMC(G, Π ∪ {{b_i}}, ub_0 + w(b_i), t);
    if ub ≤ t then
        ub_0 := ub;
        Π := Π';
        B := B \ {b_i}
  return B;
```

Example For a simple example, which is also presented in [20] itself, we assume that the subgraph passed to $GetBranchesWLMC$ is the graph G that is displayed in figure 3.1. The given ordering O sorts the vertices according to their index, so that $v_1 < v_2 < v_3 < v_4 < v_5 < v_6$. Furthermore $t = 6$ is provided, which again was derived by the calling function from $t = w(\hat{C}) - w(C)$.

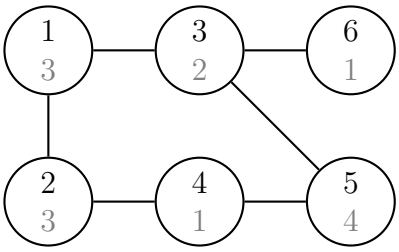

Figure 3.1: Example graph G for WLMC

The iteration steps of the loop in section 1 of algorithm 4 are visualized in figure 3.2. At first the greatest unprocessed vertex with respect to O is v_6. No ISs are present yet in Π, so the vertex is inserted into its own new IS D_1. In iteration (2) v_5 is picked. It is not adjacent to v_6, so it can be added to D_1 as well. Note that since v_5^4 is the new highest weight vertex in D_1, the upper bound $\sum_{j=1}^{|\Pi|} \hat{w}(D_j)$ increases to 4. During the next step v_4 is processed. It is a neighbor of v_5, thus it cannot be added to D_1, but instead requires its own IS D_2. The upper bound consequently rises to 5. Iteration (4) clearly assigns v_3 to D_2, since the IS D_1 only contains neighbors of v_3. This changes the upper bound to 6 as v_3^2 is the new most weighted member of D_2 now. Next, v_2 can be added to D_1, leaving the upper bound unchanged.

Probably the most interesting iteration is (6), where v_1^3 is taken into account. Both members of D_1 and D_2 are adjacent to it, so it would require a new IS. However that is not allowed either because we already have $\sum_{j=1}^{|\Pi|} \hat{w}(D_j) = 6 = t$. Adding $w(v_1^3)$ to that would exceed t. Therefore it is the first (and only) vertex which gets added to the branching vertices B.

Now for section 2 of the algorithm, we have $\Pi = \{\{v_6^1, v_5^4, v_2^3\}, \{v_4^1, v_3^2\}\}$, $ub_0 = 6$ and $B = \{v_1^3\}$. The for-loop that tries to reduce both B and ub_0 will only have one iteration which takes a look at $b_1 = v_1^3$. The function $UP\&SplitWLMC$ will prove that the combined set $\Pi \cup \{v_1^3\}$ does have an upper bound of $ub = 6$ instead of $ub_0 + w(v_1^3) = 9$ as the initial bound estimation would suggest. The details of this upper bound improvement will be discussed in the example of the following section of the thesis. As $ub = 6 \leq t$, v_1^3 can be removed from B. This ultimately leads to $GetBranchesWLMC$ returning an empty set of branching vertices.

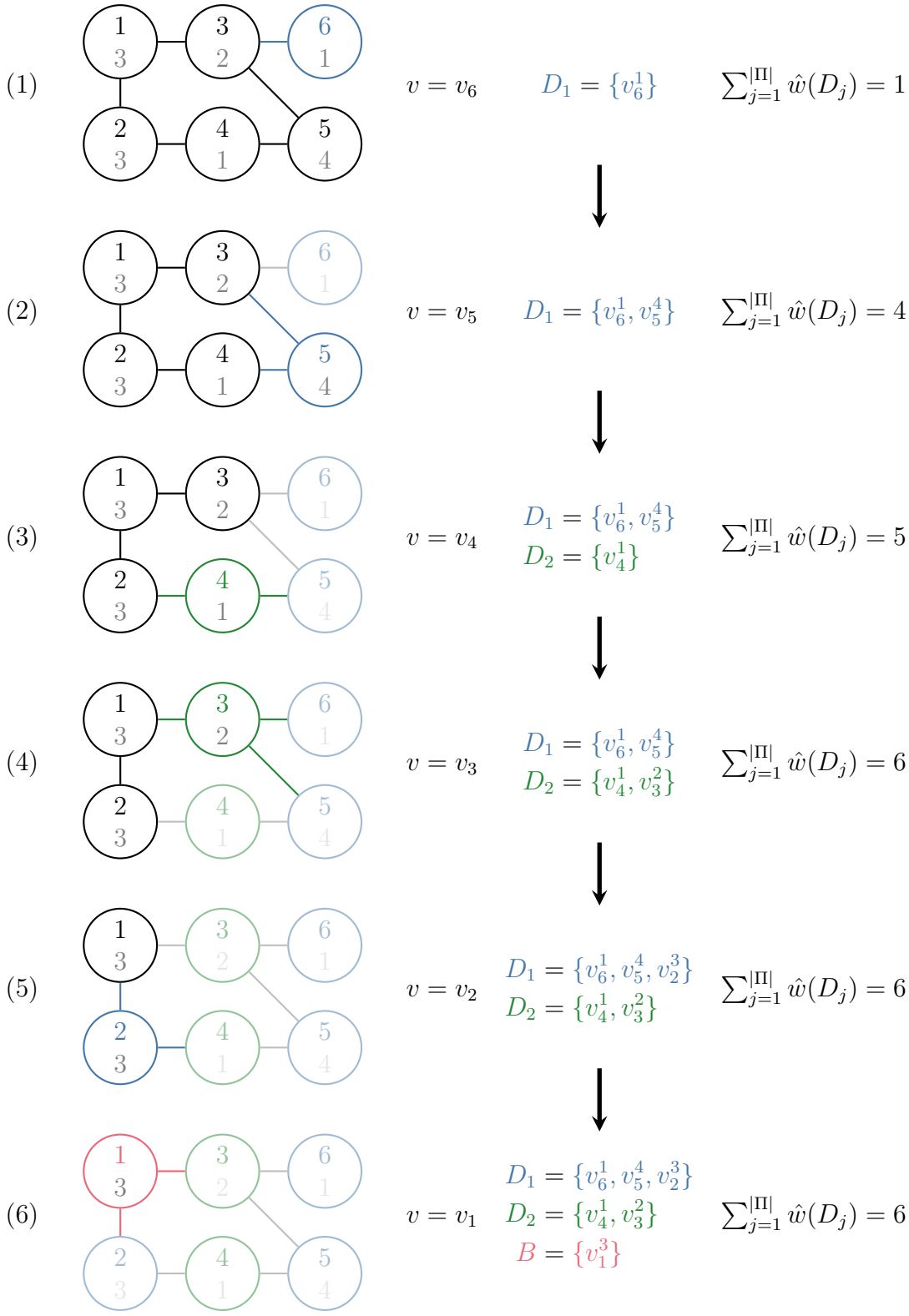

Figure 3.2: Initial IS creation on the example graph G

Unit IS propagation

The main task of the function $UP\&SplitWLMC$ listed as algorithm 5 is to determine an accurate upper bound of the weight that a clique built from the ISs Π can have. As already mentioned, the initial upper bound estimation $ub_0 = \sum_{j=1}^{|\Pi|} \hat{w}(D_j)$ assumes the extreme case where the most weighted vertices of every set in Π form a clique together. This function now tries to find out conflicts between the ISs, i.e., situations where no such clique can be created. As long as one is available, the while-loop takes a unit IS $\{v\}$ that is not marked. A unit IS is an independent set with exactly one vertex v. Each IS of Π has a flag that encodes whether it is marked. This flag is set later in the course of $UP\&SplitWLMC$ to not consider fully processed sets again. It is not set or read in any other function, but is passed around together with the ISs when they are returned and given to another call of this function again.

If the initial upper bound ub_0 would be tight, the only vertex v of the chosen unit IS must be in a clique with the to-be-selected vertices from the other ISs. Thus all vertices not adjacent to v are removed from the other sets. If there are no conflicts between the sets, this removal should not change the maximum possible weight and size of a clique. If that is however not the case, the algorithm differentiates between two different scenarios.

Scenario 1: empty set The condition labeled with 1 in the listing represents the first scenario, where an IS S_0 becomes empty from the removal. All vertices that were originally included in S_0 were then removed by v and potentially some other unit ISs in previous iterations. Vertices removed from any IS get restored. Those ISs that caused the removals from S_0 are called $S_1, \ldots, S_r$. That S_0 got completely empty means that it could not contribute to a clique if the ISs $S_1, \ldots, S_r$ are already part of the clique. This in turn means that the initial upper bound estimation was too high, since it could not include the highest vertex weights of both S_0 and all of $S_1, \ldots, S_r$ at the same time. So the upper bound would have to be decreased by at least the weight of the "worst" highest weight vertex of an IS in $\{S_0, \ldots, S_r\}$. Formally this difference is expressed as $\delta = min(\hat{w}(S_0), \ldots, \hat{w}(S_r))$.

Following that in section 1.1 the sets $S_0, \ldots, S_r$ are split up. The split of an IS S_j happens on a vertex level. A vertex $u_i^x \in S_j$ can be split into $u_i^y \in S'_j$ and $u_i^z \in S''_j$ where $y + z = x$. So the weight of the original vertex is distributed among its counterparts in the new sets S'_j and S''_j. More precisely, the $splitWLMC$ function puts the parts of the individual vertex weights up to δ into S'_j and the parts of the weight above δ into S''_j. If for a vertex $u \in S_j$ its weight does not exceed δ, i.e., $w(u) \leq \delta$, it can be copied to S'_j without changes and can be omitted in S''_j. To complete the splitting, each S'_j is added to Δ and each S''_j replaces its corresponding S_j in Π. Finally, the upper bound estimation ub is corrected by δ.

Scenario 2: most weighted vertices removed Besides condition 1, the second scenario handled by condition 2 is that an IS S_0 still contains some vertices, but its k vertices with the highest weights got removed by some unit ISs $S_1, \ldots, S_r$. That would also mean that ub cannot be tight, as the highest weight vertex of S_0 is assumed to be included in the best clique if $ub = \sum_{j=1}^{|\Pi|} \hat{w}(D_j)$. This can clearly not be the case if every other $\hat{w}(S_i) \mid 1 \leq i \leq r$ is in the clique too. Remember that every S_i was a unit IS $\{v_i\}$ at the moment of the vertex removal from S_0. So v_i has to be a neighbor of the vertex of S_0 that gets chosen for the clique. If that chosen vertex is not the highest weighted in S_0, the upper bound could be decreased accordingly.

The steps executed when condition 2 is fulfilled have a structure similar to the ones from 1. Again the removed vertices are restored. The vertices within the set S_0 are addressed in a way so that they are ordered by weight decreasing from the first element $u_1^{w_1}$ to the last one $u_{|S_0|}^{w_{|S_0|}}$. This ordering implicitly makes the k vertices which were previously removed also the first k vertices in the restored set. Similar to the first scenario a value $\delta = \min(\beta, \hat{w}(S_1), \ldots, \hat{w}(S_k))$ is calculated that denotes the needed change in the upper bound. Instead of $\hat{w}(S_0)$, another value $\beta = w_1 - w_{k+1}$ is used for obtaining δ. In scenario 2 with the most weighted vertices missing, so the possible the contribution of S_0 to the clique would not fully diminish due to the conflicts. Therefore β is only the difference between the highest originally included weight w_1 and the highest weight of the not removed vertices w_{k+1}. The latter weight is furthermore stored as γ.

Section 2.1 of the algorithm is mostly equivalent to 1.1. The only difference is that S_0' and S_0'' are created in a special manner unlike the other split sets generated by $splitWLMC$. S_0' is only filled with the k most weighted vertices and the weight of vertex $i, 1 \leq i \leq k$ is set to $\min(\delta, w_i - \gamma)$, so the minimum of either the upper bound correction value δ or the difference between the original weight of the vertex and the weight of the most weighted vertex that did not get removed. S_0'' contains all vertices present in S_0 with the the weights set to the parts of the original weights that are not already covered in S_0'.

Concluding steps Independent of conditions 1 and 2, if the corrected upper bound ub becomes smaller or equal to t, all members of the set Δ are marked and the loop is terminated. The function has then proved that a potential branching vertex b_i that was included in the parameter Π is not useful as a branching vertex.

After the loop, all removed vertices that have not yet been restored to their ISs are added to them again. Finally the improved upper bound and the updated set of ISs obtained by the union of δ and Π are returned.

Example To make the process more tangible, the example from the previous section about the branching procedure can be continued here. As an input of

Algorithm 5: UP&SplitWLMC(G, Π, ub, t)

$\Delta := \emptyset$;
while $\exists$ *unit IS* $\{v\} \in \Pi \mid \{v\}$ *is not marked* **do**
  remove vertices non-adjacent to v from their IS;
1 **if** $\exists S_0 \in \Pi \mid S_0 = \emptyset \wedge S_0$ *is not marked* **then**
    restore all removed vertices into their IS;
    Let $S_1, \ldots, S_r$ be the ISs responsible of removing all vertices from S_0;
    $\delta := min(\hat{w}(S_0), \ldots, \hat{w}(S_r))$;
1.1 **for** $S_j \in \{S_0, S_1, \ldots, S_r\}$ **do**
      $(S'_j, S''_j) := splitWLMC(S_j, \delta)$;
    $\Delta := \Delta \cup \{S'_0, S'_1, \ldots, S'_r\}$;
    $\Pi := (\Pi \setminus \{S_0, S_1, \ldots, S_r\}) \cup \{S''_0, S''_1, \ldots, S''_r\}$;
    $ub := ub - \delta$;
2 **else if** $\exists S_0 \in \Pi \mid S_0$ *is not marked* $\wedge$ *the* k *most weighted vertices are removed from* S_0 **then**
    Let $S_0 = \{u_1^{w_1}, \ldots, u_k^{w_k}, \ldots, u_{|S_0|}^{w_{|S_0|}}\} \mid w_1 \geq \ldots \geq w_k \geq \ldots w_{|S_0|}$;
    Let $u_1^{w_1}, \ldots, u_k^{w_k}$ be the k most weighted vertices removed from S_0;
    $\beta := w_1 - w_{k+1}$;
    restore all the removed vertices into their IS;
    Let $S_1, S_2, \ldots, S_r$ be the ISs responsible of removing $u_1^{w_1}, \ldots, u_k^{w_k}$ from S_0;
    $\delta := min(\beta, \hat{w}(S_1), \ldots, \hat{w}(S_r))$;
    $\gamma := w_{k+1}$;
2.1 $S'_0 := \{u_1^{min(\delta, w_1 - \gamma)}, \ldots, u_k^{min(\delta, w_k - \gamma)}\}$;
    Let $w''_j = w_j - min(\delta, w_j - \gamma) \mid 1 \leq j \leq k$;
    $S''_0 := \{u_1^{w''_1}, \ldots, u_k^{w''_k}, u_{k+1}^{w_{k+1}}, \ldots, u_{|S_0|}^{w_{|S_0|}}\}$;
    for $S_j \in \{S_1, S_2, \ldots, S_r\}$ **do**
      $(S'_j, S''_j) := splitWLMC(S_j, \delta)$;
    $\Delta := \Delta \cup \{S'_0, S'_1, \ldots, S'_r\}$;
    $\Pi := (\Pi \setminus \{S_0, S_1, \ldots, S_r\}) \cup \{S''_0, S''_1, \ldots, S''_r\}$;
    $ub := ub - \delta$;
3 **if** $ub \leq t$ **then**
    mark all ISs in Δ;
    break;
restore all the removed vertices into their ISs;
return $(ub, \Delta \cup \Pi)$;

UP&SplitWLMC we have G as seen in figure 3.1, $\Pi = \{\{v_6^1, v_5^4, v_2^3\}, \{v_4^1, v_3^2\}, \{v_1^3\}\}$, $ub = 9$ and $t = 6$.

Starting into the loop, the unit IS $\{v_1^3\}$ can be found within Π. Therefore its non-neighbors v_6^2, v_5^4 and v_4^1 are removed from their independent sets. This results in $\Pi = \{\{v_2^3\}, \{v_3^2\}, \{v_1^3\}\}$. That in turn means that there is a new unit IS $\{v_2^3\}$. Removing its non-neighbor v_3^2 from Π yields $\Pi = \{\{v_2^3\}, \emptyset, \{v_1^3\}\}$. Now the second IS in Π is empty, thus the condition labeled with 1 in algorithm 5 is triggered. After restoring the removed vertices, the sets can be labeled as

$$S_0 = \{v_4^1, v_3^2\} \qquad S_1 = \{v_1^3\} \qquad S_2 = \{v_6^1, v_5^4, v_2^3\}.$$

With that we can calculate

$$\delta = min(\hat{w}(S_0), \hat{w}(S_1), \hat{w}(S_2)) = min(2, 3, 4) = 2.$$

Now the splits are executed (section 1.1 in the algorithm listing). The resulting sets are:

$$\begin{array}{lll} S_0' = \{v_4^1, v_3^2\} & S_1' = \{v_1^2\} & S_2' = \{v_6^1, v_5^2, v_2^2\} \\ S_0'' = \emptyset & S_1'' = \{v_1^1\} & S_2'' = \{v_5^2, v_2^1\}. \end{array}$$

The sets of ISs are updated with the newly split up independent sets:

$$\begin{aligned} \Pi &= \{\{v_1^1\}, \{v_5^2, v_2^1\}\} \\ \Delta &= \{\{v_4^1, v_3^2\}, \{v_1^2\}, \{v_6^1, v_5^2, v_2^2\}\}. \end{aligned}$$

Finally the upper bound can be updated: $ub = ub - \delta = 9 - 2 = 7$. This does not yet reach $t = 6$, so the loop can proceed to its next iteration. The unit IS $\{v_1^1\}$ in the new Π leads to the removal of v_5^2, resulting in $\Pi = \{\{v_1^1\}, \{v_2^1\}\}$. The removed v_5^2 was the most weighted vertex in the second IS in Π, so the condition labeled as 2 in the algorithm is fulfilled, with $k = 1$. We label the sets

$$S_0 = \{v_5^2, v_2^1\} = \{u_k^{w_k}, u_{k+1}^{w_{k+1}}\} \qquad S_1 = \{v_1^1\}.$$

With the sets in place, the following calculations can be made:

$$\begin{array}{llll} \beta = w_1 - w_{k+1} & = w(v_5^2) - w(v_2^1) & = 2 - 1 & = 1 \\ \delta = min(\beta, \hat{w}(S_1)) & = min(\beta, w(v_1^1)) & = min(1, 1) & = 1 \\ \gamma = w_{k+1} & = w(v_2^1) & & = 1. \end{array}$$

The next step is the splitting as seen in section 2.1 of the listing. Applying the procedure to the set S_0 yields

$$\begin{array}{llll} S_0' = \{v_5^{min(\delta, w_k - \gamma)}\} & = \{v_5^{min(1, 2-1)}\} & = \{v_5^1\} \\ S_0'' = \{v_5^{w_k - min(\delta, w_k - \gamma)}, v_2^1\} & = \{v_5^{2-min(1,2-1)}, v_2^1\} & = \{v_5^1, v_2^1\}. \end{array}$$

For S_1 the split can be done using the regular $splitWLMC$ function which returns

$$\begin{aligned} S_1' &= \{v_1^1\} \\ S_1'' &= \{v_1^0\} \sim \emptyset. \end{aligned}$$

With that in place Π and Δ are updated again.

$$\Pi = \{\{v_5^1, v_2^1\}\}$$
$$\Delta = \{\{v_4^1, v_3^2\}, \{v_1^2\}, \{v_6^1, v_5^2, v_2^2\}, \{v_5^1\}, \{v_1^1\}\}.$$

Updating the upper bound like $ub = ub - \delta = 7 - 1 = 6$ makes it equal to t. This means that the condition labeled with 3 in the algorithm listing is fulfilled. Thus we mark all the sets in Δ (depicted here by underlining) and exit the loop. The final return value pair of the function will then be:

$$\left(6, \{\underline{\{v_4^1, v_3^2\}}, \underline{\{v_1^2\}}, \underline{\{v_6^1, v_5^2, v_2^2\}}, \underline{\{v_5^1\}}, \underline{\{v_1^1\}}, \{v_5^1, v_2^1\}\}\right).$$

It should be noted that as no ISs were marked in the given input parameter Π of the function, the marking states did not have an influence on the conditions 1 and 2 in algorithm 5. In cases where there are multiple iterations in the for-loop of the *GetBranchesWLMC* function, *UP&SplitWLMC* might operate on independent sets that it already modified by itself in a previous call. Then there could already be marked ISs within Π that *UP&SplitWLMC* has to deal with.

Vertex splitting

The split function presented in algorithm 6 accomplishes the task of dividing an independent set S according to a weight threshold δ. The first returned set S' contains all the same vertices that are in S, but all weights that are greater than or equal to δ are changed to δ. Thus the maximum weight that can possibly occur in S' is δ. According to the notation in the listing, the vertices where the weights get adjusted are $\{u_1, \ldots, u_k\}$. The second returned set S'' only contains the vertices $\{u_1, \ldots, u_k\}$ where the original weight was greater than δ. All weights in S'' are adapted to $w_i - \delta \mid 1 \leq i \leq k$. It is apparent that the sum of the weights that each vertex has in S' and S'' (adding zero for S'' if the vertex is not in it) is equivalent to the original weight in S. The other way round, it can be said that the weights of S have been distributed or split to S' and S''.

Algorithm 6: SplitWLMC(S, δ)

Let $S = \{u_1^{w_1}, \ldots, u_{|S|}^{w_{|S|}}\} \mid w_1 \geq \ldots \geq w_k \geq \delta \geq w_{k+1} \geq \ldots w_{|S|}$;
$S' = \{u_1^{\delta}, \ldots, u_k^{\delta}, u_{k+1}^{w_{k+1}}, \ldots, u_{|S|}^{w_{|S|}}\}$;
$S'' = \{u_1^{w_1-\delta}, \ldots, u_k^{w_k-\delta}\}$;
return (S', S'');

Example To better visualize the split function, an example how it transforms its input is shown in figure 3.3. The parameters are $S = \{v_6^1, v_5^4, v_2^3\}$ and $\delta = 2$. S did

already occur as S_2 during the first splits in the example of the previous chapter. In the figure the weights are depicted by the number of squares above the corresponding vertex. It can be seen that the parts of the weights up to the threshold δ, marked as ■, go to S'. v_5^4 and v_2^3 have weights that reach above δ. The remaining parts of their weights that were not included in S' are marked as ■ and get added to S''. v_6 is omitted from S'' because its original weight 1 is already below δ.

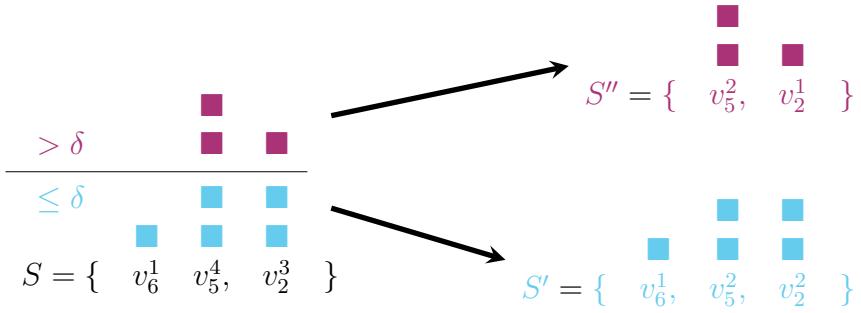

Figure 3.3: Example vertex splitting

3.1.2 WC-MWC

WC-MWC stands for **w**eight **c**over **m**aximum **w**eight **c**lique and is an exact state of the art algorithm for medium and large graphs [18]. It was published in 2018 and authored by C.M. Li, Liu, Jiang, Manyà and Y. Li.

As WC-MWC is based on the works of WLMC [20] and a branch-and-bound algorithm too, the general structure of it shows great similarities to the one of WLMC. For the initialization, the exact same function $InitializeWLMC$ is used again. The main procedure $WC\text{-}MWC$ (algorithm 27, appendix 7.2)) and the clique search function $SearchMaxWCliqueWC\text{-}MWC$ (algorithm 28, appendix 7.2)) are almost equivalent to their counterparts $WLMC$ and $SearchMaxWCliqueWLMC$. The only significant difference is that they call the respective sub-functions of WC-MWC instead of those from WLMC. The most interesting difference is at the core of the algorithm the determination of the interesting branches that should be explored recursively in the clique search function. WLMC uses $GetBranchesWLMC$ to obtain a set B of branching vertices and trivially infers $A = V \setminus B$ from that. WC-MWC on the other hand uses its function $PartitionWC\text{-}MWC$ that directly returns A and B. The following subchapter will explain the concept of the weight cover. This construct is in turn used by $PartitionWC\text{-}MWC$, which will be described in the next but one subchapter.

Weight Cover

A weight cover $\Pi = \{(D_1, w_1).(D_2, w_2), \ldots, (D_r, w_r)\}$ is a set of special pairs (D_i, w_i). In those pairs D_i is an independent set of the graph G that the weight cover refers to. w_i is a weight function where

$$\forall v \in D_i : w_i(v) > 0$$
$$\forall v \notin D_i : w_i(v) = 0.$$

A further condition for the weight function is that they "cover" the weight of each vertex, i.e. the sum of the weight functions yields the original weight of the vertices:

$$\forall v \in V : w(v) = \sum_{i=1}^{r} w_i(v).$$

Such a weight cover can be used to obtain an upper bound $UB_{WC}(G)$ for the weight of a clique within G. It is defined as the sum of the highest vertex weights of each IS D_i according to the weight functions w_i:

$$UB_{WC}(G) = \sum_{i=1}^{r} \hat{w}_i(D_i).$$

By the definition of an independent set, Each IS D_i can at most contain one member of any clique C. The definition of the weight cover requires that the weight of each vertex is preserved by the sum of its weights according to every w_i. Furthermore it is trivial that $w_i(u) \leq \hat{w}_i(D_i) \mid u \in C \cap D_i$. Combining these three facts, it becomes clear that the weight of any clique cannot surpass UB_{WC}.

Example The concept will now be illustrated on the example graph from figure 3.4, which can also be found in the original paper [18].

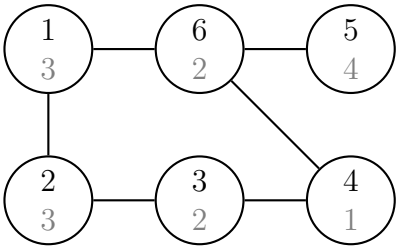

Figure 3.4: Example graph G for WC-MWC

A possible weight cover for G would be

$$\Pi = \{(D_1, w_1), (D_2.w_2), (D_3, w_3)\}.$$

The independent sets in this case are

$$D_1 = \{v_1, v_3, v_5\} \qquad D_2 = \{v_2, v_4, v_5\} \qquad D_3 = \{v_2, v_6\}.$$

A comparison with figure 3.4 can quickly verify that none of the vertices which are within the same D_i are neighbors. Finally the weight functions for the weight cover should be

$$w_1(v) = \begin{cases} 3 & v = v_1 \\ 2 & v = v_3 \\ 3 & v = v_5 \\ 0 & else \end{cases} \qquad w_2(v) = \begin{cases} 1 & v = v_2 \\ 1 & v = v_4 \\ 1 & v = v_5 \\ 0 & else \end{cases} \qquad w_3(v) = \begin{cases} 2 & v = v_2 \\ 2 & v = v_6 \\ 0 & else \end{cases}.$$

Written in a more compact manner that combines the ISs with the weight functions we have

$$D_1 = \{v_1^3, v_3^2, v_5^3\} \qquad D_2 = \{v_2^1, v_4^1, v_5^1\} \qquad D_3 = \{v_2^2, v_6^2\}.$$

Furthermore the Π from the example satisfies the condition that the sum of the weights according to the individual weight functions preserves the original vertex weights. This can be seen in figure 3.5, where the number of squares (■) directly above or below the nodes visualizes the weight of the node. The color of the squares on the right side of the figure shows by which weight function that part of the weight has been covered. ■ indicates that the part of the weight is covered by w_1, ■ is used for weight parts covered by w_2 and ■ for the ones covered by w_3.

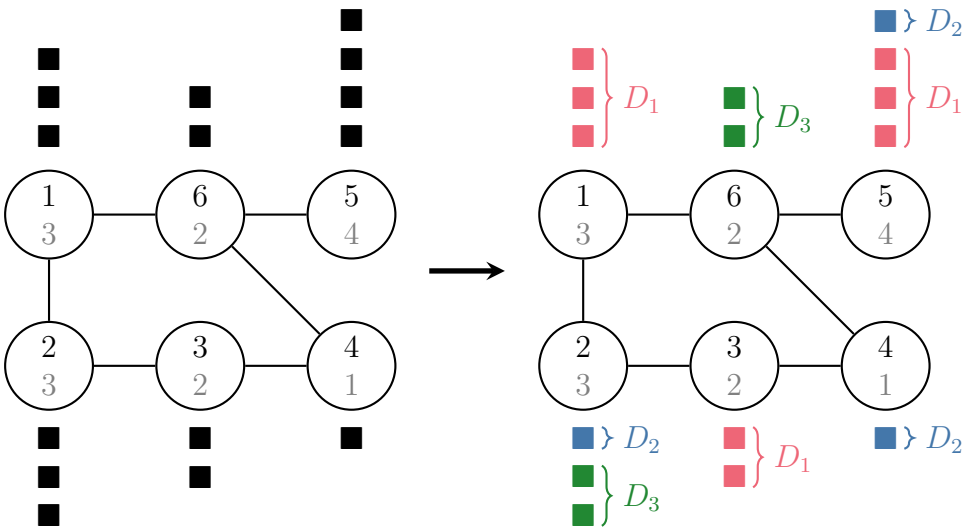

Figure 3.5: Weight cover for example graph

With the weight cover in place, the upper bound $UB_{WC}(G)$ for a clique in G can be calculated as well like

$$UB_{WC}(G) = \sum_{i=1}^{r} \hat{w}_i(D_i) = \hat{w}(D_1) + \hat{w}(D_2) + \hat{w}(D_3) = 3 + 1 + 2 = 6.$$

A look at the example graph reveals that there are no cliques with more than two members. The two pairs of adjacent vertices where the sum of the weights is the highest are (v_1^3, v_2^3) and (v_5^4, v_6^2). Those two pairs are thus the maximum weight cliques and both have a clique weight of 6. Since $UB_{WC}(G)$ is 6 as well, the upper bound is correct and even tight for the example graph.

Partitioning

Algorithm listing 7 shows the partitioning procedure of WC-MWC. It builds a weight cover, whose ISs $D_1, \ldots, D_t$ are stored as Π. Furthermore the branching vertices are maintained in A and the remaining vertices where a branching in the clique search is not required in B. Initially all sets are empty, they will be gradually filled during the for-loop that iterates through all vertices $v_i \in V$ of the given subgraph G. It is assumed that V is ordered so that $v_1 < v_2 < \ldots < v_{|V|}$ according to O. In each iteration a value β and a set of ISs Δ are introduced. The latter one will be filled with the ISs that v_i can be added to. The main idea of each iteration will be that the algorithm tries to integrate v_i into the weight cover without exceeding the clique weight limit t. If that is possible, v_i can be added to A, if not it becomes one of the branching vertices B.

Preparation phase Section 1 of the algorithm can be seen as a preparation phase where information is gathered that can be later used to decide if and how to integrate v_i into the weight cover Π. It loops through all independent sets that are already present in Π. In 1.1 the set D_j that is currently visited by the loop is added to Δ if it contains no neighbors of v_i. Otherwise the else branch 1.2 is triggered. There the highest weight neighbor u of v_i in D_j is determined. β gets updated so that at the end of the loop it will be equivalent to the maximum weight difference between $u \in D_j$ and the highest-weight member of D_j among all $D_j \in \Pi \setminus \Delta$. x is also updated accordingly so that it always represents the index of the IS D_j that the current value of β relates to. The last step within the inner loop is the break condition 1.3 that exits the loop to avoid unnecessary iterations. It checks whether the currently determined Δ and β provide enough information to be sure that v_i can be integrated into the weight cover without exceeding the weight limit t, i.e. v_i can be part of A. The detailed reasoning behind the formula will become clear in later stages of the partition function.

Integration phase Section 2 uses the information gathered in section 1 to put v_i into A and integrate it into the weight cover Π, or put it into the set of branching vertices B. For the decision what to do it uses the same condition as seen in 1.3. The term of the left side of the condition represents the upper bound of the clique weight according to the weight cover after v_i was inserted into it:

$$\underbrace{\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j)}_{\text{current weight cover}} \underbrace{+ \max\left(w(v_i) - \left(\sum_{D_k \in \Delta} \hat{w}_k(D_k) + \beta\right), 0\right)}_{\text{weight change when inserting } v_i}.$$

Isolating the first part of that term

$$\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j),$$

it can be seen that this is equivalent to the upper bound UB_{WC} presented in chapter 3.1.2. This represents the upper bound for the clique weight of the unmodified weight cover, i.e. where v_i is not yet inserted. The interesting part is the first operand of the *max*-operation in the second part of the term:

$$w(v_i) - \Big(\underbrace{\sum_{D_k \in \Delta} \hat{w}_k(D_k)}_{\substack{\text{weight that can} \\ \text{be covered by} \\ \text{existing ISs} \\ (2.2)}} + \underbrace{\beta}_{\substack{\text{weight that} \\ \text{can be} \\ \text{covered by} \\ \text{modified IS} \\ (2.1)}}\Big).$$

In the "worst" case, the whole weight $w(v_i)$ of v_i has to be added to a new separate IS. However unless Δ is empty, parts of the weight of v_i can be distributed into the ISs of Δ. When adding a vertex to an IS D_i and the weight of the added vertex according to w_i does not exceed the current maximum weight $\hat{w}_i(D_i)$ of that set, UB_{WC} will not change. Therefore, up to $\hat{w}_k(D_k), D_k \in \Delta$ of $w(v_i)$ can be covered by the IS D_k without increasing the maximum clique weight.

When slightly modifying Π, also the previously determined β can be covered by it. This modification is conducted in section 2.1. To recap, β was calculated within an independent set D_x that contained at least one neighbor of v_i. Given that D_x contains members with more weight than the highest-weight neighbor of v_i, D_x can be split up into two ISs where one of them contains no neighbors of v_i. In the following explanation the members of D_x and their weights are denoted as

$$D_x = \{v_{x_1}^{w_x(v_{x_1})}, \ldots, v_{x_a}^{w_x(v_{x_a})}, v_{x_{a+1}}^{w_x(v_{x_{a+1}})}, \ldots, v_{x_b}^{w_x(v_{x_b})}\}.$$

Furthermore in this notation vertices within D_x are sorted by their weight according to w_x in descending order:

$$w_x(v_{x_1}) \geq \ldots \geq w_x(v_{x_a}) \geq w_x(v_{x_{a+1}}) \geq \ldots \geq w_x(v_{x_b}).$$

The highest-weight neighbor of v_i within D_x is labeled as x_a:

$$v_{x_a} = \arg\max_{z \in N(v_i)} w_x(z).$$

x_a corresponds to the vertex u defined section 1.2 for the IS D_x. This yields its relation to β:

$$\beta = \hat{w}_x(D_x) - w_x(v_{x_a}) = w_x(v_{x_1}) - w_x(v_{x_a}). \qquad (*)$$

The actual modification of Π is now to split D_x into $D_{x'}$ and $D_{x''}$. $D_{x'}$ contains the same vertices as D_x, but their weights are clipped so that they do not exceed $w_x(v_{x_a})$:

$$D_{x'} = \{v_{x_1}^{w_x(v_{x_a})}, \ldots, v_{x_a}^{w_x(v_{x_a})}, v_{x_{a+1}}^{w_x(v_{x_{a+1}})}, \ldots, v_{x_b}^{w_x(v_{x_b})}\}.$$

The remaining weights that were clipped off in $D_{x'}$ are covered by $D_{x''}$, which only contains the vertices $v_{x_1}, \ldots, v_{x_{a-1}}$:

$$D_{x''} = \{v_{x_1}^{w_x(v_{x_1}) - w_x(v_{x_a})}, \ldots, v_{x_{a-1}}^{w_x(v_{x_{a-1}}) - w_x(v_{x_a})}\}.$$

Since the vertices within D_x were ordered by their weight and v_{x_a} is the highest-weight neighbor of v_i, it becomes clear that $D_{x''}$ contains no neighbors of v_i anymore. Therefore $\hat{w}_x(D_{X''}) = w_x(v_{x_1}) - w_x(v_{x_a})$ of v_i can be covered by $D_{x''}$. As formula $(*)$ shows, this part of the weight corresponds to β. Furthermore the split itself does not change UB_{WC}, as the sum of the maximum weights of the derived sets $D_{x'}$ and $D_{x''}$ is the same as the maximum weight of D_x:

$$\hat{w}_{x'}(D_{x'}) + \hat{w}_{x''}(D_{x''}) = w_x(v_{x_a}) + w_x(v_{x_1}) - w_x(v_{x_a}) = w_x(v_{x_1}) = \hat{w}_x(D_x).$$

This whole derivation leads to a rather compact realization within the code of section 2.1. The whole split only will be executed if the current ISs of Δ cannot fully cover the weight of v_i and a suitable set D_x was found. The latter is the case if $\beta > 0$. The split of D_x can be performed by calling the already introduced function *splitWLMC* with a weight threshold of $\hat{w}(D_x) - \beta$. D_x gets removed from Π, while its successors $D_{x'}$ and $D_{x''}$ are added to it. As $D_{x''}$ does not contain neighbors of v_i anymore, it can be added to Δ.

After that, in section 2.2 all members of Δ, if applicable including the just added $D_{x''}$, are used to cover the weight of v_i. The indices of the p ISs in Δ are tagged as j_1 to j_p. In a loop for every IS in Δ except the last one, v_i is added to the IS D_{j_s} with a weight of $w_{j_s}(v_i) = \hat{w}_{j_s}(D_{j_s})$. This ensures that as much weight of v_i as possible without changing UB_{WC} is covered by each D_{j_s}. For D_{j_p}, the last IS of Δ, the processing is a little different. v_i also gets added to it, but this time with an assigned weight which is the difference between $w(v_i)$ and the weight already covered by the other members of Δ. This difference is called the residual weight $w(v_i) - \sum_{s=1}^{p-1} w_{j_s}(v_i)$. With the weight remainder being handled by D_{j_p}, the whole weight of v_i is now covered by the weight cover Π. In the first $p-1$ members of

Δ, the maximum weight has not changed, so their modification does not influence UB_{WC}. Only the last set D_{j_p} could potentially induce a change of the upper bound by

$$max(w_{j_p}(v_i) - \hat{w}_{j_p}(D_{j_p}), 0) \mid \text{considering } D_{j_p} \text{ before } v_i \text{ is added.}$$

Should Δ be empty, section 2.3 simply adds v_i with its whole weight to a new IS $D_{|\Pi|+1}$. This new IS is then appended to the weight cover Π. The if condition marked with 2 in the listing 7 ensures that neither the modifications of Π conducted within branch 2.1 nor 2.2 raise UB_{WC} above t.

After the modifications to Π are finished, v_i is inserted into the set of non-branching vertices A. In the else-case 2.4 where condition 2 was not satisfied, Π remains unchanged and v_i is added to the branching vertices B. When all $v_i \in V$ were processed by the outer for-loop, A and B are complete and get returned.

Example The example graph seen in figure 3.5 will now also be used to demonstrate the partitioning procedure. As in the original paper [18], furthermore an ordering O yielding $v_6 < v_5 < v_4 < v_3 < v_2 < v_1$ and $t = 6$ are chosen. Figure 3.6 visualizes the state of the weight cover after each iteration of the main for-loop of the partitioning algorithm.

(1) Since the weight cover is initialized as empty, the first iteration where $v_i = v_1^3$ will practically skip the preparation phase 1 of the algorithm. Therefore the integration phase 2 will start with the following preconditions:

$$\Pi = \emptyset \qquad \Delta = \emptyset \qquad \beta = 0$$

$$\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j) + max\Big(w(v_1) - \Big(\sum_{D_k \in \Delta} \hat{w}_k(D_k) + \beta\Big), 0\Big) = 0 + max\big(3 - (0+0), 0\big) = 3 \leq t.$$

As the integration condition is fulfilled with $t = 6$, v_1 will be integrated into Π and added to A. The then-branches of the conditions 2.1 and 2.2 are not executed, unlike 2.3 which inserts v_1^3 into its own new IS D_1 with its full weight. This results in the weight cover shown in (1) of figure 3.6.

(2) The second iteration considers v_2^3. The preparation phase 1 cannot insert D_1 into Δ because the only member of D_1, v_1, is a neighbor of v_2. Furthermore branch 1.2 cannot determine a new β since no non-neighbor u is available. The state before the integration phase is

$$\Pi = \{\{v_1^3\}\} \qquad \Delta = \emptyset \qquad \beta = 0$$

$$\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j) + max\Big(w(v_2) - \Big(\sum_{D_k \in \Delta} \hat{w}_k(D_k) + \beta\Big), 0\Big) = 3 + max\big(3 - (0+0), 0\big) = 6 \leq t.$$

Algorithm 7: PartitionWC-MWC(G, t, O)

$\Pi := \emptyset$; // weight cover(sets of ISs)
$A := \emptyset, B := \emptyset$; // sets of vertices
for $i := |V|$ *to* 1 **do**
  $\beta := 0$;
  $\Delta := \emptyset$;
1 **for** $j := 1$ *to* $|\Pi|$ **do**
1.1 **if** $N(v_i) \cap D_j = \emptyset$ **then**
      $\Delta := \Delta \cup \{D_j\}$;
1.2 **else**
      $u := \arg\max_{z \in N(v_i)} w_j(z)$;
      if $\hat{w}_j(D_j) - w_j(u) > \beta$ **then**
        $x := j$;
        $\beta := \hat{w}_j(D_j) - w_j(u)$;
1.3 **if** $\left(\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j) + max\left(w(v_i) - \left(\sum_{D_k \in \Delta} \hat{w}_k(D_k) + \beta\right), 0\right)\right) \leq t$ **then**
      break;
2 **if** $\left(\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j) + max\left(w(v_i) - \left(\sum_{D_k \in \Delta} \hat{w}_k(D_k) + \beta\right), 0\right)\right) \leq t$ **then**
2.1 **if** $\left(\sum_{D_k \in \Delta} \hat{w}_k(D_k) < w(v_i)\right)$ and $(\beta > 0)$ **then**
      $(D_{x'}, D_{x''}) = splitWLMC(D_x, \hat{w}_x(D_x) - \beta)$;
      $\Pi := (\Pi \setminus \{(D_x, w_x)\}) \cup \{(D_{x'}, w_{x'}), (D_{x''}, w_{x''})\}$;
      $\Delta := \Delta \cup \{D_{x''}\}$;
2.2 **if** $\Delta \neq \emptyset$ **then**
      Let $\Delta = \{D_{j_1}, D_{j_2}, \ldots, D_{j_p}\}$;
      for $s := 1$ *to* $p - 1$ **do**
        $w_{j_s}(v_i) := \hat{w}_{j_s}(D_{j_s})$;
        $D_{j_s} := D_{j_s} \cup \{v_i^{w_{j_s}(v_i)}\}$;
      $w_{j_p}(v_i) := w(v_i) - \sum_{s=1}^{p-1} w_{j_s}(v_i)$;
      $D_{j_p} := D_{j_p} \cup \{v_i^{w_{j_p}(v_i)}\}$;
2.3 **else**
      $D_{|\Pi|+1} := \{v_i^{w(v_i)}\}$;
      $\Pi := \Pi \cup \{(D_{|\Pi|+1}, w_{|\Pi|+1})\}$;
    $A := A \cup \{v_i\}$;
2.4 **else**
    $B := B \cup \{v_i\}$;
return (A, B);

This makes it possible to add v_2 to Π and A as well. Like v_1, v_2 is added to its new IS D_2 which covers its full weight. Graph (2) in figure 3.6 is shows the new weight cover Π.

(3) The situation in the third iteration with $v_i = v_3^2$ is a little different. D_1 contains no neighbors of v_3 and is thus added to Δ. v_2, the only member of D_2, is adjacent to v_3. So β will remain unchanged by 1.2 again. We now have:

$$\Pi = \{\{v_1^3\}, \{v_2^3\}\} \qquad \Delta = \{\{D_1\}\} \qquad \beta = 0$$

$$\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j) + max\left(w(v_3) - \left(\sum_{D_k \in \Delta} \hat{w}_k(D_k) + \beta\right), 0\right) = 6 + max\big(2 - (3+0), 0\big) = 6 \leq t.$$

As Δ is not empty this time, the algorithm enters branch 2.2. Δ has one member, so $p = 1$ and the body of the for-loop within 2.2 is therefore never executed. The full weight of v_3 is covered by the residual weight function

$$w_{j_p}(v_3) = w_{j_1}(v_3) = w(v_3) - \sum_{s=1}^{p-1} w_{j_s}(v_3) = 2 - 0 = 2.$$

Then v_3^2 is inserted into D_1 and finally into A too. The resulting weight cover is depicted in part (3) of figure 3.6.

(4) Within the fourth iteration v_4^1 is processed. D_1 contains two vertices, v_1 and v_3. v_1^3 is the highest weight member of D_1 and not a neighbor of v_4. v_3^2 on the other hand is adjacent to v_4. Due to this, D_1 cannot be added to Δ. Section 1.2 of the algorithm will determine $u = v_3^2$ and

$$\beta = \hat{w}_1(D_1) - w_1(u) = w_1(v_1) - w_1(v_3) = 3 - 2 = 1.$$

Meanwhile D_2 can be added to Δ. The resulting state after the preparation phase is:

$$\Pi = \{\{v_1^3, v_3^2\}, \{v_2^3\}\} \qquad \Delta = \{\{D_2\}\} \qquad \beta = 1 \qquad x = 1$$

$$\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j) + max\left(w(v_4) - \left(\sum_{D_k \in \Delta} \hat{w}_k(D_k) + \beta\right), 0\right) = 6 + max\big(1 - (3+1), 0\big) = 6 \leq t.$$

During the integration phase, 2.1 is once again skipped because the full weight of v_4 can be covered by Δ. The latter is then realized in section 2.2, where v_4^1 is integrated into D_2. Π after this operation is shown in (4) of figure 3.6. Just as its predecessors, v_4 is added to A.

(5) Another interesting constellation is present in the fifth iteration where $v_i = v_5^4$. The only neighbor of v_5 is v_6, which has not been assigned to an IS yet. Therefore both D_1 and D_2 are adjoined to Δ, which leaves β at its default value:

$$\Pi = \{\{v_1^3, v_3^2\}, \{v_2^3, v_4^1\}\} \qquad \Delta = \{\{D_1\}, \{D_2\}\} \qquad \beta = 0$$

$$\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j) + max\Big(w(v_5) - \Big(\sum_{D_k \in \Delta} \hat{w}_k(D_k) + \beta\Big), 0\Big) = 6 + max\big(4 - (6+0), 0\big) = 6 \leq t.$$

This time the full vertex weight is greater than the maximum weights of the IS $D_j \in \Delta$:

$$w(v_5) = 4 > \hat{w}_1(D_1) = \hat{w}_2(D_2) = 3.$$

Adding v_5 solely to one existing IS D_j would increase its maximum weight. That in turn would increase UB_{WC} above the weight limit t. To avoid that, the weight of v_5^4 is distributed among D_1 and D_2. In section 2.2 of the algorithm the for-loop is executed once and assigns the current maximum weight of $D_{j_1} = D_1$ to $w_{j_1}(v_5) = w_1(v_5)$. So v_5^3 is added to D_1, which means a weight value of 1 is left uncovered. This uncovered residual weight is then consumed by

$$w_{j_p}(v_5) = w_{j_2}(v_5) = w_2(v_5) = w(v_5) - \sum_{s=1}^{p-1} w_{j_s}(v_5) = 4 - 3 = 1.$$

With v_5^1 added to D_2, v_5 is fully covered by the modified Π, which can be found in picture (5) of figure 3.6. Putting v_5 into the set of non-branching vertices A concludes the iteration.

(6) The sixth and final iteration handles v_6^2. During the preparation phase 1, no independent set is added to Δ. From D_1, both v_1 and v_4 are neighbors of v_6 and each one covers a weight of 3 according to w_1. That is simultaneously the highest weight covered by w_1, which makes v_1 and v_4 exchangeable candidates for u, yielding

$$\hat{w}_1(D_1) - w_1(u) = 3 - 3 = 0 \not> \beta = 0.$$

Given this, D_1 will not be the D_x later in the algorithm. Looking at D_2, it is evident that it has two members who are neighbors of v_6 as well, v_4 and v_5. Both have an equal weight of 1 according to w_2, making them both eligible for being the highest-weight neighbor u. Contrary to D_1 the highest weight member of D_2 is not a neighbor of v_6, so β can be calculated as

$$\beta = \hat{w}_2(D_2) - w_2(u) = 3 - 1 = 2.$$

At the end of the preparation phase, there is now a situation where solely a modification of the existing weight cover Π that goes beyond adding vertices can help to meet the weight limit t. The weight covering potential that this modification can deliver is encoded in β. The whole state before the integration phase 2 is as follows:

$$\Pi = \{\{v_1^3, v_3^2, v_5^3\}, \{v_2^3, v_4^1, v_5^1\}\} \qquad \Delta = \emptyset \qquad \beta = 2 \qquad x = 2$$

$$\sum_{j=1}^{|\Pi|} \hat{w}_j(D_j) + max\Big(w(v_6) - \Big(\sum_{D_k \in \Delta} \hat{w}_k(D_k) + \beta\Big), 0\Big) = 6 + max\big(2 - (0+2), 0\big) = 6 \leq t.$$

This time branch 2.1 of the integration phase will be visited. Its condition is satisfied since $\sum_{D_k \in \Delta} \hat{w}_k(D_k) = 0 < w(v_6) = 2$ and $\beta = 2 > 0$. Now the previously determined $D_x = D_2$ is split up into $D_{x'} = \{v_2^1, v_4^1, v_5^1\}$ and $D_{x''} = \{v_2^2\}$. The previous $D_x = D_2$ with its weight function is removed from Π, while $(D_{x'}, w_{x'})$ and $(D_{x''}, w_{x''})$ are added to it. Due to the latter ones now being the second and third ISs in Π, $D_{x'}$ will from now on be the new D_2, while $D_{x''}$ is referenced as D_3. Additionally, $D_{x''}$ respectively D_3 is added to Δ, it can be seen that this is valid because its only member v_2 is no neighbor of v_6. With the modified $\Delta = \{D_3\}$, section 2.2 is able to add v_6^2 to D_3 with its full weight. After inserting v_6 into A the final state after the final iteration is

$$\Pi = \{\{v_1^3, v_3^2, v_5^3\}, \{v_2^1, v_4^1, v_5^1\}, \{v_2^2, v_6^2\}\} \qquad A = \{v_1, v_2, v_3, v_4, v_5, v_6\}.$$

Π is also visualized as the last graph (6) within figure 3.6. With that, the upper bound based on the weight cover was able to prove that none of the vertices $v_1, \ldots, v_6$ is a branching vertex. This can save valuable runtime that would otherwise be wasted on exploring branches that will not lead to a new best clique.

3.1.3 TSM-MWC

Another exact state of the art BnB algorithm is TSM-MWC [8] from 2018 by Jiang, C.M. Li, Liu and Manyà. Its novelty is the **t**wo-**s**tage **M**axSAT reasoning approach which also is the origin of the algorithm's name.

Just as WC-MWC, TSM-MWC is very comparable to WLMC in many parts but again differs in the process of determining the branches. Therefore the main procedure *TSM-MWC* (algorithm 29) and the corresponding clique search procedure *SearchMaxWCliqueTSM-MWC* (algorithm 30)) are likewise only listed in the appendix 7.3 and not explained a second time.

The function *GetBranchesTSM-MWC* as seen in algorithm 8 merely calls two other sub-functions that perform the two actual MaxSAT reasoning stages. An initial set of branching vertices B, a set of ISs Π and an initial upper bound ub are calculated by *BinaryMaxSATTSM-MWC*. Unless this first stage did not already reveal that there are no interesting branching vertices and a further search is futile, the second stage realized by *OrderedMaxSATTSM-MWC* is applied. The second MaxSAT reasoning stage produces a reduced set of branching vertices that is then returned to the superordinate clique search function.

Binary MaxSAT reasoning

The first stage of finding the branching vertices uses a strategy which resembles some of the approaches that were already used in WLMC and WC-MWC. The

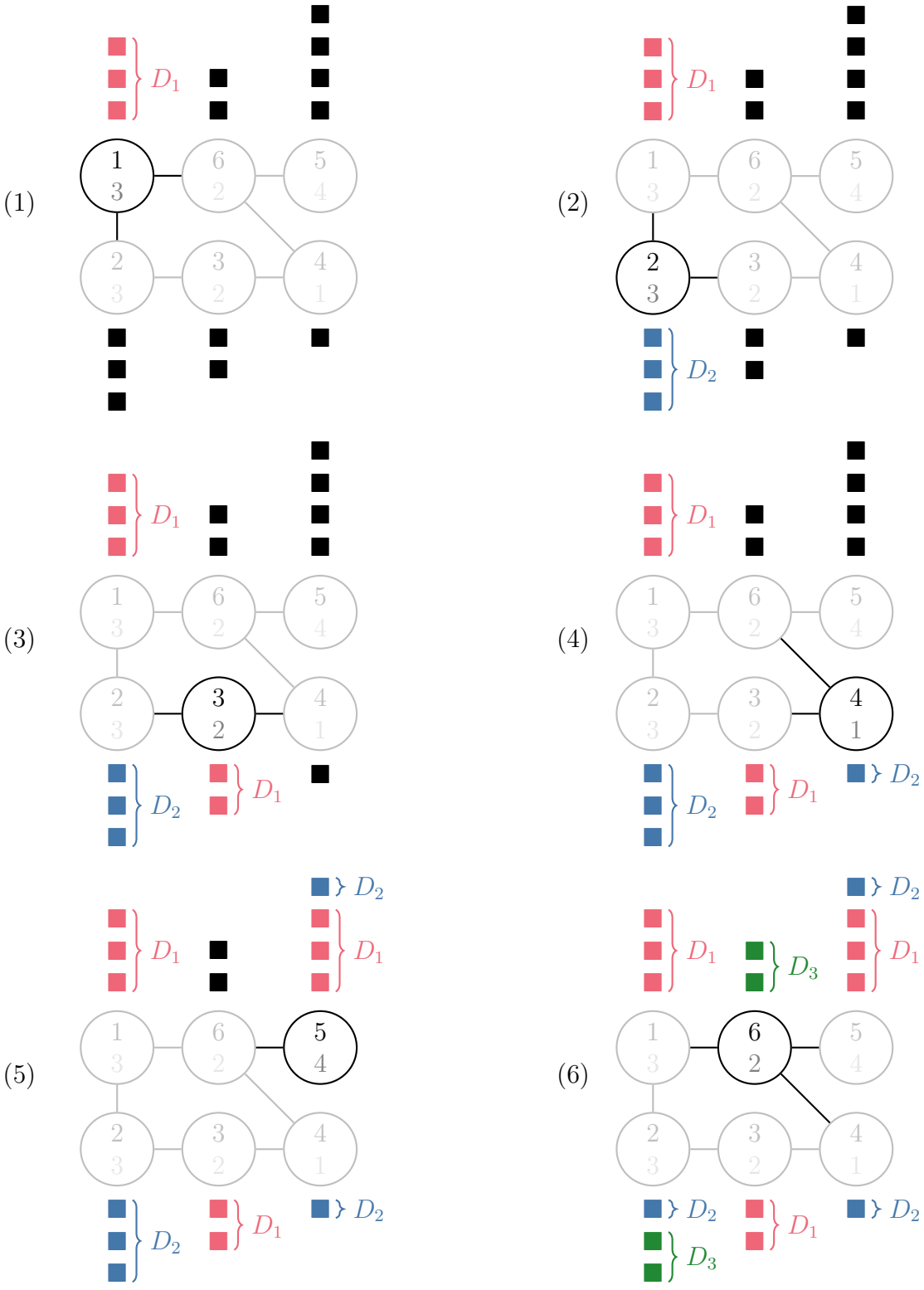

Figure 3.6: Partitioning on the example graph G

Algorithm 8: GetBranchesTSM-MWC(G, t, O)

$(B, \Pi, ub) := BinaryMaxSATTSM\text{-}MWC(G, t, o)$;
if $B \neq \emptyset$ **then**
  $B := OrderedMaxSATTSM\text{-}MWC(G, t, O, B, \Pi, ub)$;
return B;

binary MaxSAT reasoning procedure produces a set of ISs Π. This will contain ISs whose vertices cannot form a clique greater than the given weight limit t. The upper bound of weight of a clique formed from Π is returned as ub. As seen from previous algorithms, the upper bound is defined by the sum of the maximum weights of the ISs: $ub = \sum_{D \in \Pi} \hat{w}(D)$. The vertices which could not be integrated into Π without exceeding the weight limit will go to the branching vertices B. At the beginning, Π and B are empty, consequently $ub = 0$.

The main for-loop of the procedure iterates through the vertices v_i from the greatest one according to O to the smallest one. In the beginning of each iteration the current state of Π is cached as Π'. The original weight of v_i is stored as δ. Now every vertex that is not a neighbor of v_i is removed from Π. The processing that follows depends on the nature of Π after the non-neighbor removal.

If the removal produces no empty ISs, v_i cannot be integrated into any existing IS as they all have at least one neighbor of v_i. In this case branch 1 of the algorithm is triggered. First all removed vertices are restored to their previous ISs. Due to the initial condition of 1, v_i has to be added into a new IS if it should be integrated into Π. This would increase the upper weight bound ub by the weight of v_i, which is equivalent to δ. Adding v_i to Π is only allowed if the increased weight is smaller or equal than t, which is checked in condition 1.1. When the former condition is fulfilled, v_i is added to a new IS of Π with its full weight and the weight bound ub is updated accordingly. Otherwise the else-branch 1.2 gets executed. There v_i is added to the branching vertices B as it cannot be added to a new set in Π without violating t.

In the case that removing vertices which are non-adjacent to v_i results in some ISs $S_1, S_2, \ldots, S_k$ being empty, branch 2 will be invoked. Firstly, again all vertices are restored to the ISs they were just removed from. After that the inner for-loop iterates over the ISs $S_1, S_2, \ldots, S_k$ to distribute the weight of v_i among them. In the default-case 2.2, v_i is added to each set S_j with the current maximum weight $\hat{w}(S_j)$ of that set. δ, which is used to keep track of the weight that has not been distributed yet, gets decreased by the newly assigned weight $w_j(v_i) = \hat{w}(S_j)$. Path 2.1 represents the early termination of the inner for-loop. It is entered when adding v_i with its full remaining weight to S_j will not violate the weight limit t. Besides adding the remainder of v_i to S_j the upper bound ub and the remaining weight are updated accordingly.

After the inner loop, condition 2.3 checks whether the full weight of v_i could not be covered. If so, it is assumed that v_i can potentially lead the search to a new best clique and it is this added to the branching vertices B. Since integrating v_i into Π was not concluded successfully, the latter is in an invalid state. Therefore, Π is reset to Π', its state at the beginning of the current iteration of the outer loop.

Example To help the understanding of the binary MaxSAT reasoning stage, it will be demonstrated on the example graph in figure 3.7 which was initially presented in [8]. It is assumed that the ordering 0 yields $v_6 < v_5 < \ldots < v_1$ and that $t = 6$.

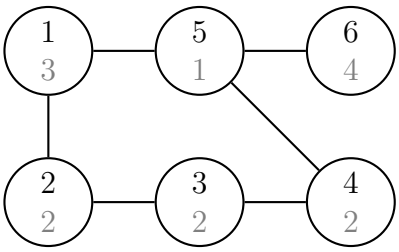

Figure 3.7: Example graph G for binary MaxSAT reasoning

(1) At first the outer loop will process v_1^3, so we initialize $\delta = 3$. Because Π is empty, branch 1 will inevitably be triggered. Since $ub + \delta = 0 + 3 < t = 6$, furthermore the condition 1.1 is satisfied. Therefore v_1^3 is added to its own IS in Π. After updating the upper bound, the resulting state at the end of the first main iteration is

$$\Pi = \{\{v_1^3\}\} \qquad ub = 3 \qquad B = \emptyset.$$

(2) The next considered vertex is v_2^2. v_1^3 is the only vertex present in Π at the moment and it is a neighbor of v_2. Given this, there are no non-neighbors of v_2 that could be removed to produce an empty IS, so the algorithm visits branch 1. With $\delta = 2 \Rightarrow ub + \delta = 3 + 2 = 5$, so again the if branch 1.1 is executed, which results in

$$\Pi = \{\{v_1^3\}, \{v_2^2\}\} \qquad ub = 5 \qquad B = \emptyset.$$

(3) When looking at v_3^2, removing the vertices not adjacent to it (i.e. v_1^3) would change Π to $\{\emptyset, \{v_2^2\}\}$. So as the operation produces an empty set D_1, the else branch 2 is chosen. Π is brought back to its state before removing v_3. With $S_1 = D_1$, the inner for loop is executed once. Due to

$$ub + max(\delta - \hat{w}(S_1), 0) = 5 + max(2 - 3, 0) = 5 < t = 6$$

the procedure continues with branch 2.2. The latter one fully adds v_3^2 to S_1. The upper bound remains unchanged because the maximum weight of $S_1 = D_1$ is not

Algorithm 9: BinaryMaxSATTSM-MWC(G, t, O)

```
ub := 0;
B := ∅;
Π := ∅;
Let V = {v_1, ..., v_n} | v_1 < ... < v_n w.r.t. O;
for i := n downto 1 do
    Π' := Π;
    δ := w(v_i);
    remove vertices non-adjacent to v_i from their ISs;
1   if ∄S ∈ Π | S = ∅ then
        restore all removed vertices into their ISs;
1.1     if ub + δ ≤ t then
            D = {v_i^δ};
            Π := Π ∪ {D};
            ub := ub + δ;
1.2     else
            B := B ∪ {v_i};
2   else
        Let S_1, S_2, ..., S_k be the empty ISs;
        restore all removed vertices into their ISs;
        for j := 1 to k do
2.1         if ub + max(δ − ŵ(S_j), 0) ≤ t then
                S_j := S_j ∪ {v_i^δ};
                ub := ub + max(δ − ŵ(S_j), 0);
                δ := 0;
                break;
2.2         else
                S_j := S_j ∪ {v_i^ŵ(S_j)};
                δ := δ − ŵ(S_j);
2.3     if δ > 0 then
            Π := Π';
            B := B ∪ {v_i};
return (B, Π, ub);
```

surpassed the weight of v_3. With no weight left to be distributed at condition 2.3, this main iteration is concluded with

$$\Pi = \{\{v_1^3, v_3^2\}, \{v_2^2\}\} \qquad ub = 5 \qquad B = \emptyset.$$

(4) A similar situation is present when processing v_4^2. Removing its non-neighbor v_2^2 results in $\Pi = \{\{v_1^3, v_3^2\}, \emptyset\}$. Again section 2 is visited, this time with $S_1 = D_2$, into which v_4^2 gets fully integrated. With this, the state after the fourth iteration will be

$$\Pi = \{\{v_1^3, v_3^2\}, \{v_2^2, v_4^2\}\} \qquad ub = 5 \qquad B = \emptyset.$$

(5) Next up is v_5^1. It is adjacent to members of both ISs of Π. Therefore removing its non-neighbors from Π creates no empty ISs, so this time branch 1 of the algorithm 9 is visited. Increasing the upper bound ub by $\delta = 1$ still barely meets the weight limit $t = 6$. So v_5^1 can be added to its own IS D_3 in section 1.1. Now we have

$$\Pi = \{\{v_1^3, v_3^2\}, \{v_2^2, v_4^2\}, \{v_5^1\}\} \qquad ub = 6 \qquad B = \emptyset.$$

(6) At the start of the last iteration which handles v_6^4, the upper bound ub already reached the weight limit t. This means that v_6 either has to distributed among existing ISs without increasing their maximum weight, or it has to become a branching vertex. δ is initialized as 4. The only neighbor of v_6 is $v_5^1 \in D_3$. Neither D_1 nor D_2 contain neighbors of v_6, so the usual vertex removal operation produces $\Pi = \{\emptyset, \emptyset, \{v_5^1\}\}$ this time. Given the empty sets, the algorithm chooses branch 2, where $S_1 = D_1$ and $S_2 = D_2$. In the first iteration of the inner for-loop, the condition 2.1 evaluates to

$$ub + max(\delta - \hat{w}(S_1), 0) = 6 + max(4 - 3, 0) = 7 \not\leq t = 6,$$

so the else branch 2.2 is selected. There v_6 is added to S_1 with only $\hat{w}(S_1) = 2$ of its weight. This still leaves $\delta = 4 - 2 = 2$ of the weight to be distributed. The second iteration of the inner loop has

$$ub + max(\delta - \hat{w}(S_2), 0) = 6 + max(2 - 3, 0) = 2 < t = 6$$

satisfied, so the if-branch 2.1 is carried out. This adds v_6 with its remaining weight to S_2. ub stays the same because $max(\delta - \hat{w}(S_2), 0) = max(2 - 2, 0) = 0$. The algorithm finishes with the state

$$\Pi = \{\{v_1^3, v_3^2, v_6^2\}, \{v_2^2, v_4^2, v_6^2\}, \{v_5^1\}\} \qquad ub = 6 \qquad B = \emptyset.$$

The binary MaxSAT reasoning algorithm was able to prove that no branching vertices worth to further explore are present in the graph G. Figure 3.8 shows the distribution of the vertex weights in Π in the style of the visualizations found in the previous weight cover chapter 3.1.2.

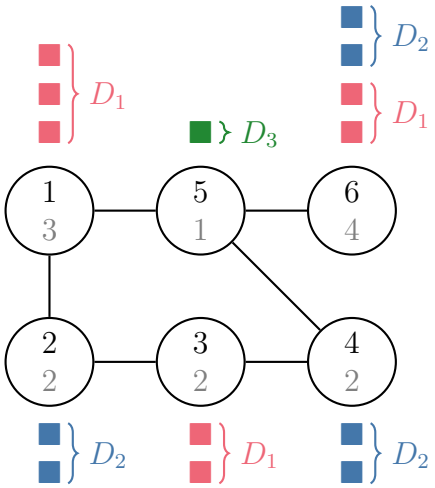

Figure 3.8: Result of the binary MaxSAT reasoning example

Ordered MaxSAT reasoning

The second stage of determining the branching vertices, the ordered MaxSAT reasoning, takes the initial branching vertices B determined by the binary MaxSAT reasoning as an input. Additionally together the corresponding set of ISs Π and upper bound for a clique weight ub are passed, besides the usual G, t and O. The first stage was not able to fully integrate the branching vertices B into any set of Π. However there might still a possibility to integrate parts of the weight of certain branching vertices into Π, so that the upper bound ub together with the parts of the branching vertices' weight that could not be integrated does not exceed t. That is exactly what the algorithm $OrderedMaxSATTSM\text{-}MWC$ shown in the algorithm listing 10 tries to do.

In its main loop the procedure iterates over the branching vertices from the largest to the smallest according to the ordering O. Just as in $BinaryMaxSATTSM\text{-}MWC$, at the beginning of each iteration a backup Π' of Π is stored and the remaining weight δ of the branching vertex b_i, that was not yet integrated into Π, is initialized with the full weight of b_i. After that, the algorithm tries to integrate parts of δ (or b_i respectively) into Π by looking at different subsets of conflicting ISs in $\Pi \cup \{\{b_i\}\}$ with increasing size one after another, hence the "ordered" in the name. First will be the subsets with 2 conflicting ISs, then those with 3, and finally those with more than 3. For each of those cases, the algorithm will iterate over the sets of conflicting ISs in an inner loop. Therein it performs an adequate integration and splitting if possible and needed. At the end of each inner iteration, δ is updated and potentially the inner loop is terminated early when the limit t is already satisfied.

The subsets with two conflicting ISs are processed in section 1. Those conflicting subsets have the form $\{S_j, \{b_i\}\}$, where S_j is an independent set within Π that

contains no neighbor of b_i, so it cannot be in a clique with it. To resolve the conflict situation here, b_i can simply be added to each of such sets S_j with the current maximum weight of that set assigned to it, as realized in line 1.1. In the following line 1.2, the just integrated weight is subtracted from δ. An early termination of the integration process is not implemented for this first phase of conflict resolving. If the integration into those sets alone would cover all of δ, b_i would have already been fully inserted into Π by $BinaryMaxSATTSM\text{-}MWC$.

Section 2 takes care of the subsets of conflicting ISs with three members. To identify those subsets, first the ISs $\{U_1, U_2, \ldots, U_r\}$ containing exactly one neighbor u of b_i each are collected. Then for every U_j the algorithm tries to find another set D_q which contains no members that are neighbors of both b_i and u. Now no clique containing one member of D_q, $\{b_i\}$ and U_j each could be formed. Out of U_j only u could be in such a clique, as it is the only neighbor of b_i there. Due to the condition for D_q, it has no vertices that can form a clique with b_i and u. Therefore the set $\{D_q, \{b_i\}, U_j\}$ is conflicting. To resolve the conflict, section 2.1 uses the strategy seen within the second scenario of the $UP\&SplitWLMC$ function. The details of this were already explained in the subsection Unit IS propagation of chapter 3.1.1. One difference is that there are no markings and no set Δ to deal with. Also the splitting parameter is denoted as β since δ is already in use, and this β is used to update δ instead of ub directly in section 2.2. Should the changes up until now be enough to make $ub + \delta$ fit within t, no further conflicts need to be searched and resolved, and the inner loop of section 2 is terminated.

If previous steps were not sufficient, section 3 inserts b_i into its own new IS inside Π. Then it performs nearly the same processing that was already presented in the first scenario of the $UP\&SplitWLMC$ function, with adaptions as in section 2. Once again the loop is terminated early should $ub + \delta$ not exceed t anymore.

At the final section 4 of each iteration, it is checked whether a large enough fraction of the weight of b_i was integrated into Π, so that the weight limit t is satisfied. If that is the case, $OrderedMaxSATTSM\text{-}MWC$ has proven that b_i does not need to be explored as a branching vertex and thus removes it from B. The not integrated remainder weight δ is added to the upper bound ub. Should the weight however not be satisfied, the partly integration of b_i into B did not serve a purpose and is therefore reverted by resetting Π to its backup Π'. For an example of how the the ordered MaxSAT reasoning procedure can reduce the number of branches, the interested reader is referred to the original paper [8].

3.1.4 MWCRedu

MWCRedu [17] is another current exact algorithm proposed by Erhardt, Hanauer, Kriege, Schulz and Strash in 2023 that employs various graph reduction strategies, including new ones.

Algorithm 10: OrderedMaxSATTSM-MWC(G, t, O, B, Π, ub)

```
    Let B = {b_1, ..., b_|B| | b_1 < ... b_|B|} w.r.t. O;
    for i := |B| downto 1 do
        Π' := Π;
        δ := w(b_i);
1       Let S_1, S_2, ..., S_k be the ISs containing no neighbor of b_i;
        for j := 1 to k do
1.1         S_j := S_j ∪ {b_i^{ŵ(S_j)}};
1.2         δ := δ − ŵ(S_j);
2       Let U_1, U_2, ..., U_r be the ISs containing exactly one neighbor of b_i;
        for j := 1 to r do
            Let N(b_i) ∩ U_j = {u};
            if ∃D_q ∈ Π | D_q ∩ N(b_i) ∩ N(u) = ∅ then
2.1             β := min(δ, ŵ(U_j), ŵ(D_q));
                (U'_j, U''_j) := splitWLMC(U_j, β);
                (D'_q, D''_q) := splitWLMC(D_q, β);
                Π := (Π \ {U_j, D_q}) ∪ {U''_j, D''_q};
2.2             δ := δ − β;
                if ub + δ ≤ t then
                    break;
3       if ub + δ > t then
            Π := Π ∪ {{b_i^δ}};
            while ∃ unit IS {v} ∈ Π do
                remove vertices non-adjacent to v from their ISs;
                if ∃S_0 ∈ Π | S_0 = ∅ then
                    Let S_1, ..., S_p be the ISs responsible of removing all vertices
                     from S_0;
                    restore all removed vertices into their ISs;
3.1                 β := min(ŵ(S_0), ..., ŵ(S_p));
                    for S_j ∈ {S_0, S_1, ..., S_p} do
                        (S'_j, S''_j) := splitWLMC(S_j, β);
                    Π := (Π \ {S_0, ..., S_p}) ∪ {S''_0, ..., S''_p};
3.2                 δ := δ − β;
                    if ub + δ ≤ t then
                        break;
4       if ub + δ ≤ t then
            B := B \ {b_i};
            ub := ub + δ;
        else
            Π := Π'
    return B;
```

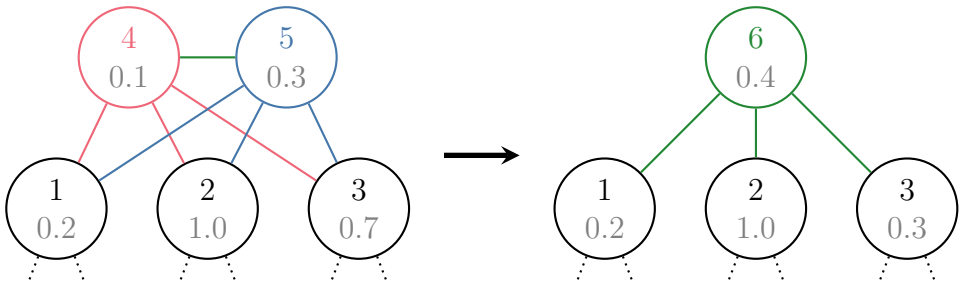

Figure 3.9: Example of a twin reduction

Reduction rules

The main means used by MWCRedu to solve the MWC problem faster are reductions of the input graph. With a greatly simplified graph, the process of finding a solution within it can be significantly accelerated. The following paragraphs introduce the rules that are used to reduce the graph.

The first reduction rule is the **Neighborhood Weight Reduction**. It removes vertices where the weight of their inclusive neighborhood is smaller than the weight of the best clique $\hat{C}$ that has been currently found. Even if such a vertex would form a clique with all its neighbors, the resulting clique will not improve the solution, thus the vertex can be omitted. This rule is equivalent to a reduction based on the upper bound UB_0 described in the FastWClq chapter 3.2.1.

The **Largest-Weight Neighbor Reduction** rule provides a tighter upper bound which is also dependent on the best currently found clique and furthermore takes the highest weight neighbor of a vertex into account. Again this reduction is already familiar from FastWClq, specifically its upper bound UB_1.

If two vertices have the same inclusive neighborhood, the **Twin Reduction** can be applied. Given that two vertices are adjacent and also share all other neighbors, they can be merged into one vertex whose weight is the sum of the two original vertices. An example is shown in figure 3.9, where $v_4^{0.1}$ and $v_5^{0.3}$ are twins that get merged into $v_6^{0.4}$.

Vertex u dominates a non-neighbor-vertex v if it has a larger or equal weight and all neighbors of v are also neighbors of u. According to the first **Domination Reduction** rule, a dominated vertex v can be removed from the graph. Any subset of v's neighbors that form a clique with it would also form a clique with u that has an equal or greater weight. In the example graph in figure 3.10, both v_4 and v_5 are connected to v_1 and v_2, but only v_5 is also connected to v_3. Furthermore, the weight of $v_5^{0.3}$ is higher than the weight of $v_4^{0.1}$. Therefore v_5 dominates v_4, so v_4 can be removed.

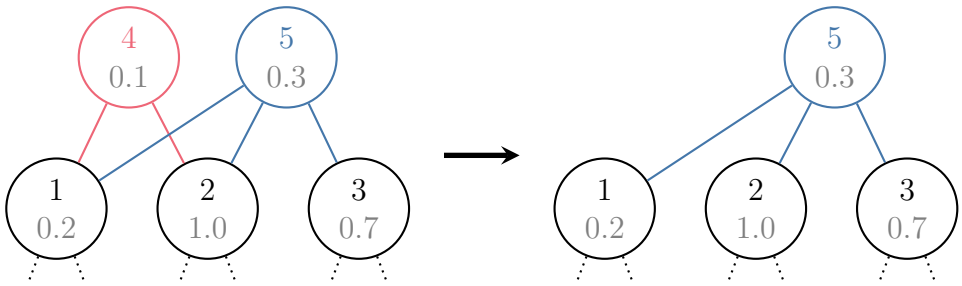

Figure 3.10: Example of a domination reduction of non-neighbor vertices

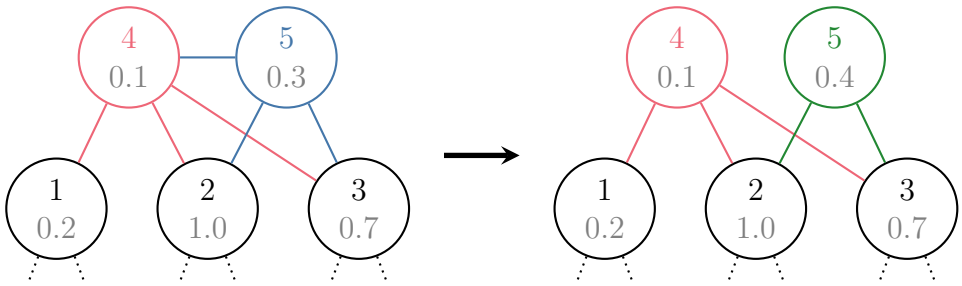

Figure 3.11: Example of a domination reduction of neighbor vertices

The second domination rule applies to two neighbor vertices u and v if all neighbors of v are also neighbors of u. Unlike in the first domination rule, it is not possible to remove v in such cases since it may be in a relevant clique together with u. However, the edge between u and v can be removed, in which case the weight of u has to be added to the weight of v, so that $w'(v) = w(v) + w(u)$. Since the precondition ensures that u is contained in any maximal clique including v, merging the weight of u into v and removing the edge in between them will produce maximal cliques of the same weight as if the edge was still there. The initial example graph in figure 3.11 shows such a situation where $v_4^{0.1}$ can be seen as u and $v_5^{0.3}$ as v. In the reduced graph, the weight update $w(v_5^{0.4}) = w(v_5^{0.3}) + w(v_4^{0.1})$ has been performed and there is no edge between $v_4^{0.1}$ and $v_5^{0.4}$ anymore.

The **Edge Bounding Reduction** rule is based on the largest weight neighbor reduction. In situations where a clique with v and a neighbor u cannot form a new best clique, it might still be possible to find a better clique that includes v but not u. Formally speaking, this is the case if the two conditions

$$w(N[v] \cap N[u]) \leq w(\hat{C})$$

$$w(N[v]) - w(u) > w(\hat{C})$$

are satisfied. Because of the second condition v should not be removed yet, since it can be part of a new best clique. However at least the edge between v and u can be removed as the maximum weight clique cannot contain both vertices at the same

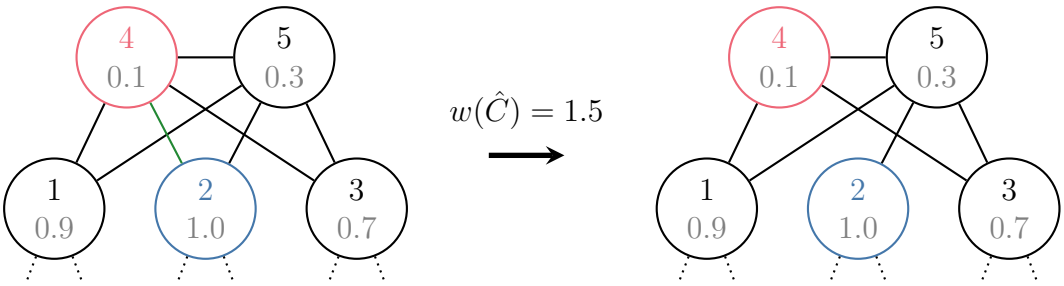

Figure 3.12: Example of an edge bounding reduction

time. An example of this is featured in figure 3.12. Vertex v_4 has its neighbor $v_2^{1.0}$. The upper bound for cliques with both v_4 and v_2 is determined by the sum of the weights of v_4, v_4 and v_5, which evaluates to 1.4. In the example the current best clique $\hat{C}$ has a weight of 1.5, so the first condition holds as

$$w(N[v_4] \cap N[v_2]) = w(\{v_2^{1.0}, v_4^{0.1}, v_5^{0.3}\}) = 1.4 \leq 1.5 = w(\hat{C}).$$

This alone is enough to justify removing the edge $\{v_4, v_2\}$. To verify that v_4 could not be removed altogether for sure, the second condition can be reviewed. This shows that v_4 might form a clique without v_2 which is better than $\hat{C}$, with an upper bound of 2.0:

$$w(N[v_4]) - w(v_2) = w(\{v_1^{0.9}, v_2^{1.0}, v_3^{0.7}, v_4^{0.1}, v_5^{0.3}\}) - w(v_2^{1.0}) = 2.0 > 1.5 = w(\hat{C}).$$

A vertex that forms a clique with its inclusive neighborhood is called simplicial. According to the **Simplicial Vertex Removal Reduction**, such a vertex can be removed in any case. If the clique that it forms together with its neighbors has a larger weight than the best clique found so far, it can should be set as the new candidate for the maximum weight clique. As the vertex can not be included in another larger clique, it does not need to be processed in the further steps. In figure 3.13, v_4 is a simplicial vertex as all its neighbors v_1, v_2 and v_3 are connected to each other and thus form a clique C with v_4. If this clique is better than the best currently found clique, i.e., $w(C) > w(\hat{C})$, the best clique has to be updated accordingly: $\hat{C} = C$.

Algorithm

The main procedure *MWCRedu* is pretty straightforward, as algorithm 11 demonstrates. First, the best clique $\hat{C}$ is initialized by the initial clique generation strategy used in the function *InitializeWLMC*. That clique and the graph G are passed to *ReduceMWCRedu*, which applies the reduction rules on the graph and potentially also improves $\hat{C}$. Finally, the reduced graph is processed by *TSM-MWC* and the result is returned.

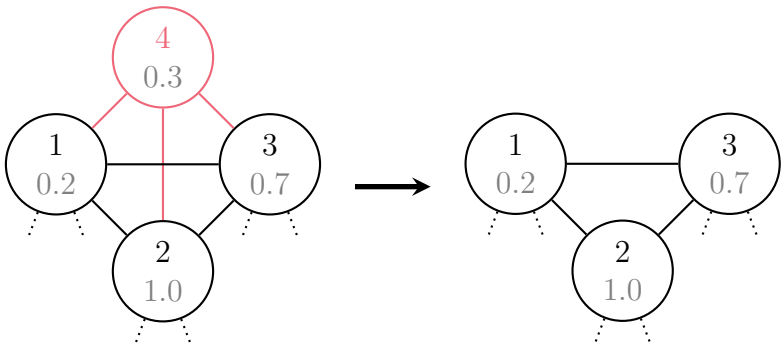

Figure 3.13: Example of a simplicial vertex reduction

Algorithm 11: MWCRedu

$\hat{C} := InitializeWLMC(G, 0)$;
$ReduceMWCRedu(G, \hat{C}, true)$;
return *TSM-MWC(G, $\hat{C}$)*;

The application of the reduction rules is conducted in $ReduceMWCRedu$ (algorithm 12). The parameters it takes are the graph G that should be reduced, the current best clique $\hat{C}$ which is on the one hand needed for certain reduction rules and on the other hand potentially improved, and a boolean parameter named lim. If the latter is set to true, not all vertices will be processed by the reduction rules at once. At the beginning only the vertices with a degree of up to 10% of the maximum degree will be processed, and the limit on the vertex degree will be iteratively increased until all vertices get processed. If lim is set to false, all vertices will be eligible for reduction at once. D_i is the set of vertices on which the reduction rule r_i will be applied. It is initialized to either all vertices or only those who meet the $vertexDegreeLimit$, depending on the value of lim. The reason for only reducing the vertices with a lower degree in the first steps is that they are more efficient to process, since fewer neighbors have to be checked. Furthermore they are less likely to be in a clique of large weight, although the number of vertices in a clique is not necessarily an indicator for its weight.

Now the main reduction loop starts. The main loop consists of two sections. Section 1 applies each reduction r_i on its eligible vertices D_i, unless the reduction rule is paused or D_i is empty. If a reduction rule cannot remove a vertex, this vertex is removed from D_i. By default, no reduction rule is paused. If a rule fails to remove 1% or more of the vertices (or edges) per second, it will be paused. After the application of each reduction rule, MWCRedu tries to improve its current best clique $\hat{C}$ by sampling cliques according to the strategy of FastWClq [7]. FastWClq is a heuristic algorithm that will be introduced in chapter 3.2.1. Condition 1.1 checks whether a paused reduction rule r_i can be unpaused again. The latter is the case

when other reduction rules have managed to remove at least 1% of the vertices or edges since r_i was paused. Section 2 of the main loop is only entered if $lim = true$ and all reductions are currently paused. If the condition is true, the reduction rules have been exhaustively applied on the current limited set of vertices. Therefore the $vertexDegreeLimit$ is increased so that the rules can be executed on more vertices. The sets of eligible vertices D_i are updated to also contain the new vertices that are not blocked by the $vertexDegreeLimit$ anymore. When all reductions have been paused, and at the same time the vertex degree is not limited anymore, the main loop terminates. In this situation there are no more vertices that could be investigated which were previously blocked by the $vertexDegreeLimit$, and no reduction rule is able to perform sufficiently fast reductions. The reduced version of the graph G and the updated best clique $\hat{C}$ are then returned.

Algorithm 12: ReduceMWCRedu(G, $\hat{C}$, lim)

```
if lim then
    vertexDegreeLimit := 0.1;
foreach r_i do
    initialize D_i;
repeat
1   foreach reduction rule r_i do
        if (r_i not paused) ∧ (D_i ≠ ∅) then
            apply r_i on every v ∈ D_i;
            update D_i;
            if reduction rate not achieved then
                pause r_i;
            update Ĉ via local search;
1.1     else if (r_i paused) ∧ (G reduced enough) then
            unpause r_i;
2   if lim ∧ (all reductions paused) then
        vertexDegreeLimit := vertexDegreeLimit + 0.1;
        foreach r_i do
            update D_i;
            unpause r_i;
until (all reductions paused) ∧ (vertexDegreeLimit > 1.0);
return (G, Ĉ);
```

3.2 Heuristic algorithms

3.2.1 FastWClq

FastWClq [7] is an heuristic algorithm proposed by Shaowei Cai and Jinkun Lin in 2016. An updated version of the algorithm was presented in 2022 [22], but the following chapter will focus on the original version. It tries to find a maximum clique in multiple iterations and stops either when an optimal solution is found or a predefined cutoff time is reached. The decision which vertex to add to a clique is made based on a heuristic that randomly samples a number of vertices from a candidate set and chooses the one which will provide the greatest weight increase together with its neighbors eligible for the clique. Over time, the number of randomly sampled vertices is increased if necessary. With each iteration the graph also gets reduced.

Main procedure

The main procedure of FastWClq is illustrated in algorithm 13. It maintains three important variables. One is a $StartSet$ which contains all vertices that have not yet been used to as a starting point for the clique construction. It is initialized to include all vertices in the graph. The currently best found clique $\hat{C}$ is initialized to an empty set. Last but not least, the strategy parameter k controls how many vertices get sampled in the heuristic process of estimating which vertices are the best ones to extend the cliques. The authors of [7] recommend using $k_0 = 4$ as an initial value for k. Everything else in the main procedure is executed within a loop that stops when the set time limit is reached. For a better overview, it is divided into three sections 1, 2, and 3, labeled at the left of the algorithm listing.

Section 1 controls the greediness of the sampling heuristic. If the $StartSet$ is empty, every vertex has been explored with the current strategy parameter k for the clique construction. In this case k is adapted by the $AdjustBMSNumberFastWClq$ function to make the exploration more greedy. The $StartSet$ is reset to all vertices of G so that the exploration can start over again with the new strategy parameter.

In section 2 the construction of a clique takes place. First a random vertex u is taken from the $StartSet$ and used as a starting point to build a clique C. The set of vertices that can be added to C, so that it remains a clique, is maintained as $Candidates$. Initially, all neighbors of u are set as potential $Candidates$. After that the clique is grown in the inner loop, until no more $Candidates$ are left. With the help of the $SelectVertexFastWClq$ function, the best of k randomly sampled vertices is chosen from the candidate set and moved into the clique. The $Candidates$ have to be updated so that they only consist of vertices that are adjacent to every member of the clique C. If it becomes clear during the construction of C that it

cannot become better than $\hat{C}$, the process of construction is aborted. The abortion criterion is that the weights of the current clique C, the selected best vertex v and its so called effective neighborhood do not surpass the weight of the best clique $\hat{C}$. More precisely, the effective neighborhood $N(v) \cap Candidates$ of a vertex v consists of all vertices that could possibly be added to the clique given that v gets added to it. Even in the best case only all the vertices of the effective neighborhood and no others could be added to the clique. Therefore the effective neighborhood is useful for obtaining an upper bound for the weight that the clique C can have when it is finished.

Finally section 3 deals with updating the results and the graph reduction. When the construction is finished and the constructed clique C is better than the best clique found so far, $\hat{C}$ is updated. The graph is then reduced so that vertices that can certainly not be included in better cliques than $\hat{C}$ are removed. This yields that if the graph is empty after the reduction, $\hat{C}$ can be returned before the time limit, since no better cliques can be found. In such a case FastWClq has proven that its solution is optimal and exits the main loop without reaching the time limit. As FastWClq is sure that it has delivered an exact solution in those cases, but on the other hand this condition is not triggered on every graph with every time limit, it was also described as a semi-exact algorithm in a later publication [22]. As the graph and especially its vertices have changed after the reduction, the $StartSet$ is reset to contain all vertices remaining in G again. Then the outer loop can start again and the search is continued on the potentially changed graph.

Strategy Parameter

The strategy parameter k is also called BMS number. BMS stands for "Best from Multiple Selection". This relates to the already mentioned process of choosing the next vertex: multiple eligible vertices are sampled, and the best one of them is selected. The function $AdjustBMSNumberFastWClq$ listed in algorithm 14 adapts the parameter by doubling it. That means that after this, twice as many vertices will be explored in each step of the clique construction. When k gets too big and surpasses its maximum allowed value k_{max}, it is reset to the initial value k_0. However with each reset k_0 is incremented by one, so the exploration still gets broader and greedier over the time. Cai et al. recommend setting the maximum value to 64.

Vertex selection

To decide which vertex from the $Candidates$ should be added, the weight gain of adding a vertex v to the clique is estimated. This weight gain is called the benefit of v. The minimum benefit $b_{min}(v)$, i.e., the worst case is present if only v can be

Algorithm 13: FastWClq

```
StartSet = V, Ĉ := ∅, k := k_0;
while within time limit do
1   if StartSet = ∅ then
        StartSet = V;
        k := AdjustBMSnumberFastWClq(k);
2   u := pop a random vertex from StartSet;
    C := {u};
    Candidates := N(u);
    while Candidates ≠ ∅ do
        v := SelectVertexFastWClq(Candidates, k);
        if w(C) + w(v) + w(N(v) ∩ Candidates) ≤ w(Ĉ) then
            Break;
        C := C ∪ {v};
        Candidates := Candidates \ {v};
        Candidates := Candidates ∩ N(v);
3   if w(C) > w(Ĉ) then
        Ĉ := C;
        G := ReduceGraphFastWClq(G, Ĉ);
        StartSet = V;
        if G = ∅ then
            return Ĉ;
return Ĉ;
```

Algorithm 14: AdjustBMSNumberFastWClq(k)

```
k := 2k;
if k > k_max then
    k := ++k_0;
return k;
```

added to the clique and no other vertices besides it:

$$b_{min}(v) = w(v).$$

The maximum achievable benefit $b_{max}(v)$ could be retrieved when v and all of its neighbors that are also included in *Candidates* can be added:

$$b_{max}(v) = w(v) + w(N(v) \cap Candidates).$$

To get an estimation of the benefit $b(v)$ that adding v will provide, the average of the minimum and maximum cases is used:

$$b(v) = \frac{b_{min} + b_{max}}{2}.$$

As shown in algorithm listing 15, k random vertices from the $Candidates$ are taken and their benefits are calculated. The vertex $\hat{v}$ with the highest benefit of those sampled vertices is selected to be added to the clique. If the cardinality of $Candidates$ is smaller than the k, the maximum benefit vertex can be determined exactly instead of using the sampling heuristic.

Algorithm 15: SelectVertexFastWClq($Candidates, k$)

```
if |Candidates| < k then
    return arg max_{v∈Candidates} b(v);
v̂ := a random vertex in Candidates;
for iteration := 1 to k - 1 do
    v := a random vertex in Candidates;
    if b(v) > b(v̂) then
        v̂ := v;
return v̂;
```

Graph reduction

To reduce the graph, upper bounds for the weights of cliques that a vertex v can be in are determined:

$$\forall C \ni v : UB(v) \geq w(C).$$

If such an upper bound is not higher than the weight of the best clique found so far, that means that none of the cliques that contain v can be the maximum weight clique. So if $UB(v) \leq w(\hat{C})$ is satisfied, v and all edges containing it can be removed from the graph, since this will have no influence on the maximum weight clique. The first simple upper bound is given by the weight of the inclusive neighborhood of v:

$$UB_0(v) = w(N[v]).$$

It is easy to see that this is a valid upper bound, it represents a situation where a vertex forms a clique with all its neighbors. By the definition of a clique, v can only be in cliques with vertices that it is connected to, so no clique could surpass this upper bound.

For a tighter upper bound, a vertex v and its neighbor with the highest weight u are considered. If neither a clique with v that contains u, nor one that does not contain

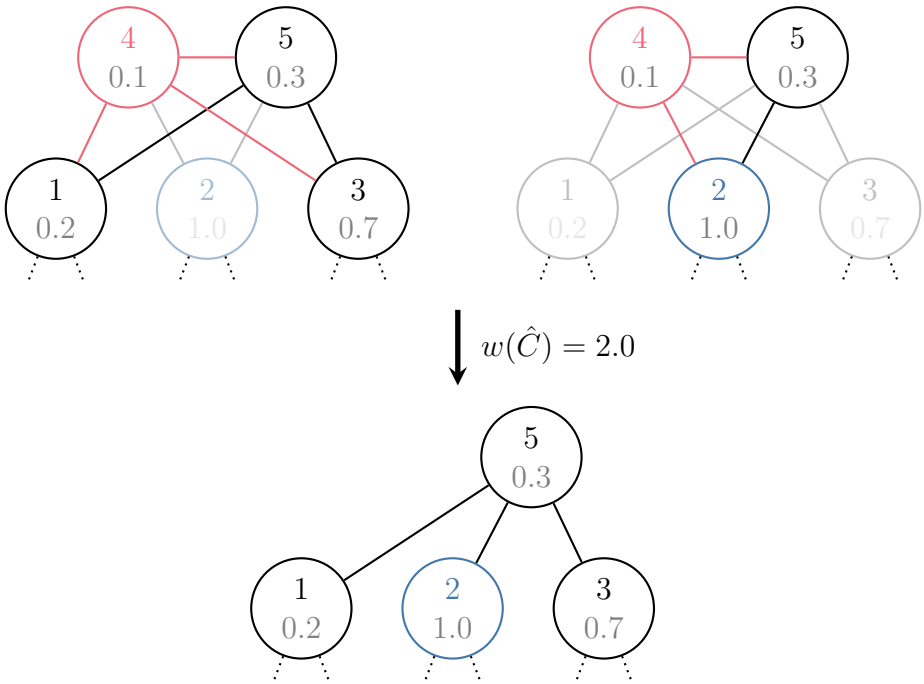

Figure 3.14: Example of a graph reduction using UB_1

u, can be better than $\hat{C}$, v can be removed. The upper bound for a clique with v not containing u is simply given as $w(N[v]) - w(u)$. Cliques that contain both v and u can only consist of neighbors of both vertices. So in this case the upper bound of the weight is dependent on the intersection of their inclusive neighborhoods $w(N[v] \cap N[u])$. Taken together, one can obtain another upper bound:

$$UB_1(v) = max\Big(w(N[v]) - w(u), w(N[v] \cap N[u])\Big).$$

An example is given in figure 3.14. The vertex v that gets evaluated is v_4, its neighbor with the highest weight is $v_2^{1.0}$. On the left of the figure the upper bound for a clique without v_2 is illustrated. Such a clique could contain v_1, v_3, v_4 and v_5 and have a weight ≤ 1.3. Note that v_1 and v_3 are not adjacent and thus could not actually be in the same clique. This is not an issue since we are only looking for an upper bound of the weight. On the right the case where v_2 is included in the clique is visualized. v_1 and v_3 are not included in $N[v_2]$ and neither in $w(N[v_4] \cap N[v_2])$. Hence the upper bound can only consist of $w(v_2^{1.0})$, $w(v_4^{0.1})$ and $w(v_5^{0.3})$ and evaluates to 1.4. In the example the current best clique has a weight of $w(\hat{C}) = 2.0$ which is larger than $UB_1(v_4) = max(1.3, 1.4)$. Therefore v_4 can be removed from the graph according to UB_1 without degrading the maximum weight clique. UB_0 would not have been able to prove that v_4 can be omitted, since $UB_0(v_4) = w(N[v_4]) = 2.3 \geq w(\hat{C}) = 2.0$.

Given the upper bounds, the reduction can be performed as seen in algorithm 16. The input data is a graph G that should be reduced and a clique C inside that graph.

C should be the best clique found so far to eliminate as many vertices as possible. At first, it is checked for all vertices whether their upper bounds are smaller than or equal to the weight of C. If that is the case, they are added to a queue of vertices that should be removed, the *RemovalQueue*. As $UB_0(v)$ is easier to compute, it is considered first. Only if UB_0 cannot show that v can be excluded, $UB_1(v)$ is evaluated. After this, all vertices in the *RemovalQueue* and their incident edges are removed from the graph. Since the upper bounds of the remaining neighbors $N_r(u)$ of the removed vertices u will change with the removal, those upper bounds are checked again. If this reveals that some vertices $v \in N_r(u)$ are now eligible for removal as well, they are added to the queue too. The reduction stops when the *RemovalQueue* is empty. The authors of [7] hint that the reduction process can be sped up for large graphs by only using UB_0 at first, and checking the more computationally expensive UB_1 after the graph is already partly reduced.

Algorithm 16: ReduceGraphFastWClq(G, C)

```
foreach v ∈ V do
    if (UB_0(v) ≤ w(C)) ∨ (UB_1(v) ≤ w(C)) then
        RemovalQueue := RemovalQueue ∪ {v};
while RemovalQueue ≠ ∅ do
    u := pop a vertex from RemovalQueue;
    remove u and its incident edges from G;
    foreach v ∈ N_r(u) do
        if (UB_0(v) ≤ w(C)) ∨ (UB_1(v) ≤ w(C_0)) then
            RemovalQueue := RemovalQueue ∪ {v};
return G;
```

3.2.2 SCCWalk

SCCWalk [6] is a state of the art heuristic algorithm proposed by Wang, Cai, Chen and Yin in 2020. It works by iteratively modifying several initial cliques, searching for better solutions. During that, two novel strategies called strong configuration checking and walk perturbation are used.

Operators

In general, four operators for changing a clique C are considered. For each operator, a set of vertices eligible for it is defined. Furthermore a value $\Delta_{operator}$ describes the amount by which the weight $w(C)$ of the clique changes after applying the operator.

The **Add** operator adds a vertex v that is adjacent to all vertices in C to the clique. v should not already a be part of C. The set of candidates for add is given as

$$AddSet = \{v \mid v \notin C, v \in N(u)\ \forall u \in C\}.$$

The weight of the clique after applying the Add operator will be increased by the weight of the added vertex, which yields

$$\Delta_{add}(v) = w(v) \mid v \in AddSet.$$

Drop is an operator for removing a vertex from C. Since this operator cannot be applied to vertices outside of the clique, we obtain

$$DropSet = \{v \mid v \in C\}.$$

The weight change of C after dropping a vertex v from it is defined as the negative weight of v:

$$\Delta_{drop}(v) = -w(v) \mid v \in DropSet.$$

Swap removes a vertex u from the clique C and adds another vertex v instead. Like in the Add operator v has to be a neighbor of all members of C except the removed u. This results in a set of eligible pairs of vertices

$$SwapSet = \{(u, v) \mid u \in C, v \notin C, v \in N(x)\ \forall x \in C \setminus \{u\}\}.$$

The clique weight will be changed by the difference of the weights of the two vertices involved in the swapping process, which produces

$$\Delta_{swap}(u, v) = w(v) - w(u) \mid (u, v) \in SwapSet.$$

Finally, the operator **Jump** adds some vertex v to C and removes every vertex u_i from the clique that is not a neighbor of v. While v does not have to be a neighbor of any member of C, it is required that more than one vertex u_i in C is not connected to v. Otherwise no vertex at all or just one would be removed by Jump and it would be equivalent to Add or Swap. The respective set is specified as

$$JumpSet = \Big\{\{u_1, \ldots, u_t, v\} \;\Big|\; v \notin C, \big(u_i \in C \setminus N(v)\ \forall i \in \{1, \ldots, t\} | t > 1\big)\Big\}.$$

The clique weight after performing Jump is increased by the weight of the added vertex $w(v)$ and decreased by the weights of the removed vertices $w(u_i)$:

$$\Delta_{jump}(u_1, \ldots, u_t, v) = w(v) - \sum_{i=1}^{t} w(u_i) \mid \{u_1, \ldots, u_t, v\} \in JumpSet.$$

Strong Configuration Checking

The idea of Configuration Checking (CC) in general is to avoid revisiting vertices when their usefulness for finding the solution has not changed. For each vertex v, a configuration change value $confChange(v)$ is stored. If the configuration of a vertex has changed since the last visit, $confChange(v) = 1$ is set, and v can be considered for adding it to the clique. Otherwise, we set $confChange(v) = 0$ which means that v is excluded from being added to the clique in construction. To employ a Configuration Checking strategy, rules for updating $confChange(v)$ are needed. In their version of CC, called Strong Configuration Checking (SCC), Wang et al. propose the rules SCC-InitialRule, SCC-AddRule, SCC-DropRule, SCC-SwapRule and SCC-JumpRule. The following paragraphs will explain those rules and examples for them and the operators in general. In the example graphs, a 1 in the upper right corner of a vertex v denotes that $confChange(v) = 1$. A 0 means that $confChange(v) = 0$.

The **SCC-InitialRule** sets $confChange(v) = 1\ \forall v \in V$. It is used to initialize all configuration change values at the start of a search, where every vertex is unexplored and could potentially be of interest.

When the add operator was applied to extend the clique by a vertex v, the **SCC-AddRule** sets the configuration change values of all its neighbor vertices v' to 1: $confChange(v') = 1\ \forall v' \in N(v)$. Figure 3.15 shows an example where initially we have a clique C containing v_2 and v_5. As only v_4 is adjacent to both v_2 and v_5, we have $AddSet = \{v_4\}$. Since also $confChange(v_4) = 1$ is given, the operation $Add(v_4)$ can be performed. After the Add operator is applied, C consists of v_2, v_4 and v_5. Now no other vertices are neighbors to very member of C, so $AddSet = \emptyset$. It is also trivial to see that $\Delta_{add}(v_4) = w(v_4^{0.1}) = 0.1$. Furthermore, changes of the configuration change values can be observed. All neighbors v' of v_4 now have $confChange(v') = 1$: for v_1 this implies a switch of the value form 0 to 1, all others already had it set to 1 before. v_6 is not adjacent to v_4, so $confChange(v_6)$ remains unchanged.

When a vertex v is dropped from the clique, the **SCC-DropRule** sets its configuration change value to 0, so that v does not get added to the clique again until its configuration is changed by another rule. An example can be seen in figure 3.16. Initially we have a clique with v_2, v_4 and v_5. The corresponding operator set contains all vertices that are in the clique, so $DropSet = \{v_2, v_4, v_5\}$. v_4 is picked to be dropped. After $Drop(v_4)$ is executed, it is removed from the clique and $confChange(v_4)$ has changed from 1 to 0 accordingly. All other configuration change values are not modified. The change of the clique weight is given by $\Delta_{drop}(v_4)) = -w(v_4^{0.1}) = -0.1$.

Similar to the SCC-DropRule, the **SCC-SwapRule** sets the configuration change value of the vertex that gets removed during the swap operation to 0. So if $Swap(u, v)$

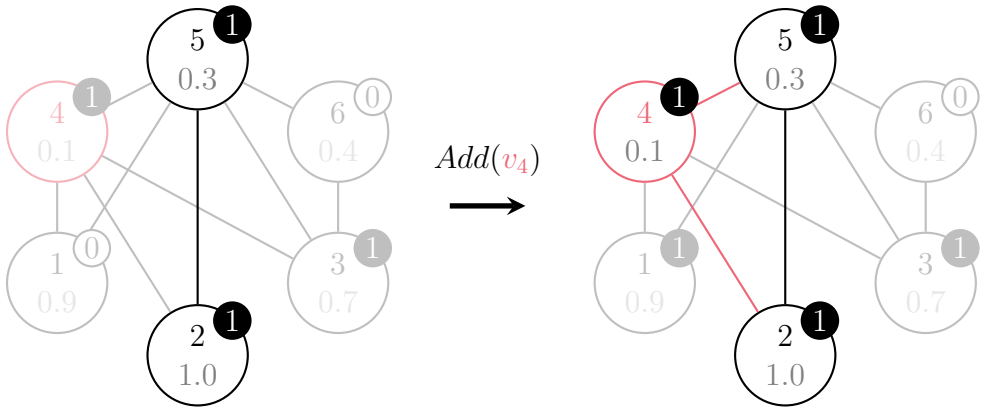

Figure 3.15: Example of an Add operation

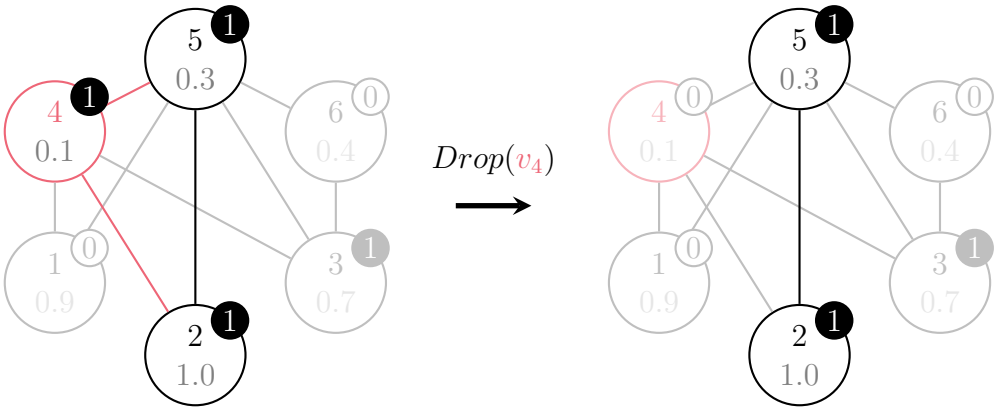

Figure 3.16: Example of a Drop operation

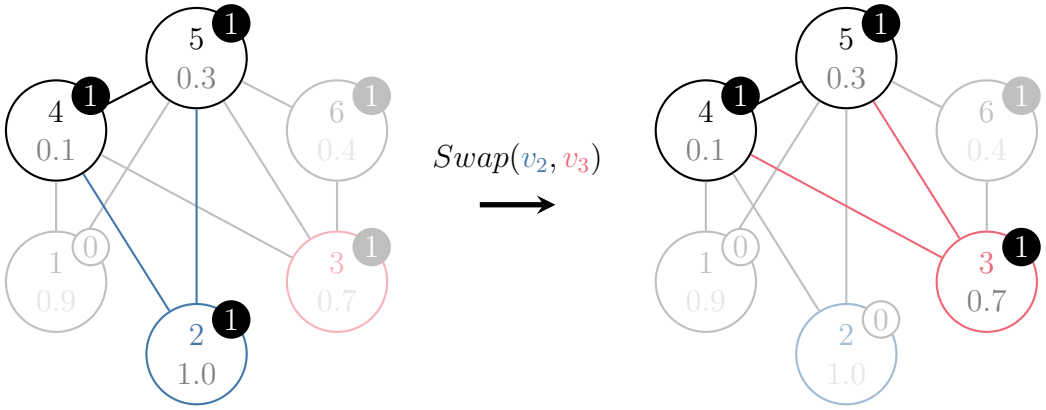

Figure 3.17: Example of a Swap operation

is applied, where v takes the place of u in the clique, the SCC-SwapRule ensures that $confChange(u) = 0$. Figure 3.17 presents an example. Initially, v_2, v_4 and v_5 are in the clique. The pairs eligible for swapping are the following: $SwapSet = \{(v_2, v_1), (v_2, v_3), (v_2, v_6)\}$. The second pair gets chosen for the operation, so we perform $Swap(v_2, v_3)$. v_2 gets removed from the clique and v_3 gets added. Since v_2 is equivalent to the u in the SCC-SwapRule, $confChange(v_2)$ changes from 1 to 0. The other configuration change values remain the same again. The weight change is $\Delta_{swap}(v_2, v_3) = w(v_3^{0.7}) - w(v_2^{1.0}) = 0.7 - 1.0 = -0.3$.

Likewise the **SCC-JumpRule** resets the configuration change values of the vertices $u_1, \ldots, u_t$ removed by $Jump(u_1, \ldots, u_t, v)$: $confChange(u_i) = 0 \; \forall i \in \{1, \ldots, t\}$. In the jump example in figure 3.18, the initial clique again consists of v_2, v_4 and v_5. The jump set contains one tuple: $JumpSet = \{(v_2, v_4, v_6)\}$. This one tuple is then used to execute $Jump(v_2, v_4, v_6)$. v_2 and v_4 are equivalent to the vertices u_1 and u_2 from the previous formal descriptions of the jump operation, so they get removed from C. v_6 on the other hand can be seen as the vertex v that gets added to the clique. The configuration change values of the removed vertices are unset, so after the operation we have $confChange(v_2) = 0$ and $confChange(v_4) = 0$. The weight delta can be calculated as $\Delta_{jump}(v_2, v_4, v_6) = w(v_6^{0.4}) - (w(v_2^{1.0}) + w(v_4^{0.1})) = 0.4 - (1.0 + 0.1) = 0.7$.

It can be observed that in strong configuration checking, after a vertex was once removed from the clique by any operation (which always sets $confChange(v) = 0$), it can only be considered for the clique again if one of its neighbors gets added (SCC-AddRule).

Main procedure

The main procedure of SCCWalk is portrayed in algorithm 17. First the solution clique $\hat{C}$ is initialized to an empty set. Like FastWClq, SCCWalk has a main loop

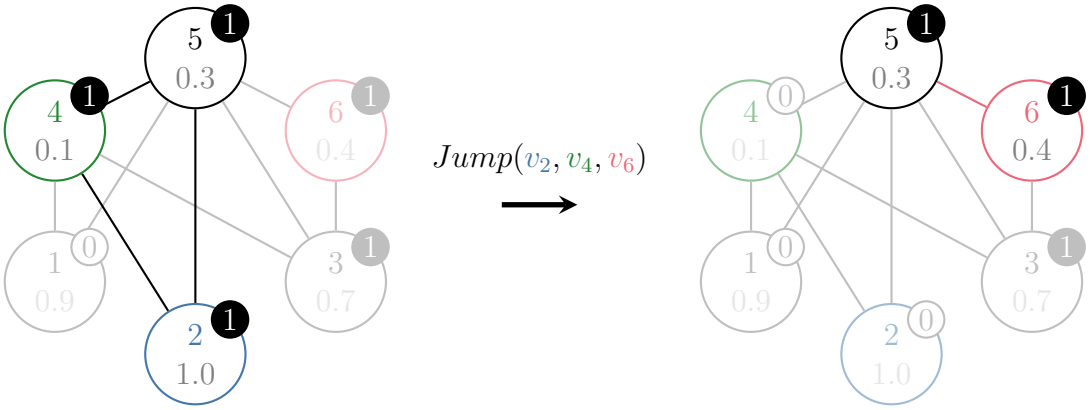

Figure 3.18: Example of a Jump operation

that runs until a set time limit is reached. However, there is no early return in SCCWalk, meaning it will not run in less time than the given cutoff time.

In the first phase (section 1) of each iteration the algorithm sets all configuration change values to 1 according to the SCC-InitRule. A clique C is initialized with a random maximal clique using the function $InitGreedySCCWalk$. Another clique that gets handled by the algorithm is $C_{localbest}$, it is initialized with the initial C. In the following inner for loop a local search from the initial C is performed. C gets modified using the operators introduced before in search for a better clique. $C_{localbest}$ is used to store the best clique found during that search. The loop variable $unimprovestep$ counts for how many iterations $C_{localbest}$ could not be improved. The parameter L controls how many steps without such an improvement are allowed. The authors use $L = 4000$ for graphs with a density below 0.99, and $L = 15000$ for higher densities. When $unimproveStep$ reaches its maximum value L, the outer loop goes into the next iteration and a new search from another initial clique is started.

During Section 2.1, SCCWalk modifies the clique using one of the operators Add, Swap or Drop. Firstly the algorithm determines the best vertex for the Add operation v and the best pair for the Swap operation u, u'. In each case, the best vertices are the ones where the weight change that the operation would result in is the highest. Vertices can only be taken into account if they are members of the sets corresponding to the operation and if their configuration change value is 1. In cases where two vertices (or vertex pairs) would induce the same weight change, the winner is determined by the vertex age and the older one is selected. The age of a vertex is given by the number of steps passed since it changed its membership status in the clique (included or excluded). As long as a possible Add operation exists, the operation yielding a higher weight change is chosen between the Add and Swap operation with the previously found best operands. If is C is maximal, $AddSet$ will be empty. In such cases, the best operation is picked between Swap and Drop. To determine the best Drop operand x, not every vertex from the $DropSet$ is taken

into account. Instead, only half of the vertices from $DropSet$ are randomly selected and put into the $DropSubSet$ used for the further selection process. Again, if multiple candidates for x provide the same benefit, the oldest one is preferred. After the chosen operator was applied, the corresponding SCC-Rule is used to adapt the configuration change values.

In section 2.2 the algorithm evaluates the success of the clique modification. If the resulting clique C is better than the best clique previously found in the local search $C_{localsearch}$, the latter one is set to the former one and the inner loop variable $unimproveStep$ is reset, since an improvement just happened. If that is not the case and no improvement took place since $maxUStep$ steps, it is assumed that the algorithm is stuck in limited area of the search space. To mitigate this problem, SCCWalk uses its walk perturbation strategy. The function $WalkPerturbationSCCWalk()$ is used to get a tuple of operands $u_1, \ldots, u_t, u'$ for a jump operation to get into another area of the search space. The jump operation is then performed and the SCC-JumpRule gets applied accordingly. If that jump operation improved C beyond $C_{localbest}$ the same updates as if the other operations yielded an improvement are performed. In the experiments in [6], $maxUStep$ is set to 500 for graph densities below 0.99 and otherwise to 3000.

After the inner loop is completed, in section 3, the solution clique $\hat{C}$ is updated to $C_{localbest}$ if the latter one is better. When the outer loop is finished, $\hat{C}$ is returned.

For very dense graphs with $d \geq 0.99$, in addition to the parameter adaptions already mentioned, removed vertices are blocked from being added to the clique again for 7 steps.

Greedy initialization

The greedy initialization sub-procedure as listed in algorithm 18 starts with picking a random vertex from the input graph. This vertex is used as the initial member of the clique C. After that vertices are added to C in a loop. In each iteration a set $Candidates$ is determined by choosing all vertices u that are neighbors of every member of the clique C. Unless $Candidates$ is empty, a random vertex v is selected from it and added to the clique. If $Candidates$ becomes empty, the clique cannot be extended anymore, i.e., it is a maximal clique. In this case the function is finished and returns the clique C. The random nature of this initialization is intended in order to get different initial cliques for better exploration in the outer loop of the main procedure.

Walk perturbation

The walk perturbation procedure is used to find a jump operation to escape a part of the search space that SCCWalk is trapped in. Its formal description is listed

Algorithm 17: SCCWalk

$\hat{C} = \emptyset$;
while *within time limit* **do**
1 apply SCC-InitRule;
$C := InitGreedySCCWalk()$;
$C_{localbest} := C$;
for $unimproveStep := 0$ *to* $L - 1$ **do**
2.1 $v := \underset{z \in AddSet \mid confChange(z)=1}{\arg\max} \Delta_{add}(z)$;
$(u, u') := \underset{(z,z') \in SwapSet \mid confChange(z')=1}{\arg\max} \Delta_{swap}(z, z')$;
if $AddSet \neq \emptyset$ **then**
if $\Delta_{add}(v) > \Delta_{swap}(u, u')$ **then**
$C := C \cup \{v\}$;
else
$C := C \setminus \{u\} \cup \{u'\}$;
else
$DropSubSet :=$ $|DropSet|/2$ random vertices from $DropSet$;
$x := \underset{z \in DropSubSet}{\arg\max} \Delta_{drop}(z)$;
if $\Delta_{drop}(x) > \Delta_{swap}(u, u')$ **then**
$C := C \setminus \{x\}$;
else
$C := C \setminus \{u\} \cup \{u'\}$;
apply SCC-AddRule, SCC-DropRule and SCC-SwapRule;
2.2 **if** $w(C) > w(C_{localbest})$ **then**
$C_{localbest} := C$;
$unimproveStep := 0$;
else if $(unimproveStep \ \% \ maxUStep) = (maxUStep - 1)$ **then**
$(u_1, \ldots, u_t, u') := WalkPerturbationSCCWalk()$;
$C := C \setminus \{u_1, \ldots, u_t\} \cup \{u'\}$;
apply SCC-JumpRule;
if $w(C) > w(C_{localbest})$ **then**
$C_{localbest} := C$;
$unimproveStep := 0$;
3 **if** $w(C_{localbest}) > w(\hat{C})$ **then**
$\hat{C} := C_{localbest}$;
return $\hat{C}$;

in algorithm 19. First, a random allowed jump operation is picked and its delta

Algorithm 18: InitGreedySCCWalk

$C :=$ {one random vertex};
repeat
  $Candidates := \{u \mid u \in N(x)\ \forall x \in C\}$;
  if $Candidates \neq \emptyset$ **then**
    $v :=$ random vertex from $Candidates$;
    $C := C \cup \{v\}$;
until $Candidates = \emptyset$;
return C;

is calculated and saved as Δ^*_{jump}. In the loop that follows, the walk perturbation procedure tries to sample the jump operation with the highest (positive) weight change out of *candJump* samples. The authors recommend to set the parameter *candJump* to 100. If a sampled jump operation $Jump(u'_1, \ldots, u'_t, v')$ has a higher delta than the highest one found so far, it is set as the new best jump operation tuple $(u_1, \ldots, u_t, v)$. If a jump operation has the same delta as the best one found so far, it is only chosen if its v' has a higher age than v. v' is the vertex that gets added in the sampled jump operation, v the one added in the current best jump operation.

Algorithm 19: WalkPerturbationSCCWalk

pick a random $(u_1, \ldots, u_t, v) \in JumpSet \mid confChange(v) = 1$;
$\Delta^*_{jump} := \Delta_{jump}(u_1, \ldots, u_t, v)$;
for $i := 1$ *to* $candJump$ **do**
  pick a random $(u'_1, \ldots, u'_t, v') \in JumpSet \mid confChange(v') = 1$;
  if $\Big(\Delta_{jump}(u'_1, \ldots, u'_t, v') > \Delta^*_{jump}\Big) \vee \Big(\big(\Delta_{jump}(u'_1, \ldots, u'_t, v') = \Delta^*_{jump}\big) \wedge \big(age(v') > age(v)\big)\Big)$ **then**
    $(u_1, \ldots, u_t, v) := (u'_1, \ldots, u'_t, v')$;
    $\Delta^*_{jump} := \Delta_{jump}(u'_1, \ldots, u'_t, v')$;
return $(u_1, \ldots, u_t, v)$;

3.2.3 SCCWalk4L

Together with the original SCCWalk, the authors also proposed a derivative called SCCWalk4L in [6]. It aims to be more suitable for large graphs, hence the suffix '4L'. It employs Best from Multiple Selection (BMS) and the reduction rules UB_0 and UB_1 that have already been used by FastWClq [7].

The main procedure of SCCWalk4L is shown in algorithm 20. It can be seen that it is very similar to the original SCCWalk. The lines where the walk perturbation is performed are left out in the listing for space reasons, they are the same as in algorithm 17. The first difference to the original algorithm is the selection of the swap pair. The change is marked with *A in the listing. Instead of choosing the best swap pair from the whole $SwapSet$, it gets picked by a special heuristic which is implemented in the function $GetSwapPairSCCWalk4L()$. Another significant difference is marked with *B. When finding a new best clique $\hat{C}$, the graph is reduced by the sub procedure $ReduceGraphSCCWalk4L$. The reduction function will eliminate vertices from the graph G and the current clique C which could never be part of a clique better than the updated $\hat{C}$. Furthermore the authors of [6] suggest to set the parameters $maxUStep = 50$ and $L = 400$ for large sparse graphs and $maxUStep = 250$ and $L = 4000$ for medium sized graphs with higher densities.

Swap pair selection

Checking all possible swap pairs like in SCCWalk could take too long in large graphs. Thus a BMS strategy is realized by the function $GetSwapPairSCCWalk4L()$. As a starting point, this function takes a random swap pair (v, v') from the $SwapSet$. The calculated weight change corresponding to the swap operation with (v, v') is set as the initial value of the best weight change Δ^*_{swap}. After this, k random pairs (u, u') from the $SwapSet$ are sampled and the one pair (v, v') with the best weight change is returned at the end. Should there be a tie regarding the weight change Δ_{swap}, the pair where the vertex that would be added by the swap operation (u' or v') has a higher age is preferred. As in FastWClq, the BMS number k controls the exploration by determining how many possible swap pairs should be tested. However, this time k has a fixed value during the whole execution of the algorithm. Wang et al. recommend setting $k = 100$ for SCCWalk4L.

Graph reduction

The graph reduction procedure of SCCWalk4L is shown in algorithm 22. It can be seen that it is quite similar to the one of FastWClq (algorithm 16). A reduction according to the more advanced upper bound UB_1 is only performed if the graph already has less than 100,000 vertices. In contrast to FastWClq, the reduction takes place while the working clique C is still being modified and explored. Because of this, vertices that get removed from the graph and are a part of C have to get removed from the clique as well.

Algorithm 20: SCCWalk4L

$\hat{C} = \emptyset$;
while *within time limit* **do**
1 apply SCC-InitRule;
$C := InitGreedySCCWalk()$;
$C_{localbest} := C$;
for $unimproveStep := 0$ *to* $L - 1$ **do**
2.1 $v := \underset{z \in AddSet \mid confChange(z)=1}{\arg\max} \Delta_{add}(z)$;
*A $(u, u') := GetSwapPairSCCWalk4L()$;
if $AddSet \neq \emptyset$ **then**
if $\Delta_{add}(v) > \Delta_{swap}(u, u')$ **then**
$C := C \cup \{v\}$;
else
$C := C \setminus \{u\} \cup \{u'\}$;
else
$DropSubSet :=$ |$DropSet$|/2 random vertices from $DropSet$;
$x := \underset{z \in DropSubSet}{\arg\max} \Delta_{drop}(z)$;
if $\Delta_{drop}(x) > \Delta_{swap}(u, u')$ **then**
$C := C \cup \{x\}$;
else
$C := C \setminus \{u\} \cup \{u'\}$;
apply SCC-AddRule, SCC-DropRule and SCC-SwapRule;
2.2 **if** $w(C) > w(C_{localbest})$ **then**
$C_{localbest} := C$;
$unimproveStep := 0$;
else if $(unimproveStep \ \% \ maxUStep) = (maxUStep - 1)$ **then**
... // $WalkPerturbation$
3 **if** $w(C_{localbest}) > w(\hat{C})$ **then**
$\hat{C} := C_{localbest}$;
*B $G, C := ReduceGraphSCCWalk4L(G, \hat{C}, C)$;
if $V = \emptyset$ **then**
return $\hat{C}$;
if $C = \emptyset$ **then**
break;
return $\hat{C}$;

Algorithm 21: GetSwapPairSCCWalk4L

pick a random $(v, v') \in SwapSet \mid confChange(v) = 1$;
$\Delta^*_{swap} := \Delta_{swap}(v, v')$;
for $i := 1$ *to* k **do**
  pick a random $(u, u') \in SwapSet \mid confChange(u') = 1$;
  if $\Big(\Delta_{swap}(u, u') > \Delta^*_{swap}\Big) \vee \Big(\big(\Delta_{swap}(u, u') = \Delta^*_{swap}\big) \wedge \big(age(u') > age(v')\big)\Big)$ **then**
    $(v, v') := (u, u')$;
    $\Delta^*_{swap} := \Delta_{swap}(u, u')$;
return (v, v');

Algorithm 22: ReduceGraphSCCWalk4L$(G, \hat{C}, C)$

foreach $v \in V$ **do**
  if $\Big(UB_0(v) \leq w(\hat{C})\Big) \vee \Big(\big(|V| \leq 100,000\big) \wedge \big(UB_1(v) \leq w(\hat{C})\big)\Big)$ **then**
    $RemovalQueue := RemovalQueue \cup \{v\}$;
while $removalQueue \neq \emptyset$ **do**
  $u :=$ pop a vertex from $removalQueue$;
  remove u and its incident edges from G;
  if $v \in C$ **then**
    $C := C \setminus \{v\}$;
  foreach $v \in N_r(u)$ **do**
    if $\Big(UB_0(v) \leq w(\hat{C})\Big) \vee \Big(\big(|V| \leq 100,000\big) \wedge \big(UB_1(v) \leq w(\hat{C})\big)\Big)$ **then**
      $RemovalQueue := RemovalQueue \cup \{v\}$;
return G, C;

3.2.4 MWCPeel

Together with MWCRedu, which was already introduced in chapter 3.1.4, the authors also proposed an heuristic algorithm named MWCPeel in the same paper [17]. It makes use of the same reduction rules as MWCRedu. Its pseudo-code is shown in algorithm listing 23.

First, an initial clique is computed like in MWCRedu. After that follows the reduction loop. In every iteration, first the exact reduction rules presented in 3.1.4 are applied by the function *ReduceMWCRedu*. Only in the first iteration of the reduction loop, a vertex degree limit (parameter *lim*) is used in the reduction. This is followed by a heuristic reduction, the so-called peeling. A certain percentage of the vertices gets removed based on a score. The percentage will be 10 % for graphs

with 50.000 and more vertices, 1 % for graphs with 5.000 or less vertices and linearly interpolated between 10 % and 1 % for graph sizes in between [23]. The score is the weight of the inclusive neighborhood $w(N[v])$. So those vertices get removed, for which the upper bound for the weight of the cliques they can be in is the lowest. Those two reduction strategies are applied over and over until the maximum score in the current reduced graph $max_{v \in V'}(w(N[v]))$ is less than 90 % of the maximum score in the original graph $max_{u \in V}(w(N[u]))$. The reduction is also stopped if the difference between the maximum score in the reduced graph $max_{v \in V'}(w(N[v]))$ and the minimum score in the reduced graph drops below 90 %. After the loop, the reduced graph is again passed to the exact TSM-MWC-Algorithm.

Algorithm 23: MWCPeel

$\hat{C} := InitializeWLMC(G, 0)$;
$isFirstIteration := true$;
repeat
  $G, \hat{C} := ReduceMWCRedu(G, \hat{C}, isFirstIteration)$;
  $isFirstIteration := false$;
  $num := \begin{cases} 0.1 \cdot |V| & |V| > 50,000, \\ \max(0.01 \cdot |V|, 0.1 \cdot \frac{|V|}{50,000}) & \text{otherwise} \end{cases}$
  remove num vertices with the lowest score $w(N[v])$;
until *stopping criteria met*;
return *TSM-MWC(G, $\hat{C}$)*;

3.3 Other approaches

The algorithms that were explained in the previous chapters only present a curated subset of all currently available maximum weight clique algorithms. Some further examples are Östergård's Algorithm [24], Kumlander's Algorithm [25], MaxWClq [26] and OTClique [27]. The selection was made to keep the scope of the thesis feasible. Not every approach that was considered (according to the literature) clearly outperformed by other ones was taken into account. Some interesting and more recent methods that use machine learning (see [28] and [23]) were proposed as well that did not make it into this thesis either.

As a first naive approach, some tests with an adapted variants of the Bron-Kerbosch algorithm [29] were also made. The Bron-Kerbosch algorithm finds all maximal cliques in a graph, and when filtering those cliques for the highest weighted one, it can also be used to determine the maximum weight clique. First experiments showed that its performance is notably worse than the algorithms presented in the previous chapter. Because of that and since the Bron-Kerbosch algorithm is not a real MWC algorithm to begin with, it was excluded from this thesis as well.

4 Realization

This chapter starts with the description of the own custom algorithmic approaches that have been used. After that, the test data used for the experiments is specified. Finally the test setup that is used to actually gather the measurements is introduced. This test setup was used run all algorithms and their variants, both the new ones described in this chapter and those from the literature, on the mentioned test data.

4.1 Custom algorithms

4.1.1 UEW

The first custom created algorithm is named UEW which stands for **un**explored **w**eight. It can be described as a one-shot greedy algorithm. UEW is a heuristic algorithm that will only construct one clique. It is expected that it will be very fast. Regarding the optimality of its solutions, the algorithm will not provide any guarantee or even proof. The found result clique might end up being sub-optimal more often compared to other heuristic algorithms. These suspected properties will have to be analyzed during the experiments later on.

Besides the clique $\hat{C}$ that is being built, the algorithm holds a set $Candidates$ containing all vertices that could be added to $\hat{C}$ in the current step. The central measure of this heuristic algorithm, the "unexplored weight" of a vertex is calculated by

$$w_{UE}(v) = \alpha \cdot w_O(v) + \sum_{u \in N(v)} w_O(u).$$

The factor α is a weight that determines how much the weight of a vertex is prioritized in comparison to its neighbors' weights. This parameter is also referred to as the "own weight priority". A value ≥ 1 is recommended. $w_O(v)$ is the "own unexplored weight" defined as

$$w_O(v) = \begin{cases} w(v) & v \in Candidates, \\ 0 & \text{otherwise.} \end{cases}$$

The procedure of UEW outlined in algorithm listing 24 starts with setting all vertices as potential candidates to be added to the clique. After that the main loop starts

where in each iteration a vertex is added to the result clique until no more candidates are left. This is done by selecting the vertex from the candidates that has the highest unexplored weight. The set of candidates has to be updated each time a vertex is added to the clique. The vertex which is added gets removed from $Candidates$ as it already is in the clique now, furthermore $Candidates$ is intersected with the neighbors of the added vertex to make sure the candidate set only contains vertices that would keep $\hat{C}$ a valid clique if they were added to it.

Algorithm 24: UEW

$Candidates := V$;
$\hat{C} := \emptyset$;
while $Candidates \neq \emptyset$ **do**
  $c := \underset{v \in Candidates}{\arg\max}\ w_{UE}(v)$;
  $\hat{C} := \hat{C} \cup \{c\}$;
  $Candidates := Candidates \setminus \{c\}$;
  $Candidates := Candidates \cap N(c)$;
return $\hat{C}$;

4.1.2 UEW-R

The second custom created algorithm is UEW-R, where the R stands for "Reduction". It is a derivative of UEW, that also incorporates concepts of FastWClq. It creates cliques iteratively and applies graph reductions in between. In cases where the reduction step reduces the graph to an empty set, just like FastWClq, UEW-R can even guarantee the optimality of its solution.

The pseudocode of UEW-R can be seen in algorithm 25. The best found clique is stored in $\hat{C}$ and initialized to an empty set. There is an outer loop in which the clique search and the graph reduction happen. The loop terminates when the graph is reduced to an empty set. At the end, $\hat{C}$ is returned.

Section 1 of the outer loop of UEW-R creates a new clique $C_{current}$ in each iteration. For this it is using the same strategy as the classic UEW algorithm. Again, in its inner loop the clique $C_{current}$ is assembled by always choosing the candidate vertices with the highest unexplored weight. When a new iteration of the outer loop begins, not only is $C_{current}$ reinitialized, but also the set $Candidates$ is reset to contain all vertices of the graph again. Since later in the algorithm the graph is reduced, V can be different and consequently the resulting $C_{current}$ can be another clique as well.

In section 2 the clique $C_{current}$ created in the current iteration gets compared to the best clique $\hat{C}$ so far. Due to the nature of the algorithm, $\hat{C}$ can also be seen as the $C_{current}$ of the previous iteration. If the comparison is positive and the new

clique has a higher weight, $\hat{C}$ is updated accordingly and the graph is reduced by the function *ReduceGraphFastWClq* and the outer loop can start its next iteration. UEW in general can choose a suboptimal clique if it gets "distracted" by a vertex with a high-weight inclusive neighborhood, where not many of its neighbors have edges in between each other. If the graph gets reduced and vertices that cannot be part of the best clique are removed, this results in less distractions for the UEW strategy. That in turn increases the probability that a more optimal solution is found. If however the weight comparison of $C_{current}$ and $\hat{C}$ shows that $\hat{C}$ is better or at least of equal quality, the algorithm terminates. Without improvements of $\hat{C}$, the graph reduction would not change the graph. This would lead to the $C_{current}$ constructed in the next iterations always being the same, leaving the algorithm in an inifinite loop.

Algorithm 25: UEW-R

```
Ĉ := ∅;
repeat
1   C_current := ∅;
    Candidates := V;
    while Candidates ≠ ∅ do
        c := arg max_{v∈Candidates} w_UE(v);
        C_current := C_current ∪ {c};
        Candidates := Candidates \ {c};
        Candidates := Candidates ∩ N(c);
2   if w(C_current) > w(Ĉ) then
        Ĉ := C_current;
        G := ReduceGraphFastWClq(G, Ĉ);
    else
        return Ĉ;
until G = ∅;
return Ĉ;
```

4.1.3 Optimizations for similar graphs

One thing that many algorithms have in common, is that a current best clique is held across iterations and updated when a better one is found. Some algorithms like MWCRedu and MWCPeel even require an initial clique which needs to be determined by some heuristic. This seems to present a suitable opportunity to reuse the best clique $\hat{C}_0$ found in a similar graph G_0. Some vertices from $\hat{C}_0$ may not be present anymore in G_1 or not pairwise adjacent in G_1. Therefore a clique $\hat{C}'_0 \subseteq \hat{C}_0$ is determined. To obtain $\hat{C}'_0$, vertices are iteratively removed from $\hat{C}_0$ until a valid

clique within G_1 results. In the best case $\hat{C}'_0$ and $\hat{C}_0$ will be the same. In the worst case no member of $\hat{C}_0$ is present in G_1 and consequently $\hat{C}'_0$ will be empty. When processing the iterative version of an algorithm, $\hat{C}'_0$ is set as the initial best clique, and the algorithm is executed as usual after that. This strategy is applied to WLMC, WC-MWC, TSM-MWC, MWCRedu, FastWClq, SCCWalk, SCCWalk4L, MWCPeel and UEW-R. It is not applied to UEW, since as a one-shot algorithm, it has no ability to recover from a unsuitable initial clique. In the iterative version of UEW-R, an additional graph reduction based on the weight of $\hat{C}'_0$ was performed before the beginning of the outer loop.

If the differences between the graphs G_0 and G_1 do not have a major influence on the highest weighted cliques, $\hat{C}'_0$ might already be the best clique in G_1. Even if there is a better clique in G_1, $w(\hat{C}'_0)$ is still likely to be a rather tight lower bound for the weight of the best clique. This means that graph reductions based on such lower bounds can potentially eliminate large parts of the graph G_1 even in the first iteration of a reduction process. Working on a quite small graph from the start instead of only getting a smaller graph over the time could speed up the search for a solution significantly.

Initial clique conversion

Algorithm listing 26 specifies the full procedure of converting $\hat{C}_0$ to $\hat{C}'_0$. It is important to note that every time neighborhoods are used in this algorithm, this refers to the neighborhoods within the current graph G_1. In the first step, $\hat{C}'_0$ is initialized with $\hat{C}_0$ but without the vertices that do not exist in the current graph G_1. After that a loop is executed until $\hat{C}'_0$ becomes a valid clique in G_1. Within the loop, a set of *RemovalCandidates* is determined. *RemovalCandidates* are all members of $\hat{C}'_0$ that are not adjacent to some other members of $\hat{C}'_0$. A vertex v from the *RemovalCandidates* will be in conflict with its non-neighbors $\hat{C}'_0 \setminus N[v]$ of the *RemovalCandidates* set. v and every $x \in \hat{C}'_0 \setminus N[v]$ cannot be members of a clique at the same time. For every removal candidate v, the difference between the weight of the vertices conflicting with v and the weight of v itself is calculated. This difference represents the benefit of removing v instead of the vertices it conflicts with. With all the differences calculated, the vertex u with the highest difference is removed from $\hat{C}'_0$. When the loop terminates, all conflicts within $\hat{C}'_0$ are resolved and it can be returned.

4.2 Test data

The performed tests can be divided into two categories. On one hand, there are the standalone runs, where the algorithms are executed on each graph in an isolated manner. The other test category are the iterative runs. For the latter, the algorithms

Algorithm 26: ConvertInitialClique($G_1, \hat{C}_0$)

$\hat{C}'_0 := \hat{C}_0 \setminus \{v \mid v \notin G_1\}$;
while $\hat{C}'_0$ *is not a clique in* G_1 **do**
  $RemovalCandidates := \{v \mid v \in \hat{C}'_0, \hat{C}'_0 \setminus N[v] \neq \emptyset\}$;
  $u := \underset{v \in RemovalCandidates}{\arg\max} \; w(\hat{C}'_0 \setminus N[v]) - w(v)$;
  $\hat{C}'_0 := \hat{C}'_0 \setminus \{u\}$;
return $\hat{C}'_0$;

would first be executed on one graph the same way as they were in the standalone case. After that, adapted versions of the algorithms would be executed on graphs similar to the graphs used in the first execution step reusing the information about the maximum weight clique from the first step. For both test categories, various graphs were randomly generated. The properties of those graphs are described in the following two chapters.

In general the term **Algorithm Run** is used to describe a single execution of an algorithm on a single graph. There can be multiple algorithm runs of the same combination of graph and algorithm, which can have varying runtimes and in the case of heuristic algorithms even output different cliques.

4.2.1 Test graphs for standalone runs

For the standalone runs, the used set of different vertex numbers was $N_{Test} = \{25, 50, 75, 100, 125, 150, 200, 250, 500, 1000, 2000, 4000\}$. The density set was $D_{Test} = \{0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.5\}$. Furthermore, three different vertex weight distributions were used. The first one had all vertex weights set to 1.0, which makes the situation equivalent to a maximum clique problem. Next, a uniform distribution with weights in the interval $[0.001, 1.0]$ was used. The last was a normal distribution with $\mu = 0.5$ and $\sigma = 0.25$ which was also cut off to fit the interval $[0.001, 1.0]$.

To generate the edges, first the number of edges required by the density d and vertex count n was determined by $m = \frac{d*n(n-1)}{2}$. To create graphs where not all vertices have a similar amount of neighbors, some vertices need to be more likely than others to be part of an edge. Therefore the probability of choosing vertex $v_x \in \{v_1, \ldots, v_{|V|}\}$ for an edge is multiplied with a factor

$$\alpha_x = \left\lceil \frac{|V|}{3}(x + 0.5)^{-2} \right\rceil.$$

Afterwards the probabilities are normalized. Additionally the already chosen edges were kept track of to avoid picking the same edge multiple times.

For each combination of vertex number, density and weight distribution, three different graphs were generated. To provide a better intuition for the scale of the graphs, their respective edge counts are listed in table 4.1. In total, 864 graphs were generated and tested with the algorithms.

n \ d	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5
25	0	1	3	15	30	60	90	150
50	1	6	12	61	122	245	367	612
75	2	13	27	138	277	555	832	1387
100	4	24	49	247	495	990	1485	2475
125	7	38	77	387	775	1550	2325	3875
150	11	55	111	558	1117	2235	3352	5587
200	19	99	199	995	1990	3980	5970	9950
250	31	155	311	1556	3112	6225	9337	15562
500	124	623	1247	6237	12475	24950	37425	62375
1000	499	2497	4995	24975	49950	99900	149850	249750
2000	1999	9995	19990	99950	199900	399800	599700	999500
4000	7998	39990	79980	399900	799800	1599600	2399400	3999000

Table 4.1: Numbers of edges m of the generated graphs

4.2.2 Test graphs for iterative runs

To test the performance of the iterative variants of the algorithms, for every standalone run graph ("original graph") two graphs similar to it were generated. One of the similar graphs was generated by taking the standalone graph and with a 2% chance each removing an existing vertex, adding a new vertex, removing an existing edge and adding a new edge. The vertex weights were changed by a value which was sampled from a normal distribution with $\mu = 0.0$ and $\sigma = 0.01$. The second similar graph was generated with the same method as the first one, but this time with a 5% probability of each add- and remove-operation. The vertex weight change for the second similar graph was sampled from a normal distribution with $\mu = 0.0$ and $\sigma = 0.1$. In both cases, the number of edges for the newly added vertices was chosen according to the density of the original graph, and the weights of the new vertices were also generated from the same distribution as the original graph.

For the iterative run tests, the newly generated similar graphs were treated as the "older" graph G_0, while the original graph was processed as if it was G_1, the newer one. For each of the generated similar graphs, the test procedure was to first run the

normal algorithms on the similar graphs. The resulting clique was then fed as a seed into another algorithm run, where an iterative version of the algorithm processed the original graph. The results of the iterative processing of the original graph could then be compared to the non-iterative processing of it, which was already conducted during the standalone run tests. This comparison can be used to reveal whether there is a benefit to the iterative approach.

4.3 Evaluation criteria

The first criterion that all algorithms will be evaluated on is their **runtime** t on each of the graphs. Runtimes over 5 minutes are not considered. The reasons for this time limit is that the main focus of the thesis are smaller graphs, whose maximum weight clique should be found quickly and sometimes even in real time. Additionally a lot of graphs are evaluated, which in turn means that the allowed time per algorithm run must be limited to keep the tests temporally feasible. If an algorithm exceeds this time limit on particular graphs, those graphs are viewed as not solved by the algorithm and not taken into account when calculating average runtimes for a certain graph type with this particular algorithm. Nevertheless the information about which graphs could not be solved in time by an algorithm is still kept track of, since it provides insight into the limitations of the algorithm.

The second evaluation criterion is the **quality** q of the solution. For a particular graph, the solution quality of an algorithm run a_i is defined by the ratio

$$q(a_i) = \frac{w(\hat{C}_{a_i})}{w(\hat{C}_{exact})}$$

where $\hat{C}_{a_i}$ is the best clique found by the algorithm run a_i and $\hat{C}_{exact}$ is the exact solution for the maximum weight clique problem on that particular graph. To determine the quality for heuristic runs on a graph, it is necessary to get the graph processed by an exact algorithm at least once, if needed without a time limit.

4.4 Test environment

4.4.1 Own implementations

All of the presented algorithms were implemented from scratch in C++, sharing commonly needed code like classes for managing graphs and independent sets between the algorithms. Those implementations mostly stayed close to the pseudo code in the listings which can be found in the corresponding chapters about the algorithms. The decision to create own implementations was made because reference

implementations from the authors were not available for all of the chosen algorithms, and naturally could not be available for the newly introduced UEW, UEW-R and iterative algorithm versions. Furthermore implementing the algorithms on a common code base should help the comparability of the results. It can reduce performance differences which are caused by differently implemented base functionality instead of the actual concepts that are characteristic to the particular algorithms. All the own implementations operate on double precision floating point type weights. Although the authors of some of the algorithms might not have intended them to work with those kinds of weights, experiments have shown that the concepts of the algorithms still work under those conditions.

In addition to the algorithms themselves, utility classes to generate, manage, measure, import and export both algorithm runs and graphs were implemented. Based on this whole framework a set of programs was build. The "TestGraphGenerator" realizes the generation of the test graphs which are described in chapter 4.2. The "TestRunGenerator"-program generates algorithm run specification files for all the combinations of test graphs and algorithms, including different parametrizations of the algorithms. Finally the "SingleRunExecutor" takes an such an algorithm run specification file, executes the respective algorithm on the respective graph, and exports the measured runtime as well as the clique that was found by the algorithm. For development purposes, also the program "CliqueExplorer" was created. It allows to generate, manipulate and compare graphs and algorithm runs through a graphical user interface. A screenshot of CliqueExplorer is shown in figure 4.1. Used third-party libraries are Qt [30] for the graphical user interface, GraphViz [31] for visualizing graphs and pugixml [32] for reading and writing XML files.

4.4.2 Reference implementations

In addition to the own implementations of the algorithms, available reference implementations of the authors were tested. The source code of WLMC, TSM-MWC and FastWClq (newer version as presented in [22]) is available for download on the web pages of Prof. Chu-Min Li [33] and Prof. Shaowei Cai [34]. The reference implementations of WC-MWC [18] and SCCWalk4L [6] were kindly provided by their authors. Reference implementations for MWCRedu and MWCPeel were unfortunately not available at the time of the experiments, since the corresponding paper [17] was not published yet. The reference implementations were integrated into the test suite described in 4.4.1. Their initialization procedures were adapted so that the graph data type used in the test suite was converted to the internal formats of the reference implementations. Besides that and minor adaptions needed to make everything compile, the reference implementations were left in their original state.

The reference implementations were written in C or C++. They are all programmed in a rather low level fashion, storing most of their data in global arrays or vectors, while rarely making use of classes or structs to encapsulate the data they work with.

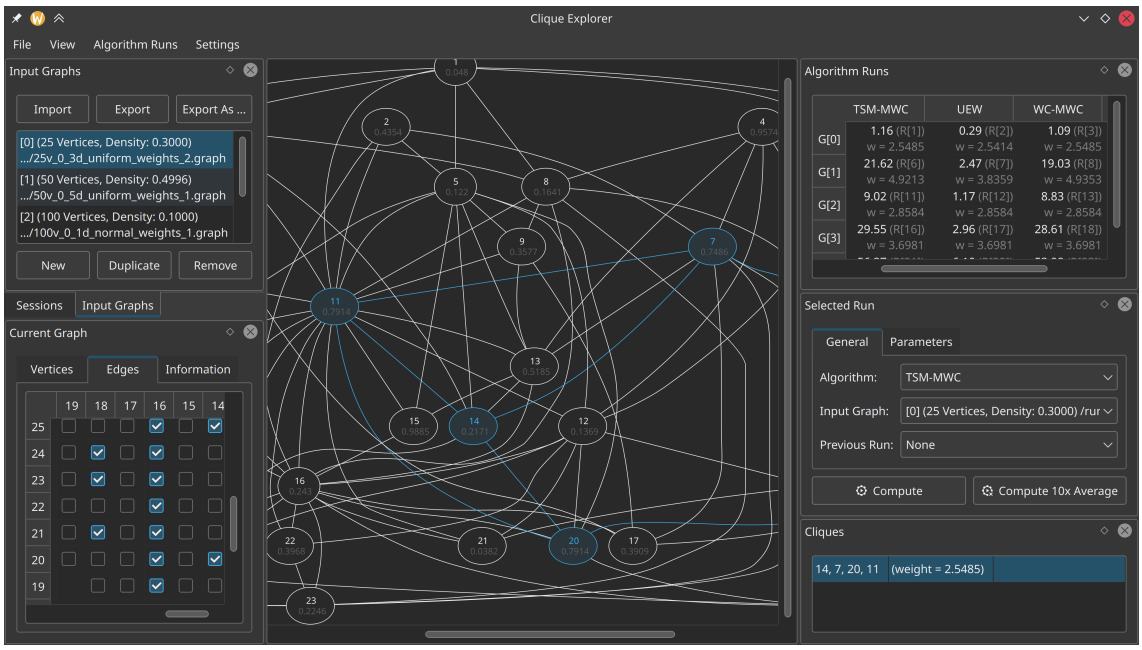

Figure 4.1: Screenshot of the program CliqueExplorer

Most of the reference implementations contain all of their code in one large source file. In general the code of the reference implementations is less close to the algorithm listings than the code of the own implementations. The reference algorithms only accept integer vertex weights. As the graphs a generated with double precision floating point weights, the weights are premultiplied with a constant factor of 100,000 before being converted to integers and passed to the reference implementations.

4.4.3 Execution

For the actual final tests whose results will be presented in the next chapter, every algorithm was run three times on each graph. From those three runtimes the average was calculated. If an algorithm took longer than 5 minutes to process a graph, its execution was canceled.

The runs were executed on three Raspberry Pi 4 Model B with 8 GB of RAM and a Quad core Cortex-A72 processor running at 1.8 GHz [35]. The used operating system was Raspberry Pi OS Lite (64-Bit) released on 11th December 2023, which is based on Debian 12 (bookworm) [36]. Each Raspberry Pi was additionally equipped with an active cooling case to mitigate thermal throttling, and a 4-digit display to show the number of remaining runs. The physical test setup can be seen in figure 4.2. On each of the computers the different algorithms were executed on all graphs with at least three minutes of cool-down time between the runs belonging to one algorithm.

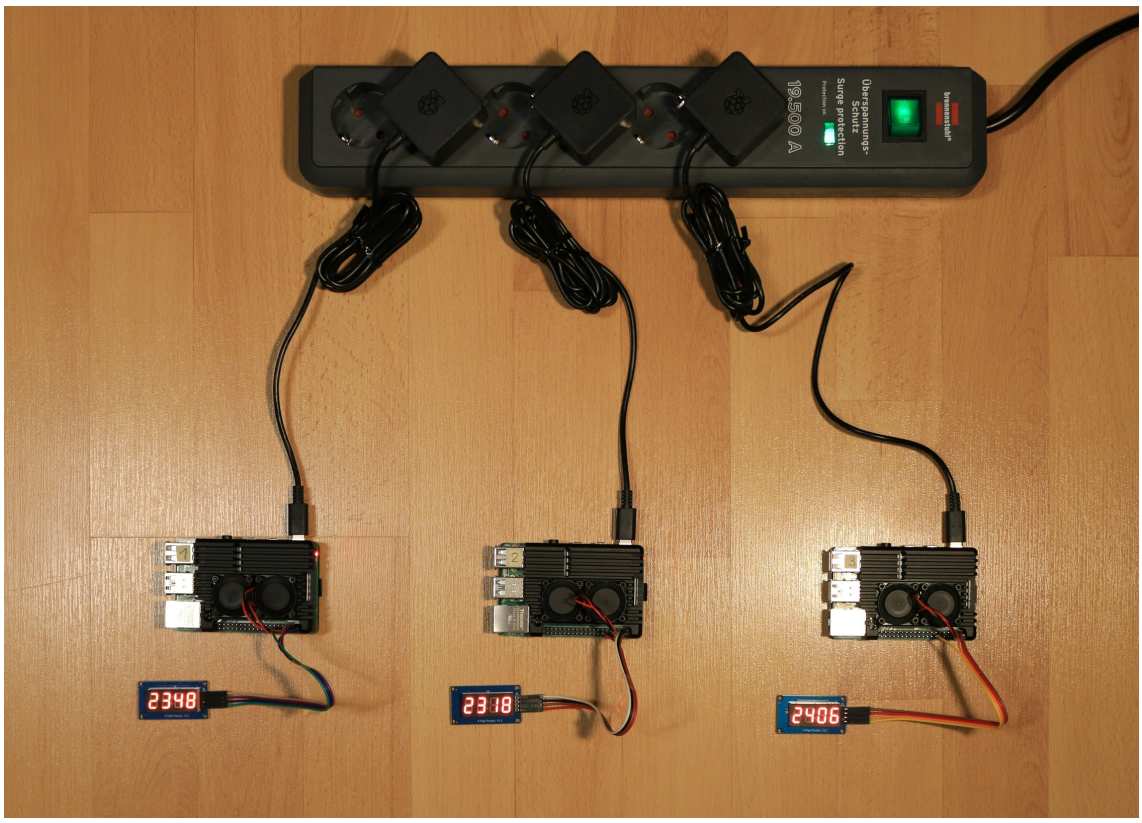

Figure 4.2: The test setup

The order in which the graphs were processed by an algorithm was randomized. Per computer up to three graphs were processed in parallel, sparing one core for orchestrating the runs and potential system tasks.

5 Results

This chapter will first present the most important measurements obtained from the test setup described in chapter 4. More detailed measurement data can be found in the appendix and on the attached DVD. Secondly, conclusions regarding the suitability of the algorithms for different scenarios are drawn from the data.

For calculating the qualities of the heuristic algorithm runs, the results of the reference implementation of TSM-MWC were used. In cases where the latter did not finish on a Raspberry Pi within 5 minutes, it was executed again on a desktop computer without a time limit. For the graphs with 4000 vertices and a density of 0.5 no ground truth cliques and consequently no qualities could be computed. The runs that where supposed to provide a ground truth for those hardest graphs did run for a week but did not remotely come close to completion at the end of that week, so they were cancelled.

The measurements obtained in the experiments are presented in diagrams with four subplots each. Every subplot subplot represents results for graphs with varying vertex count and one particular fixed density. That density can be found in the lower right corner of every subplot. To provide a more compact overview of the performance of the algorithms, the diagrams within this chapter focus on the graphs with the densities 0.001, 0.01, 0.1 and 0.5. The diagrams for the remaining densities can be found in the appendix at chapter 7.4. For the heuristic algorithms, directly below the diagrams with the runtimes, another diagram with the qualities is shown. When studying the diagrams, please note that the runtime axes (t) and vertex count axes (n) are always scaled logarithmically. Every data point in the plots represents the average of the successful runs (finished within the time limit) of a certain algorithm parametrization, vertex count and density. With three possible vertex weight distributions, three graphs generated for every combination and the three Raspberry Pi computers that processed all runs each, up to 27 measurements are incorporated into every data point. When the it is mentioned in the following sections that an algorithm “fails”, this usually means that it did not finish within the runtime limit.

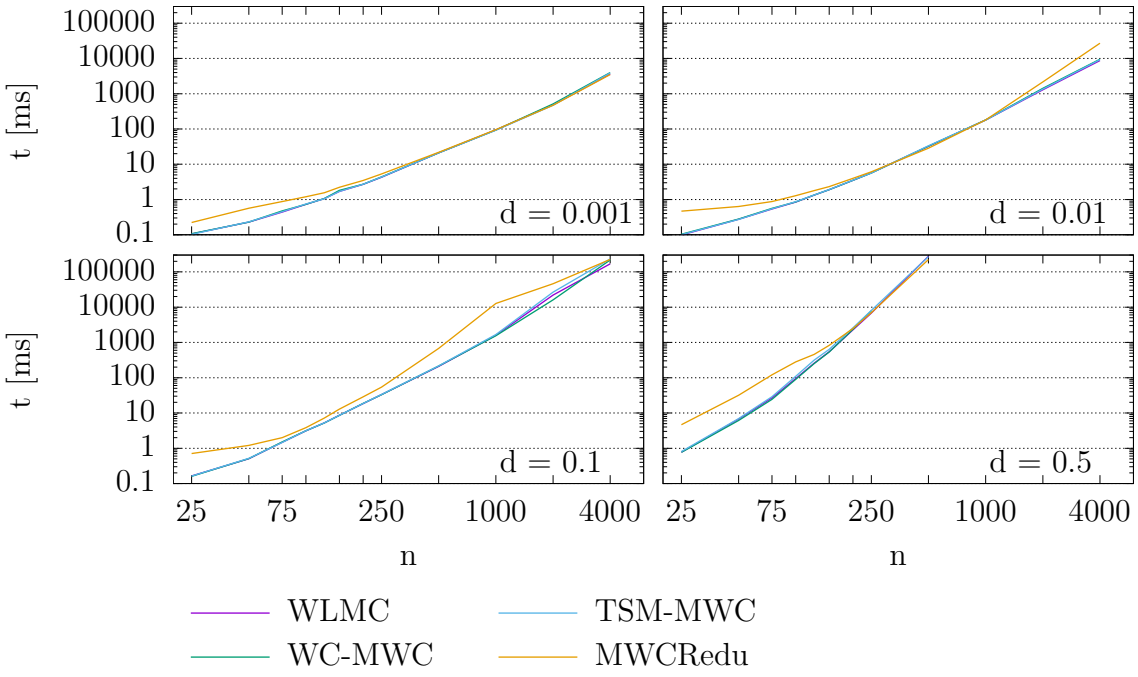

Figure 5.1: Runtimes of the exact algorithms

5.1 Exact algorithms

Overall, the runtimes of the different exact algorithms displayed in figure 5.1 follow a similar trend and show a pretty direct correlation to both vertex count and density. For graphs with $d \leq 0.05$ (see figure 7.1 for more densities), all exact algorithm runs finished within the time limit. For $d = 0.1$, WC-MWC met the time limit successfully except for one of 27 instances, while the other algorithms surpassed the limit 18/27 times. When looking at the graphs with the highest tested density, i.e., $d = 0.5$, no exact algorithm found a solution for graphs with 1000, 2000 or 4000 vertices in time. Even for $d = 500$, WC-MWC never finished early enough, the other algorithms failed at least half of the time. In most cases the runtimes are very similar, apparently the different strategies the algorithms use to minimize the amount of branches do not make that much of a difference on the tested graphs. The only algorithm that deviates a bit more from the others is MWCRedu. Its additional elaborate graph reduction techniques appear to have little success on most of the smaller and medium-sized graphs, but at the same time require a lot of runtime. This makes it actually slower than the other algorithms in those instances. However for the hardest graphs that were solved by any of the heuristic algorithms, it is about 20% faster than the others. For future research it could be interesting to investigate whether this advantage of MWCRedu potentially continues and maybe even grows with larger graphs and with less strict time limits.

When comparing the runtimes of the own implementations in figure 5.1 to the ones of

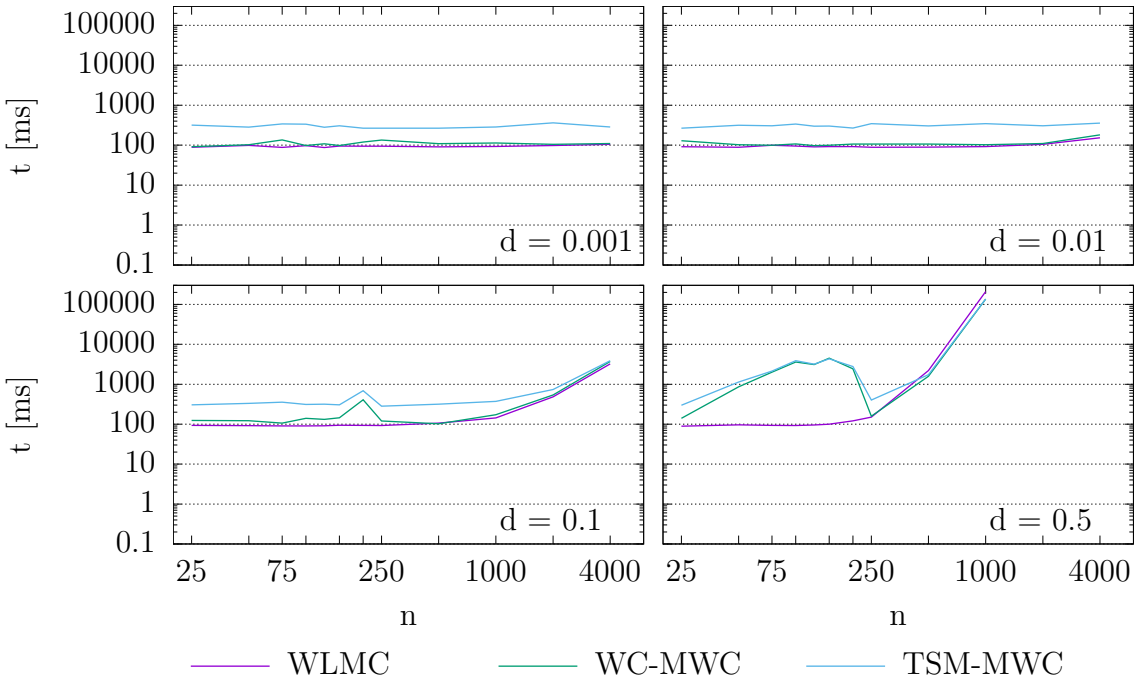

Figure 5.2: Runtimes of the reference implementations of the exact algorithms

the reference implementations shown in figure 5.2, it can be seen that for the lower densities the reference implementations have an almost constant runtime. Their runtimes only start to rise with the higher densities and vertex counts. While the constant offset makes them slower than the own implementations at the more simple graphs, they have a significant performance advantage on the bigger and more dense graphs. A possible reason for the offset could be that parts of the initialization procedures of the reference implementations allocate large amounts of memory of a constant size independent of the actual size of the graphs. It should be noted that the reference implementations themselves measure loading and solving time separately, unlike the test environment, which counts both into one runtime. The initialization procedures have been adapted so that they do not have to read the graphs from the disk, as the files are already loaded by the test environment. Some reference implementations also use their initialization procedure to collect useful properties that go beyond the minimum data required to represent the graphs, which commends counting the initialization time too. Another interesting occurrence are the runtime bumps of WC-MWC and TSM-MWC on the bottom left and especially on the bottom right plots of figure 5.2. Similar bumps cannot be found in the plots of the own implementations, leading to the presumption that this is an issue of implementations rather than the algorithmic concepts.

5.2 Heuristic algorithms

Most of the presented heuristic algorithms have parameters that can be tuned to control certain properties of the algorithms. In the following chapters, at first selected parametrizations of the algorithms are compared. In the end, an overview is presented which also draws comparisons across algorithms, to see which heuristic algorithm with which parametrization is suitable for certain graphs.

5.2.1 FastWClq

FastWClq was tested with cutoff times of 20 ms, 1 s and 60 s. The other parameters were fixed to the values recommended by the authors of [7]. The results are displayed in figure 5.3. FastWClq proves to have a comparatively very good runtime performance on the larger graphs. Not only did all runs succeed within the time limit, they also never ran significantly longer than the largest chosen cutoff time, i.e., 60 s. For the smaller and less dense graphs, the runtime rises with the density and vertex count, with the different parametrizations taking approximately the same time until they reach their cutoff time. The smaller and less dense graphs are apparently easier to reduce up to a point where FastWClq can prove that it found the optimal solutions, so that it can terminate early without approaching the cutoff time. At the point where the respective cutoff time is crossed the runs exhibit at a relatively constant runtime despite rising densities and numbers of vertices. At this plateau, FastWClq terminates the search for a clique without necessarily being sure that it has found the optimal solution. For the largest graphs, and especially the more dense of those, FastWClq tends to overshoot the cutoff times. This can be explained by the fact that FastWClq only checks whether it is within the cutoff time after it finished an iteration which consists of sampling a clique and potentially reducing the graph. Such an iteration naturally takes more time if there are more vertices and each vertex has a higher degree on average. That postpones the check for the cutoff time for a greater amount of time. The reference implementation shows a similar behavior, although not as strongly, as visible in figure 7.11.

For many graph instances, FastWClq manages to find the exact solution or a clique very close to it. At lower vertex counts, all parametrizations produce high quality solutions. When a certain vertex count is crossed, the quality begins to drop. The higher the density or the lower the cutoff time, the earlier the quality drop happens. This can easily be explained by the fact that FastWClq does not have enough time to sample a sufficient number of cliques, as the amount of possible cliques rises with the size and density of the graphs. Another observation is that the weight of the found clique declines to as low as 40% compared to the actual maximum weight clique for a low density of 0.001. In contrast to that, for the highest tested density 0.5, all parametrizations achieve a solution quality of at least 75%. There are some probable reasons for this. One factor is that in dense graphs, there are likely more cliques

with a weight that is close to the exact solution weight. This increases the chance that a clique which is at least close to the optimal solution is sampled. Another factor lies within the heuristic of FastWClq. To estimate the benefit of a candidate vertex for a clique, the weight of its neighborhood is taken into account. In dense graphs, on average more members of this neighborhood should be connected to the rest of the clique as well, which means that the neighborhood weight is a good metric for the benefit of a vertex. In less dense graphs, presumably only a smaller part of the neighborhood can be used in the finished clique, which makes the neighborhood weight a less reliable benefit indicator.

5.2.2 SCCWalk

The test runs for SCCWalk were conducted with the cutoff times 1 s and 60 s. Like Wang et al. recommended, $L = 4000$ and $maxUStep = 500$ are used for the runs of both cutoff times. Runs with a 20 ms cutoff time were not launched, as with the given L and $maxUStep$ parameters, even a single iteration of SCCWalk would violate this cutoff time for most of the graphs. Since SCCWalk has no early exit within the algorithm in contrary to FastWClq, it will not run for less than the cutoff time, like it can be seen in figure 5.4. From the smallest graph on, the runtime stays relatively constant with rising vertex numbers. Similar to the behavior of FastWClq, above a certain vertex count the runtime starts to exceed the cutoff time considerably, especially if the latter is only 1 s. This rise in the runtime already starts with smaller graphs compared to the measurements from FastWClq. Considering that SCCWalk evaluates the potential weight changes of all eligible members of the $AddSet$ and $SwapSet$ and potentially also of the $DropSet$ in every iteration, significant runtime increases are expectable, given those sets increase in size with larger graphs. With both cutoff times, all runs for graphs with $n = 4000$ and $d = 0.1$ fail to stay within the time limit of 5 minutes. Regarding graphs with $d = 0.5$, both those with 4000 and 2000 vertices cannot be processed in time at all, while those with 1000 vertices are successfully handled at least half of the time.

While the speed of SCCWalk is not the most remarkable, the quality of its solutions is very good. Up until a density of 0.1, the algorithm finds the optimal solution in most cases, in the other cases still at least a 99% quality is achieved on average. With the $d = 0.5$ graph instances, a slight decrease in quality is visible for larger graphs. However on average, the quality stays above 95% even for those harder instances. When observing the plots, it does become clear that the quality difference between the two parametrizations is negligible on the tested graphs. Hence choosing a cutoff time of 1 s over 60 s should be preferred in this context, as it results in the same quality while the runtimes for smaller graphs are shorter. A potential reason for this result is that the local search strategy of SCCWalk is already quite successful in finding the maximum weight clique during the first iterations. Due to that, the

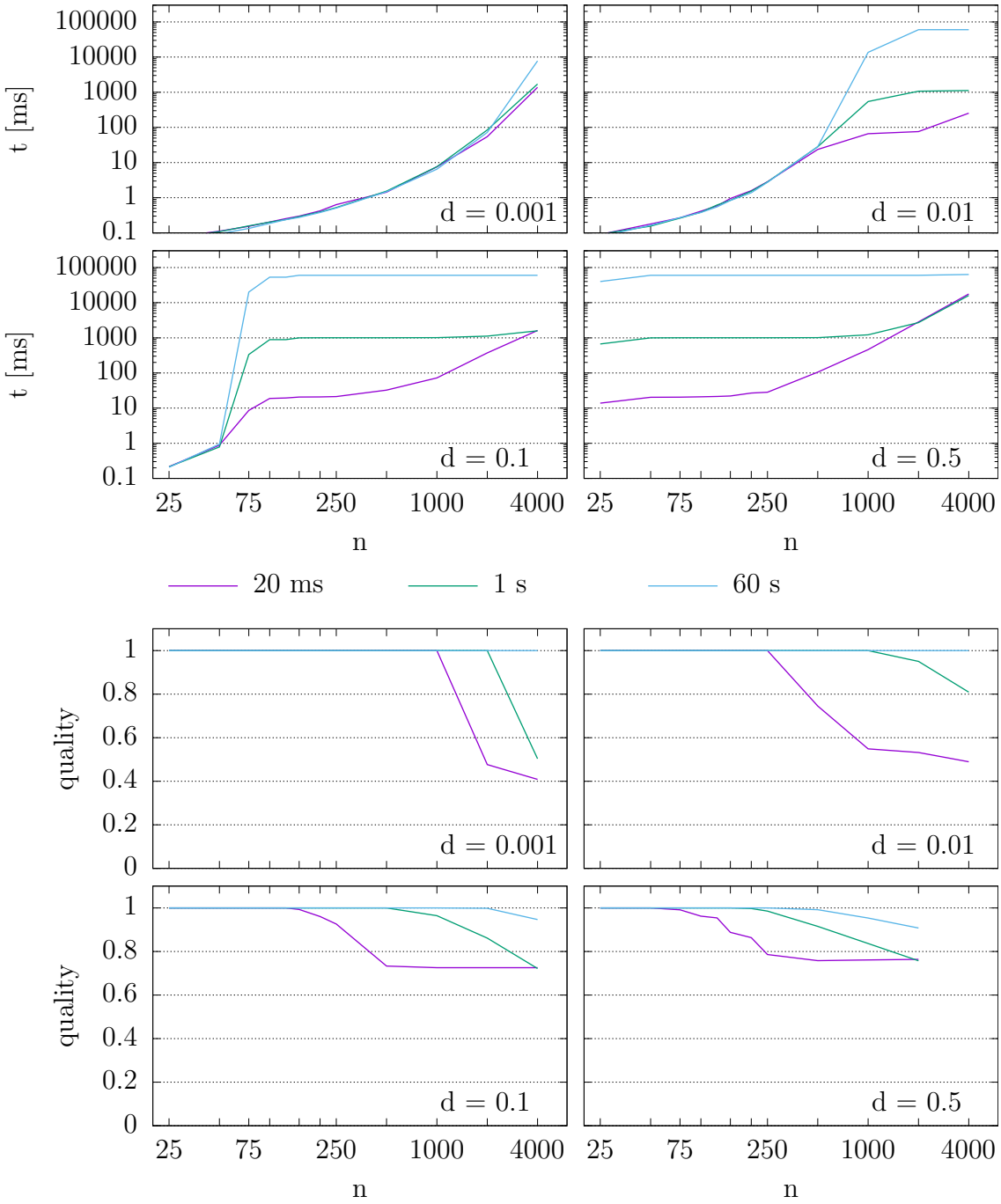

Figure 5.3: Measurements of FastWClq with different cutoff times

following iterations, which runs with a higher cutoff time have more of, can hardly or not at all find a better clique to improve the solution.

5.2.3 SCCWalk4L

For the SCCWalk4L tests, the same two cutoff times as for SCCWalk were used. However for the other parameters, that were not alternated between the runs, this time $maxUStep = 50$ and $L = 400$ are set, as the original paper [6] suggests. Because SCCWalk4L, just like FastWClq, uses both a graph reduction to make an early exit if possible and a cutoff time, the general shape of the runtime diagrams found in figure 5.5 resembles the ones of FastWClq. The SCCWalk4L plots do however overshoot the cutoff time more quickly, just as SCCWalk does. At the lower vertex counts the SCCWalk4L runtime manages to stay significantly under the cutoff time and below the one of SCCWalk. At a density of 0.5, that advantage is gone and the runtimes of SCCWalk4L and SCCWalk are relatively similar. SCCWalk4L also fails to meet the time limit on the same graph types as SCCWalk. The available reference implementation exhibits less of an overshooting effect and thus never violates the time limit as it can be verified in figure 7.11. The reason for that could be that the reference implementation checks whether it approaches the cutoff time at multiple locations within the algorithm, allowing it to follow it more strictly. The own implementation only does such a check once in each iteration, as the pseudo code listed in the original paper [6] suggests it.

Regarding the quality, there are no notable differences to SCCWalk either. Again the cutoff time has no clear effect on the resulting quality. Overall, the only differentiation point of SCCWalk4L from SCCWalk remains the lower runtime on smaller graphs. The solution quality of the reference implementation shows a less strong drop in the quality on larger graphs, which can probably be attributed to a more efficient implementation that is able to explore more cliques in a shorter time. One particularity that can be noticed is a sudden quality decrease of 8% with 1 s cutoff time, $d = 0.001$ and $n = 4000$. A deeper analysis of the data came to the conclusion that this is due to some outliers. Most runs of that graph configuration actually finish with about 100% quality. The average is dragged down by two runs on a graph with a normal weight distribution with 69% and 53% quality and one run on a graph with uniform vertex weight distribution that reached 55% quality. Together with the fact that not all runs on those problematic graphs have a sub-par quality, this serves as a reminder that algorithms containing random elements always bear a certain risk of unexpected and unwanted results.

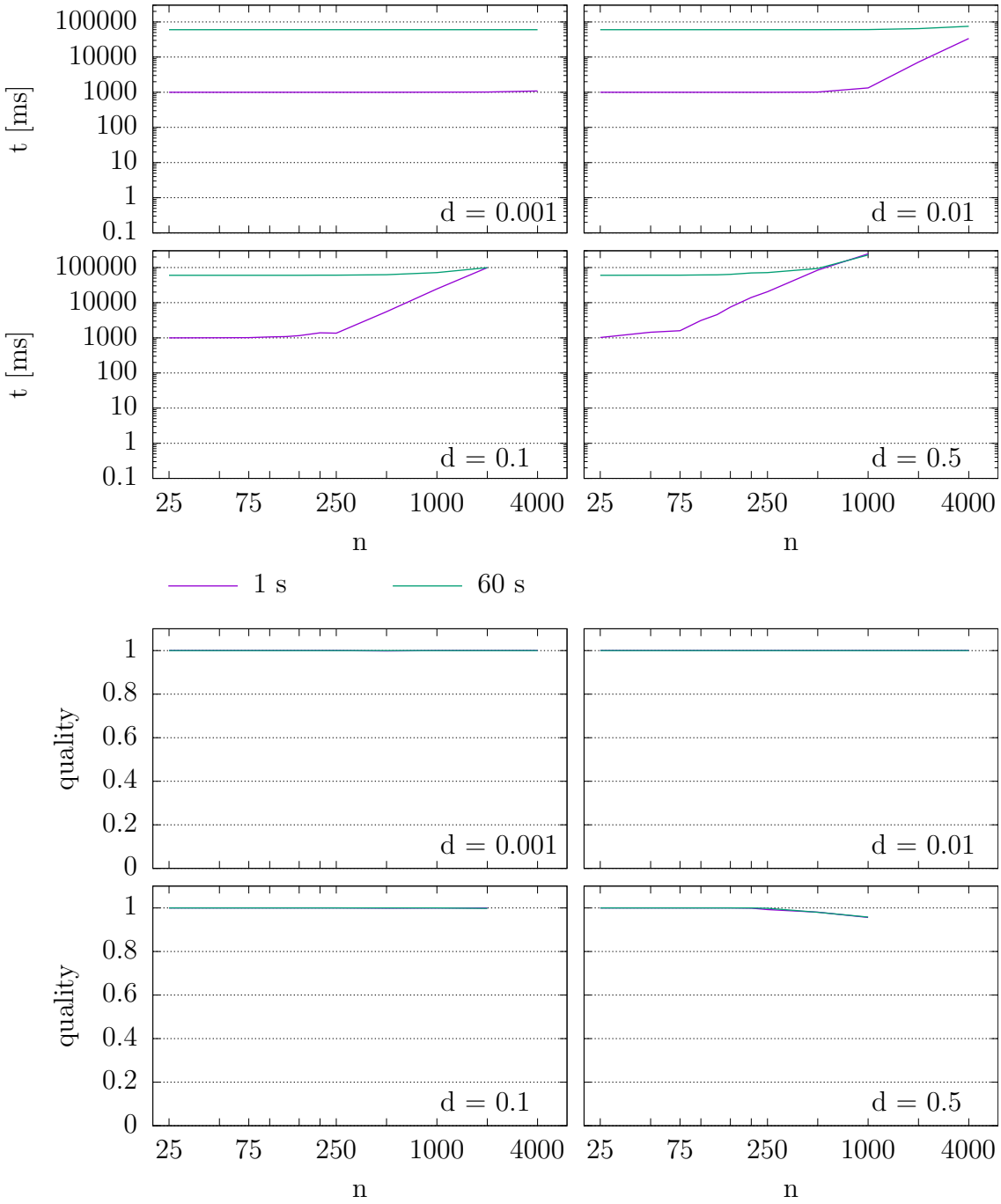

Figure 5.4: Measurements of SCCWalk with different cutoff times

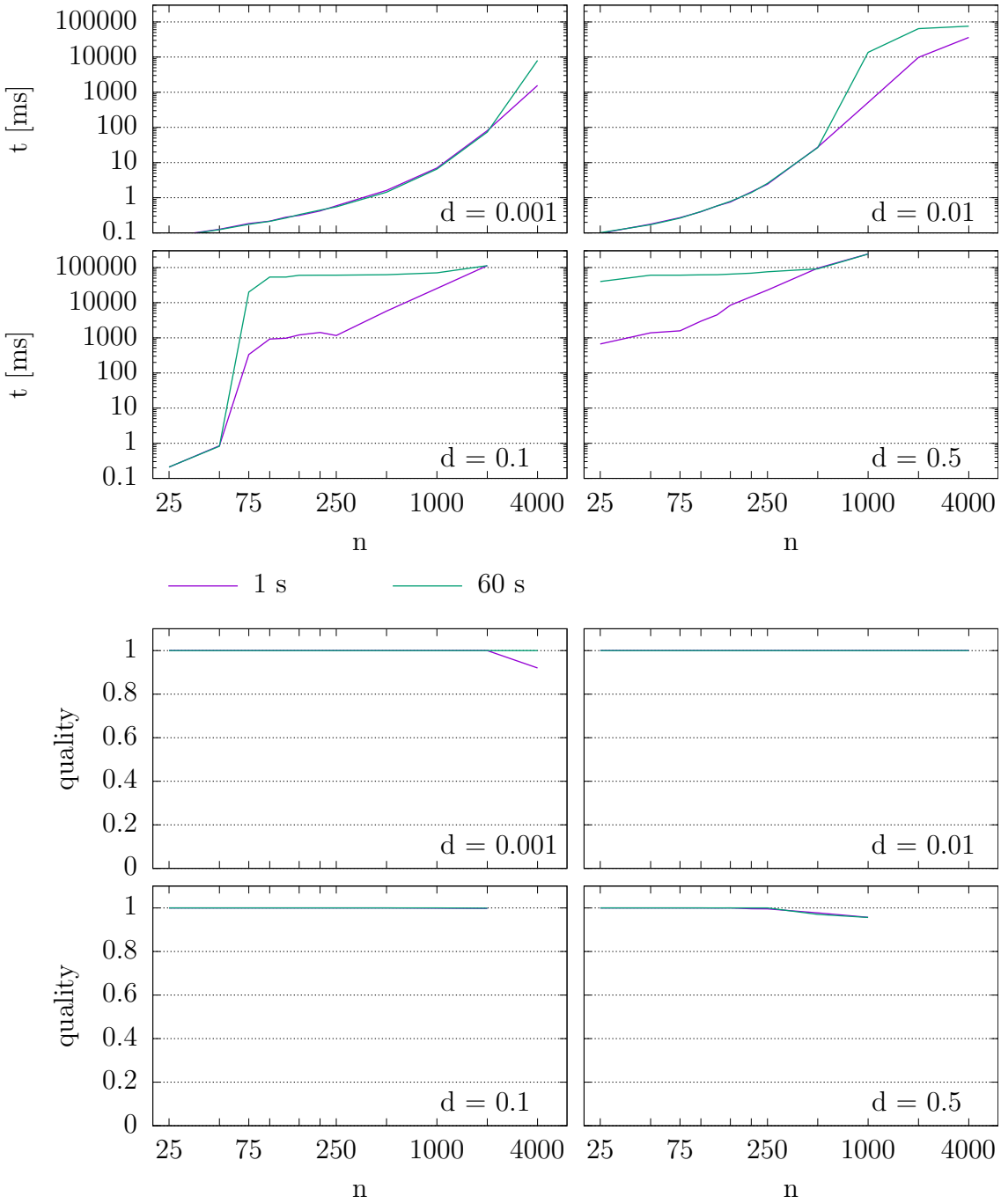

Figure 5.5: Measurements of SCCWalk4L with different cutoff times

5.2.4 MWCPeel

For MWCPeel, whose performance is plotted in figure 5.6, only one configuration was tested, as the corresponding paper [17] did not explicitly mention parameters that should be tuned. The runtime graphs tend to proceed relatively steep compared to other algorithms without a cutoff time. MWCPeel starts to fail to compute its solution within the time limit at smaller graphs than the other heuristic algorithms. For graphs with $d = 0.1$, no runs terminate in time for $n = 4000$, only one run succeeds for $n = 2000$ and also for $n = 1000$ two runs fail. At $d = 0.5$, all runs on graphs with 1000 or more vertices fail. Furthermore 22/27 runs for $n = 500$ and even some single runs for graphs with 200 and 250 vertices violate the time limit. A possible explanation is that the algorithm is engineered for larger and more sparse graphs and that the employed reduction strategies have too much overhead while not achieving significant reductions on the graph instances that were tested within the thesis. If the reduction and peeling strategies fail to reduce the graph considerably, MWCPeel as a hybrid algorithm will have the combined runtime of its own preprocessing steps and TSM-MWC.

For the cases where MWCPeel produces a solution in time, its quality is quite good. It does not undercut 94%, most of the time the quality is even at 100%. As explained in chapter 3.2.4, the process of MWCPeel consists of two ain steps. The first one is a heuristic graph reduction, which in the best case does not affect the vertices of the maximum weight clique. If the latter is the case, the second stage, which uses the exact TSM-MWC algorithm, will also find the exact solution. This best case seems to occur often on the example graphs. In the suboptimal case where too many vertices would get removed by the heuristic reduction, MWCPeel also employs a safeguard which stops the graph from being reduced too much, which in turn explains why the quality does not drop dramatically.

5.2.5 UEW

For UEW, the own weight priority parameter α was tested with the values 1, 2 and 4. It can be seen in figure 5.7 that the runtime is not significantly affected by the parameter. The lines representing the different parametrizations are almost indistinguishable. The runtime increases quite evenly with growing vertex count and density. The runtimes on graphs with a small density are the best ones among all the algorithms. The relative increase in runtime with higher densities is rather strong, in the graphs with the combination of the highest densities and highest vertex counts the runtime approaches the approximately half of the time limit. None of the runs failed to complete in time. Most algorithms do not manage to achieve the latter, with the exception of FastWClq. While FastWClq is slower on the smaller and less dense graphs, it is up to ten times faster than UEW on the hardest graphs of the experiments.

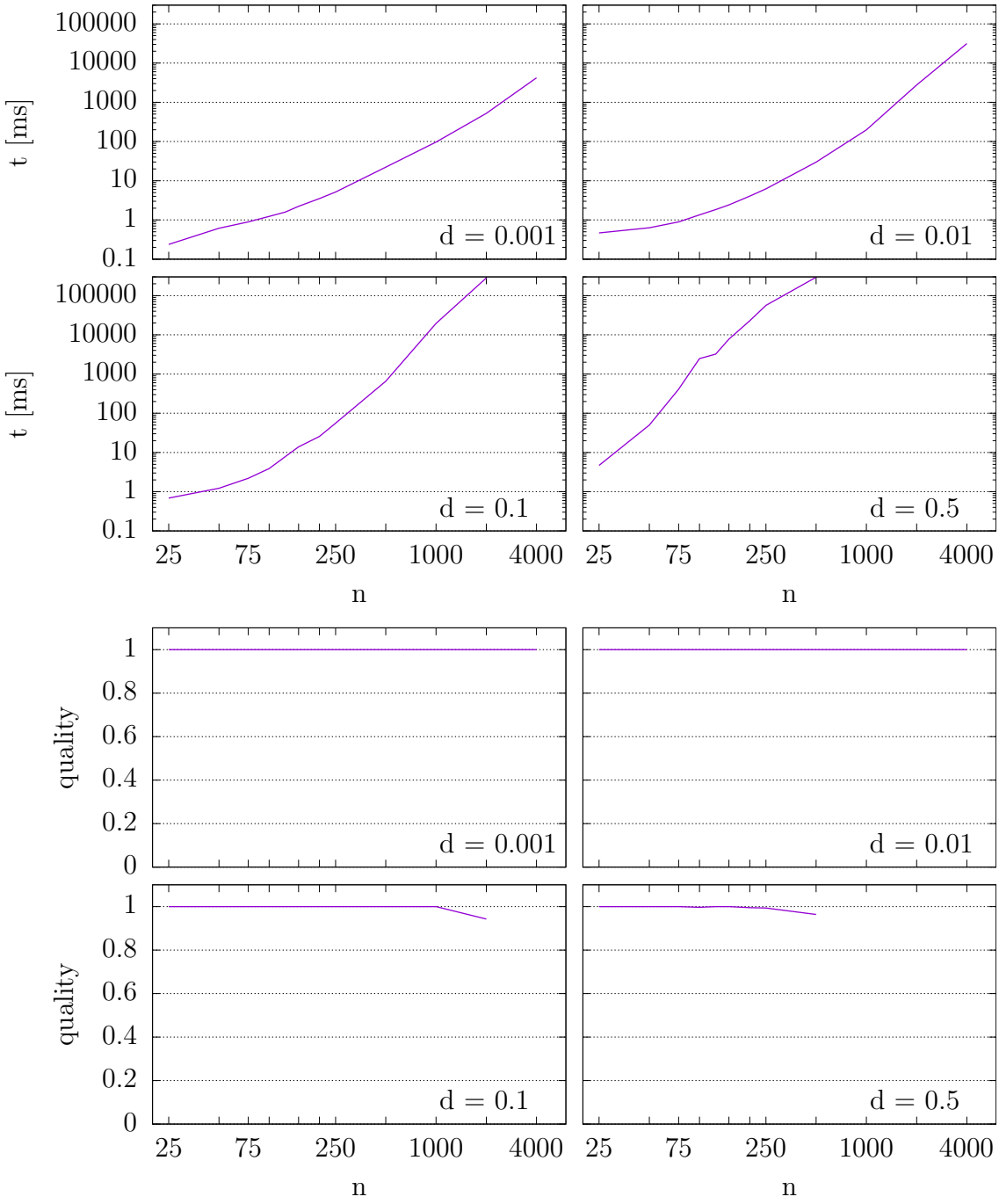

Figure 5.6: Measurements of MWCPeel

In the graphs with smaller densities, there does not seem to be a single value for the own weight priority α that has a clear advantage with regards to the quality of the solutions. With the highest tested density 0.5 however, a parametrization of $\alpha = 4$ shows a consistent albeit small quality benefit over the other ones. The plots for $d = 0.2$ and $d = 0.3$ in figure 7.7 confirm the observation that $\alpha = 4$ is a suitable parametrization for more dense graphs. As 4 was the highest tested parameter value here and a quality proportional to α seems to be present for the dense graphs, higher own weight priorities would be worth testing in future work.

In general the quality always stays above 0.8, but rarely reaches the optimal value of 1.0. While other heuristic algorithms exhibit a better quality in many scenarios, it is notable that the quality of UEW shows no clear dependency on the number of vertices of the graph.

5.2.6 UEW-R

Similar to UEW, UEW-R was tested with the own weight priorities 1, 2, and 4. The graphs in figure 5.8 reveal that again the runtime is not influenced by the value chosen for α, but steadily raising with the vertex count and density. UEW-R fails to timely compute solutions for the biggest graphs with $n = 4000$ and $d = 0.5$ most of the time with all parametrizations, i.e., 22 or 23 times out of 27. The runtimes are consistently and considerably higher than the ones of UEW. This was expected since UEW-R performs the same steps as UEW but (depending on the graph) repeats those steps and additionally conducts graph reductions. On lower-density-graphs, those additional iterations and reductions pay off and provide solutions of higher quality, often even the exact solution is found. For the more dense graphs, the reduction strategy does not seem to be effective and the resulting quality is similar to the one of the original UEW algorithm.

5.2.7 Comparison

With the help of figure 5.9, this section summarizes the key takeaways regarding the performance of the heuristic algorithms and draw some comparisons between them. The measurements of FastWClq, SCCWalk and SCCWalk4L that can be seen in the plots are all based on a cutoff time of 1 s. This value was found to achieve a good balance between speed and quality for all algorithms. Additionally using the same cutoff time for the algorithms improves the meaningfulness of the comparison. The own weight priority α for UEW and UEW-R was set to 4 in the comparison graph, as this proved to deliver a good quality in many cases.

The runtime graphs in the upper part of figure 5.9 show that for lower densities, UEW is significantly faster than its competitors, while at the larger graphs with higher densities, it gets overtaken by FastWClq. SCCWalk is on the slower side for

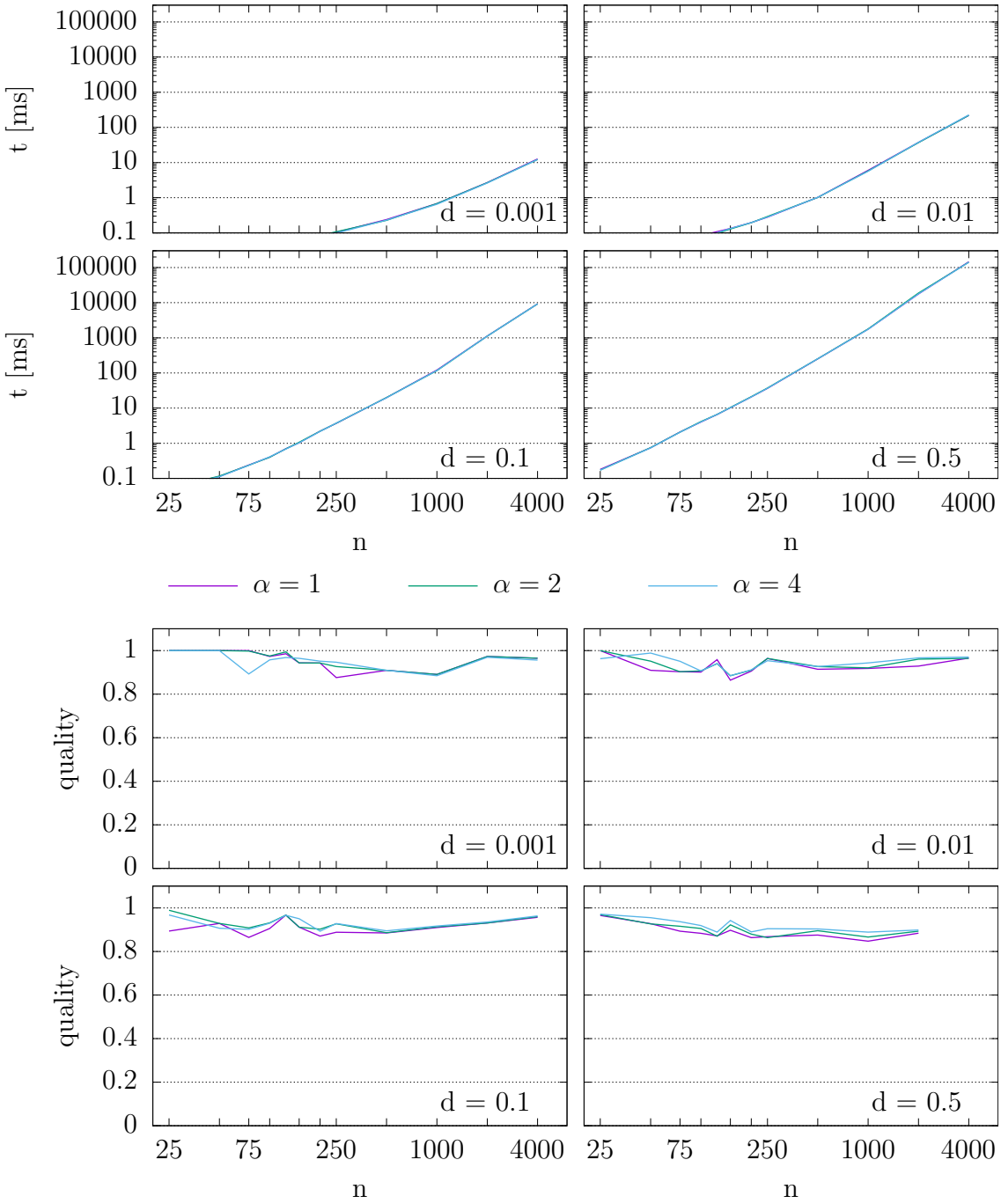

Figure 5.7: Measurements of UEW with different own weight priorities

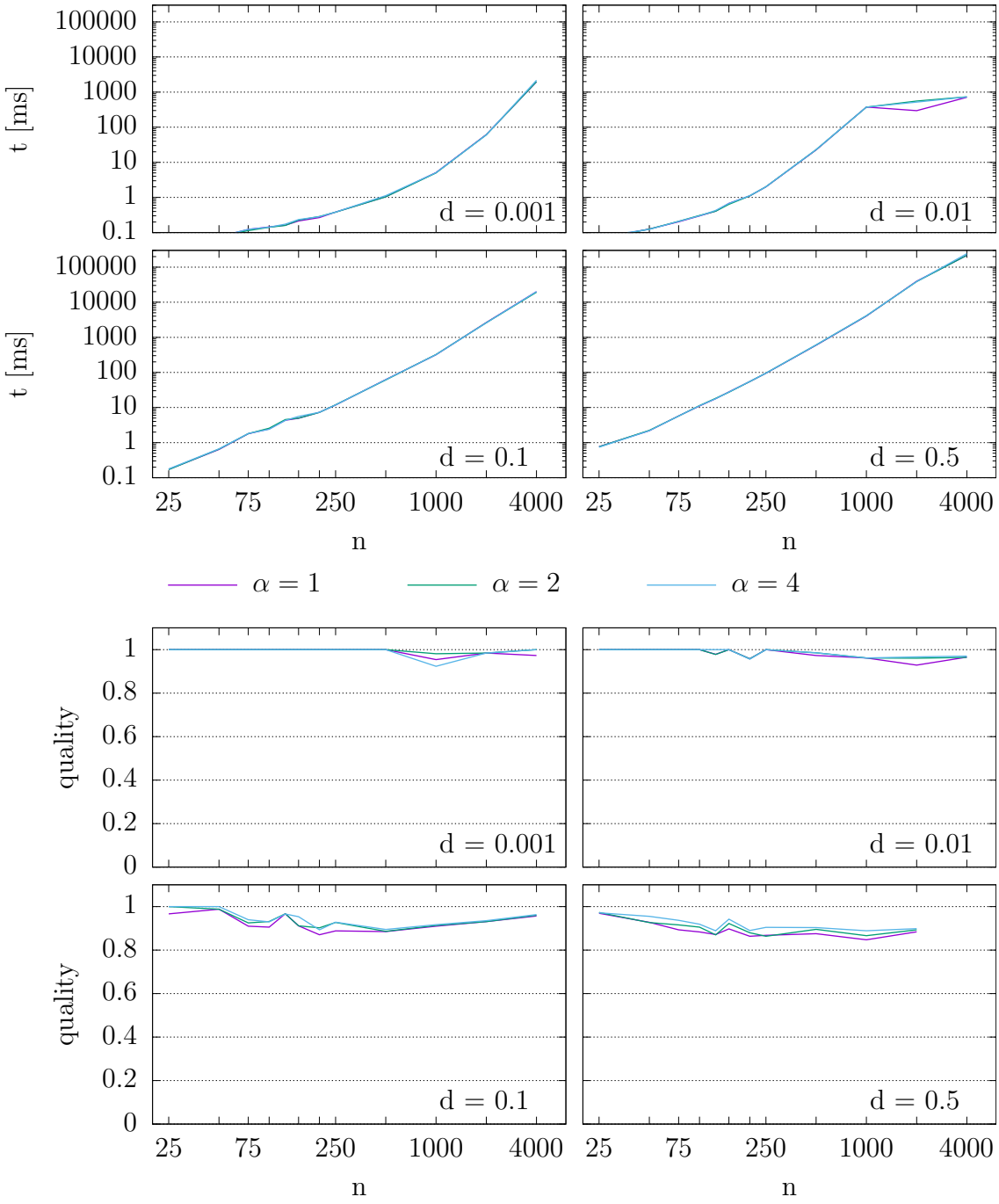

Figure 5.8: Measurements of UEW-R with different own weight priorities

most graphs, which is expected due to it not being able to terminate before its cutoff time. SCCWalk4L is in the middle field for smaller and more sparse graphs. At the larger and more dense graphs, it joins the slower pace of SCCWalk. MWCPeel has quite mixed runtime results in the different scenarios. It is always slower than UEW and UEW-R. On very sparse graphs, sparse graphs with few vertices and dense graphs with more vertices, it is noticeably slower than FastWClq, SCCWalk and SCCWalk4L. On the other types of graphs, MWCRedu does have an advantage over the aforementioned algorithms. Those patterns are also visible in figure 7.9 on the graphs with the other densities.

The qualities of the solution cliques delivered by the heuristic algorithms are plotted in the lower part of figure 5.9. At the graphs with low to medium vertex counts, UEW visibly has the lowest qualities , while for the highest tested vertex counts FastWClq clearly produces the lowest-quality cliques. At lower densities and smaller graphs, UEW-R exhibits a quality that is significantly better than UEW and often even optimal. At the more dense graphs, the qualities of UEW and UEW-R tend to be the same. FastWClq, SCCWalk, SCCWalk4L and MWCRedu demonstrate excellent qualities on the smaller and middle-sized graphs. On the graphs with the most vertices, their qualities do decline. This decline is the strongest for FastWClq, of medium intensity for MWCRedu. SCCWalk and SCCWalk4L have a rather slight quality decline that is only present at the most dense graphs. A positive general observation is that in the most cases the heuristic algorithms deliver a quality that is comfortably above 80% for the tested data.

As an additional overview, figures 5.10 and 7.10 present a runtime comparison between the heuristic algorithms (except UEW-R) and the exact algorithm TSM-MWC. The parameters used in there for the heuristic algorithms remained the same as in the previous comparison. In most instances, TSM-MWC is faster than MWCPeel, sometimes by a large amount. Again, just like MWCRedu, MWCPeel uses TSM-MWC internally together with some novel graph reduction techniques. Towards the larger graphs with $d = 0.5$, the plot lines of both algorithms meet just before the runtime limit. This leads to the assumption that MWCPeel could potentially turn out to outperform TSM-MWC on larger graphs if the time limit was higher. With the data at hand however, similarly to the observations with MWCRedu, the additional graph reduction strategies do not seem to be effective enough on the given graphs to justify their additional runtime. TSM-MWC is slower than UEW in any case, and slower than FastWClq on larger graphs. At medium densities, TSM-MWC tends to be faster than SCCWalk and SCCWalk4L. On smaller graphs, TSM-MWC is also obviously faster than heuristic algorithms with cutoff times where they do not find an early exit. Since TSM-MWC performed quite similar to WLMC and WC-MWC, the former statements should also apply to the those other exact algorithms as well. As a rule of thumb, exact algorithms might generally be preferred on smaller graphs, as they promise to find the best solution in sometimes shorter amounts of time. Heuristic algorithms on the other hand could be

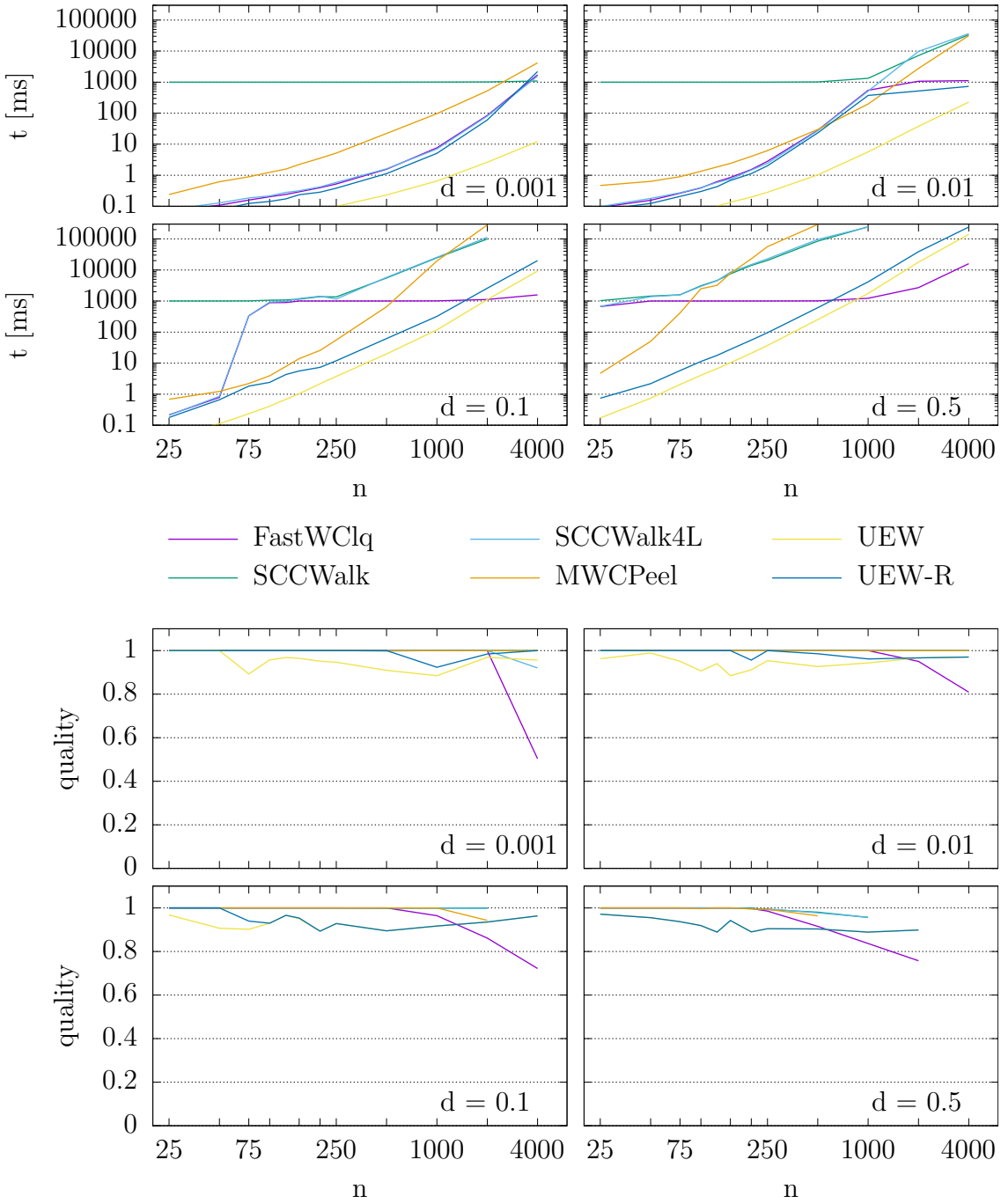

Figure 5.9: Measurements of the heuristic algorithms

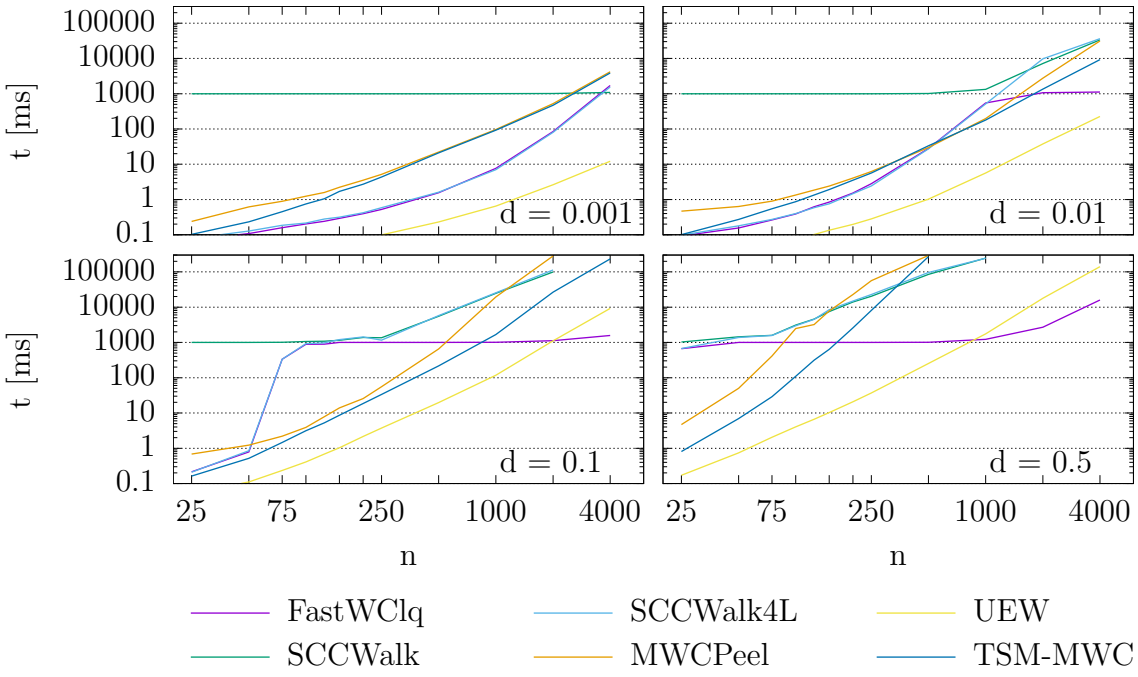

Figure 5.10: Measurements of the heuristic algorithms compared to TSM-MWC

more suitable for larger graphs where the runtime of the exact algorithms becomes prohibitively high.

5.3 Iterative runs on similar graphs

Overall, the measurements suggest that the versions of the algorithms adapted for iterative runs do not have a clear universal advantage compared to the original algorithms. For several algorithms and parametrizations the advantages in runtime and quality are either quite small or non-existent, thus most of the respective diagrams are only listed in the appendix in section 7.4.3. Visible differences do however occur in more specific scenarios, which will be highlighted in this chapter. In the charts, the data series with the label "non-iterative" represents the measurements of the unmodified algorithms on the standalone run graphs. Those results were already presented and discussed in the previous chapters. The subject of interest here are the runs in which the iterative versions of the algorithms were used, with the adaptions described in chapter 4.1.3. Like explained in chapter 4.2.2, in a first step, a graph similar to the standalone one, but with modifications made with a 2% chance each, was processed by the algorithms. Using the results from that, the iterative algorithms processed the unmodified standalone graphs. The results from that second run are represented in the data series "iterative (2% change)". Similarly, the data series "iterative (5% change)" represent the second runs with iterative algo-

rithm versions from experiments where the graphs with 5% change probability were processed in the first step.

5.3.1 Findings

For all of the exact algorithms, the iterative versions were either similarly fast or even slower than their non-iterative counterparts. Seemingly the runtime overhead that the execution of the *ConvertInitialClique* function (algorithm 26) brings is not outweighed by the benefits of the potentially better initial clique it produces. $\hat{C}'_0$ might either be not better than the initial cliques that the algorithms produce in their own initialization procedures, or at least not suitable to reduce the search space more than those.

Regarding the heuristic algorithms, a notable case is FastWClq with a cutoff time of 20 ms and to a certain extent also with 1 s. As evident in figure 5.11 (and figure 7.22), the quality drops on graphs with a high vertex count and low density are less strong when the the iterative version of FastWClq is used. On the flip side, for both FastWClq and SCCWalk4L, the iterative versions take considerably more runtime in the situations where the non-iterative version is able to finish well before the cutoff time is reached. Yet the runtime increase does at least not lead to a violation of the cutoff time. In contrast to this are the situations observable in figure 5.12, where SCCWalk4L with a cutoff time of 60 s does not perform an early exit before reaching its cutoff time. There the iterative version is faster than the original one by up to a third (note the logarithmic scale), considering the case with 2% change probability between the previous and current graph. Furthermore, some slight quality improvements with UEW-R adapted for iterative runs can be seen in figures 7.35 to 7.40. Those improvements are however mostly below 5% and show no clear pattern.

5.3.2 Further potential

In general, the results obtained from the adapted iterative version of the algorithms reveal less benefit than expected. A potential reason could be that the strategy proposed in chapter 4.1.3 is not generally suitable for iterative runs on similar graphs. Another possibility is that the strategy is a valid approach, but that the similar graphs G_0 used in the tests are already too different from the original graphs G_1. This in turn leads to two possible tasks for future work, which are outlined in figure 5.13. At first, it could be investigated how different the maximum weight clique on the similar graph $\hat{C}_0$, its converted version $\hat{C}'_0$ and the maximum weight clique on the original graph $\hat{C}_1$ are for the various graph instances. If it turns out that those differences are too distinct, this indicates that the second task should be conducted. The latter consists of executing and analyzing test runs with more similar graphs.

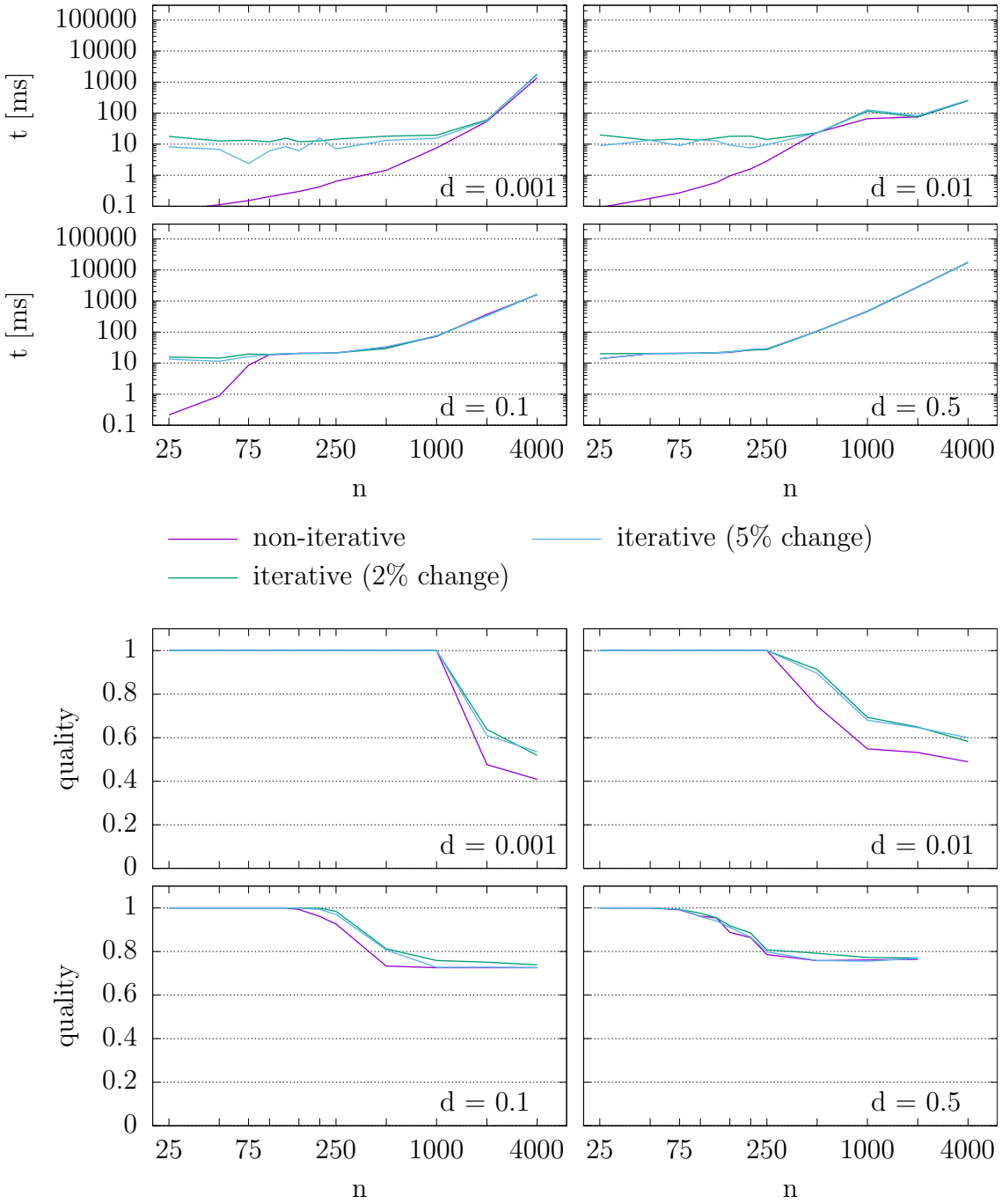

Figure 5.11: Measurements of iterative runs of FastWClq (20 ms cutoff time)

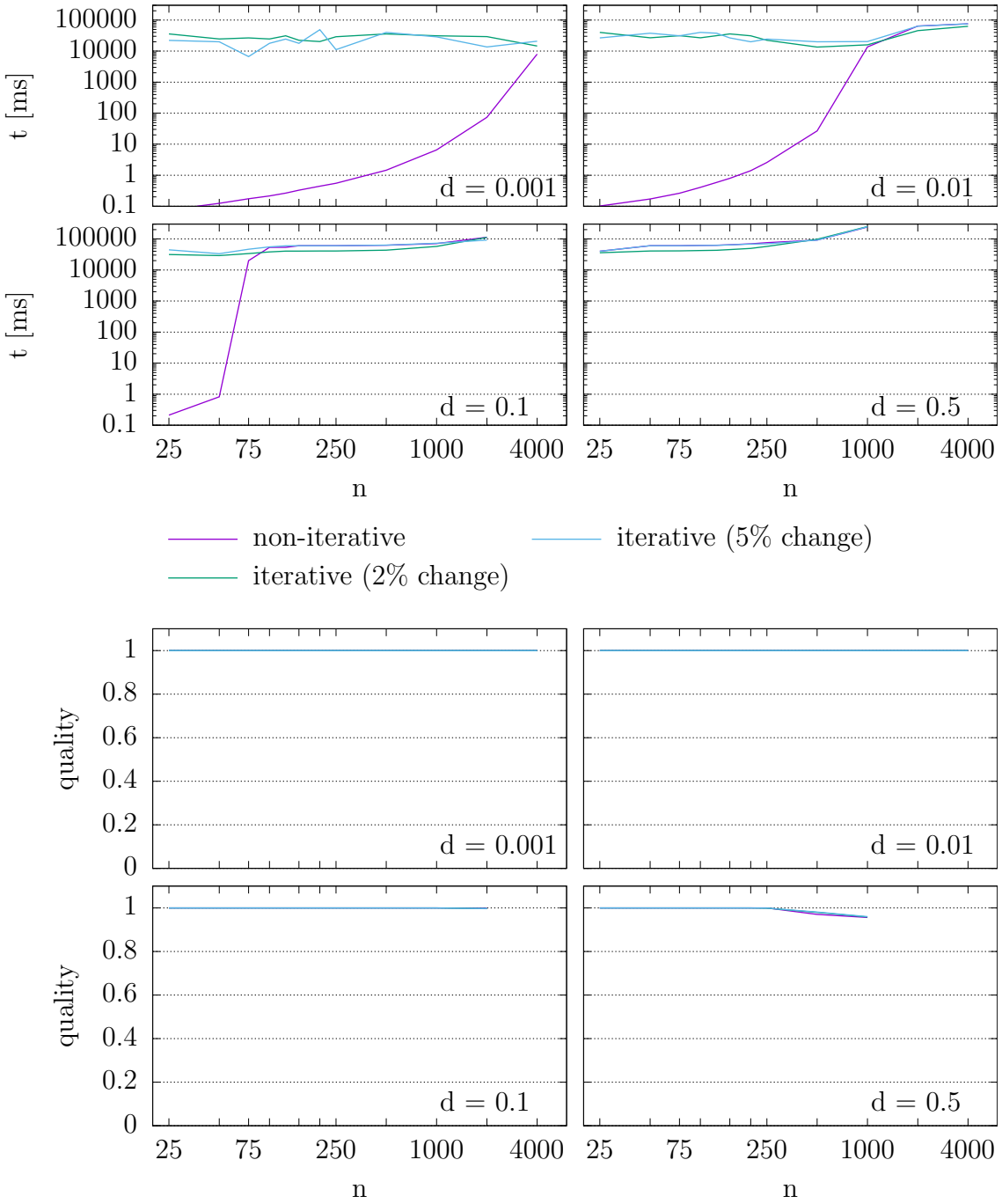

Figure 5.12: Measurements of iterative runs of SCCWalk4L (60 s cutoff time)

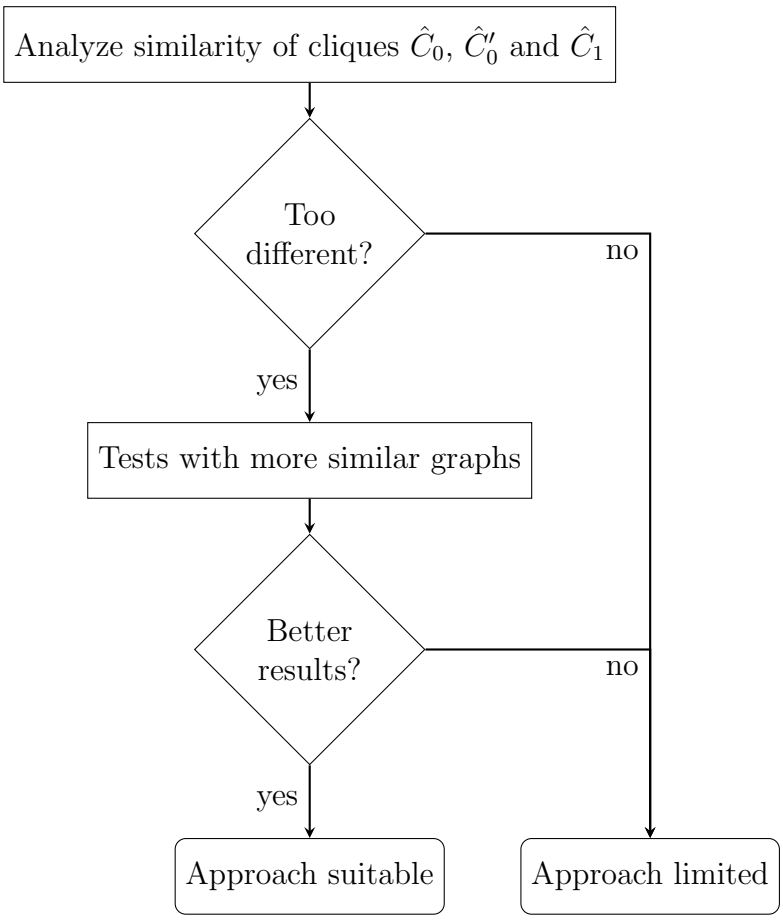

Figure 5.13: Approach for further research on iterative runs

Within that, many scenarios can be examined. For example the different types of change operations on the original graph (remove vertices, add vertices, remove edges, add edges, change weights) can be done isolated from each other to produce many similar graphs. This would allow to obtain more detailed insights on the individual impacts of the change operations. Also generating similar graphs with overall lower change probabilities would be interesting. If those tests on additional similar graphs reveal further advantages, the proposed iterative approach could be considered more widely applicable. Otherwise its use is confirmed to be limited to the specific scenarios described in the previous section 5.3.1.

5.4 Recommendations

After the previous chapters extensively presented and analyzed the results of the test runs, the following ones should provide a compact overview over which algorithms and parametrizations are suitable for which types of the tested graphs. To

support this, special tables dubbed "recommendation matrices" are used. In those recommendation matrices, the horizontal headers contain the graph densities and the vertical headers contain the vertex counts. The cells contain the recommended algorithm (according to a certain criterion) for dealing with graphs of the respective combination of vertex count and density. Recommendations for the same algorithm are colored in the same way to aid the identification of potential patterns.

5.4.1 Exact algorithms

The recommendation matrix for the exact algorithms is depicted in table 5.1. The recommended algorithms were selected by the lowest average runtime. The empty cells at the lower right corner represent graph instances for which no algorithm finished within the time limit. Hence no definite guide can be provided for those graphs. Since the runtimes of the exact algorithms were often quite similar, those recommendations are not that strong and using different algorithms will usually not have a too significant effect. In general, WLMC seems to be a good choice for many graphs with small to medium densities, whereas WC-MWC is more appropriate for higher densities. Choosing MWCRedu might be indicated on some larger graphs, while there is no clear pattern for TSM-MWC.

The recommendation matrix for the reference implementations of the exact algorithms (table 7.1) shows an even greater preference for WLMC throughout the graphs. But however for the combinations of the highest densities and vertex counts, other algorithms appear more frequently as the best choice. As that observation in the lower right corners of the recommendation matrices can be made for both the own and reference implementations, this reinforces that the choice of WC-MWC, TSM-MWC or MWCRedu is more advisable for those larger graphs.

5.4.2 Heuristic algorithms

In the recommendation matrices for the heuristic algorithms, some of the names had to be abbreviated to fit inside their cells. FastWClq is abbreviated to "FWC", SCCWalk to "SCC", SCCWalk4L to "S4L" and UEW-R to "U-R". The parameters for which the recommendations apply are written below the (abbreviated) name in smaller letters. Different parametrizations of the same algorithms use different shades of the same color as their cell background.

Runtime

If a short average runtime is the highest priority for a given application, UEW will be the best choice if the graphs have a small to medium vertex count and density.

n \ d	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5
25	TSM-MWC	TSM-MWC	WL MC	WC-MWC	WC-MWC	TSM-MWC	WL MC	WC-MWC
50	WC-MWC	WL MC	TSM-MWC	WL MC	WL MC	WL MC	WL MC	WC-MWC
75	WL MC	WL MC	WL MC	WL MC	TSM-MWC	WC-MWC	WC-MWC	WC-MWC
100	WC-MWC	WL MC	WL MC	WL MC	WL MC	WL MC	WC-MWC	WC-MWC
125	TSM-MWC	WL MC	WL MC	WL MC	WC-MWC	WL MC	WC-MWC	WC-MWC
150	WL MC	TSM-MWC	WC-MWC	WL MC	WC-MWC	WC-MWC	WC-MWC	WC-MWC
200	WL MC	WC-MWC	WL MC	TSM-MWC	WL MC	WC-MWC	WC-MWC	WL MC
250	WL MC	WL MC	WC-MWC	TSM-MWC	WC-MWC	WC-MWC	WC-MWC	WL MC
500	WL MC	WL MC	MWC Redu	WL MC	WL MC	WC-MWC	WC-MWC	MWC Redu
1000	TSM-MWC	MWC Redu	WL MC	TSM-MWC	WC-MWC	WC-MWC	TSM-MWC	-
2000	TSM-MWC	MWC Redu	WL MC	WL MC	WC-MWC	WC-MWC	-	-
4000	MWC Redu	WL MC	WL MC	WL MC	WL MC	-	-	-

Table 5.1: Exact algorithms with the best runtime

n \ d	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5
25	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 4$
50	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$
75	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 4$
100	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 1$	UEW $\alpha = 1$	UEW $\alpha = 1$
125	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 1$
150	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 1$
200	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 1$
250	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 1$	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 4$	FWC 0.02 s
500	UEW $\alpha = 2$	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 4$	FWC 0.02 s	FWC 0.02 s	FWC 0.02 s
1000	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 4$	FWC 0.02 s	FWC 0.02 s	FWC 0.02 s	FWC 0.02 s	FWC 0.02 s
2000	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 2$	FWC 0.02 s	FWC 0.02 s	FWC 0.02 s	FWC 1 s	FWC 1 s
4000	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 1$	FWC 0.02 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s

Table 5.2: Heuristic algorithms with the best runtime

For the larger and more dense graphs, FastWClq should be selected as per the recommendation matrix shown in table 5.2. The pattern in the matrix is quite clear this time, which strengthens its recommendations.

UEW was not part of the reference implementation comparisons, table 7.2 reveals that instead SCCWalk4L takes its place as the fastest reference algorithm for smaller and more sparse graphs. Meanwhile FastWClq remains advisable for large dense graphs but also gains traction in the middle diagonal, where the number of edges is in a medium range.

Quality

Besides the runtime, heuristic algorithms can also be judged by their quality. A recommendation matrix visualizing the algorithm parametrizations with the best

n \ d	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5
25	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 4$	U-R $\alpha = 2$	MWC Peel	MWC Peel	MWC Peel
50	UEW $\alpha = 1$	U-R $\alpha = 4$	U-R $\alpha = 4$	U-R $\alpha = 1$	U-R $\alpha = 4$	MWC Peel	MWC Peel	FWC 0.02 s
75	UEW $\alpha = 1$	U-R $\alpha = 1$	U-R $\alpha = 1$	U-R $\alpha = 1$	MWC Peel	MWC Peel	MWC Peel	MWC Peel
100	U-R $\alpha = 2$	U-R $\alpha = 2$	U-R $\alpha = 1$	MWC Peel	MWC Peel	MWC Peel	FWC 1 s	FWC 1 s
125	U-R $\alpha = 2$	U-R $\alpha = 4$	U-R $\alpha = 4$	U-R $\alpha = 4$	MWC Peel	MWC Peel	FWC 1 s	MWC Peel
150	U-R $\alpha = 1$	U-R $\alpha = 2$	U-R $\alpha = 2$	MWC Peel	MWC Peel	MWC Peel	MWC Peel	SCC 1 s
200	U-R $\alpha = 1$	S4L 60 s	FWC 60 s	MWC Peel	MWC Peel	FWC 1 s	FWC 1 s	FWC 60 s
250	U-R $\alpha = 2$	S4L 1 s	U-R $\alpha = 1$	MWC Peel	MWC Peel	FWC 1 s	FWC 60 s	FWC 60 s
500	U-R $\alpha = 2$	U-R $\alpha = 4$	S4L 60 s	MWC Peel	MWC Peel	S4L 1 s	SCC 60 s	FWC 60 s
1000	FWC 60 s	S4L 60 s	MWC Peel	MWC Peel	FWC 60 s	FWC 60 s	FWC 60 s	SCC 60 s
2000	FWC 60 s	MWC Peel	MWC Peel	MWC Peel	SCC 1 s	SCC 60 s	FWC 60 s	FWC 60 s
4000	SCC 1 s	SCC 1 s	MWC Peel	MWC Peel	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$	-

Table 5.3: Heuristic algorithms with the best quality

quality is found in table 5.3. The first criterion for the recommendations here was the average achieved solution quality. When two parametrizations have the same average quality, the one with more runs completed within the time limit is chosen. Should there still be a tie situation, the algorithm parametrization with the lower average runtime is preferred. Because for many graph types several algorithms delivered 100% quality, the aforementioned secondary criteria will often be crucial to the decision. The cell corresponding to graphs with $n = 4000$ and $d = 0.5$ does not contain a recommendation because no ground truth could be calculated for those graphs in a reasonable amount of time.

For very small and sparse graphs, UEW is still a good choice when quality is important, while for a little larger graphs, UEW-R delivers the best results. For input data on the upward middle diagonal, i.e., from the large sparse graphs to the small dense graphs, MWCPeel should be chosen. For the larger and more dense graphs,

FastWClq performs the best, while for the largest dense graphs UEW is recommended again. The latter is largely influenced by the fact most algorithms did not process those graphs in time, besides FastWClq, which exhibits a subpar quality on them.

Taking a look at the best choices for the reference implementations regarding quality, the pattern in the corresponding matrix (table 7.3) is mostly similar to the runtime one, but now SCCWalk4L also is the best choice on the large and more dense graphs.

Quality to runtime ratio

Ultimately, when making a decision about which heuristic algorithms to use, the runtime and quality might both be of importance. For this reason another set of recommendations is made in table 5.4. The criterion to determine the "winner" there is the quality achieved per runtime, i.e., $\frac{q}{t}$. There is again no well-founded recommendation available for $n = 4000$ and $d = 0.5$ due to the missing ground truth.

The results regarding which algorithm to choose are basically the same as in table 5.2. However, there are differences in the recommended parameters. Setting the own weight priority to $\alpha = 4$ is now more advisable than $\alpha = 1$. The reason becomes more clear when revisiting chapter 5.2.5. The runtime differences between the parameter settings are small and rather arbitrary, while the quality differences are more distinct. This makes the quality the deciding factor for those recommendations.

In general however, the quality always stays within a range of 0.4 to 1.0, whereas the runtime spans multiple orders of magnitude. This leads to the ratio metric being close to the runtime metric, also among the reference implementation results, see table 7.4. Depending on the requirements of the application, the quality or runtime within the ratio may need to be weighted with a custom factor to obtain individual recommendations.

n \ d	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5
25	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 4$
50	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$
75	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 4$
100	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 4$
125	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 4$
150	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$
200	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$	UEW $\alpha = 4$
250	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 1$	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 2$	UEW $\alpha = 4$	FWC 0.02 s
500	UEW $\alpha = 2$	UEW $\alpha = 1$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 4$	FWC 0.02 s	FWC 0.02 s	FWC 0.02 s
1000	UEW $\alpha = 4$	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 2$	FWC 0.02 s	FWC 0.02 s	FWC 0.02 s	FWC 0.02 s
2000	UEW $\alpha = 4$	UEW $\alpha = 1$	UEW $\alpha = 2$	FWC 0.02 s	FWC 0.02 s	FWC 0.02 s	FWC 1 s	FWC 1 s
4000	UEW $\alpha = 2$	UEW $\alpha = 4$	UEW $\alpha = 1$	FWC 0.02 s	FWC 1 s	FWC 1 s	FWC 1 s	-

Table 5.4: Heuristic algorithms with the best quality to runtime ratio

6 Conclusion

6.1 Summary

In this thesis, different state of the art exact and heuristic algorithms for solving the maximum weight clique problem were presented. An own heuristic algorithm called UEW was proposed alongside its extension UEW-R and an approach to optimize consecutive runs of MWC algorithms on similar graphs. In total 220,320 test runs of the algorithms with various graphs and parametrizations were conducted to gather data about the suitability of the algorithms for different use cases regarding graph size and density.

The tests have shown that the exact algorithms have a very similar performance on the evaluated graphs. WLMC tends to be slightly faster than other algorithms on smaller and more sparse graphs, while WC-MWC, TSM-MWC and MWCRedu gain more traction on larger and more dense graphs. As for the heuristic algorithms, FastWClq for larger and more dense graphs and the newly proposed UEW for the other graphs show the best performance both in regard to the runtime and the solution quality to runtime ratio. When the primary focus lies on quality, additionally MWCPeel can be a good choice for graphs with an average or slightly above average amount of edges, as well as UEW-R for the small to medium configurations of the inspected graph types. Meanwhile the current approach for iterative runs on similar graphs likely requires further investigation and optimization.

6.2 Outlook

In future work, more algorithms or modifications of the inspected ones could be investigated. The proposed UEW heuristic could be integrated with other algorithms. The possibilities reach from using it to deliver initial cliques for exact algorithms to combining it with the benefit estimation in FastWClq. Also analyzing optimization potential to speed up the existing implementations, especially for tasks that are executed often, might result in significant speed-ups. At the time when a reference implementation of MWCRedu and MWCPeel is released by the authors of [17], this could also be tested against the other reference implementations, which, together with a yet to be created low level implementation of UEW and UEW-R, could bring a new perspective to the comparisons.

While this thesis tapped into the issue of iterative runs on similar graphs, chapter 5.3.2 already discussed that more diverse tests would help to get a clearer understanding of the potential there.

To further broaden the comparison, the set of test graphs could be extended. More graphs for the same configurations of vertex number, density and weight distribution could be generated to obtain more stable results. The steps in which the parameters are varied could be decreased to get more fine-grained comparisons. Different vertex degree distributions and other graph generation techniques in general can be explored too. Besides synthetically generated graphs, more "real" data taken from different domains could be taken into account as well. While this thesis focused on rather small graphs, an augmentation of the comparison with larger or more dense graphs could be very interesting as well.

Depending on the time criticality of the application, the time limit within which the algorithms have to produce a result can be varied. Running algorithms more often on each graph yields another possibility to get a more realistic estimation of their performance. This is especially the case for non-deterministic heuristic algorithms like FastWClq or SCCWalk.

What most of the proposed extensions of the comparisons have in common is a potentially great increase of the overall runtime of the comparison. So the available resources regarding time and computing power have to be taken into account.

7 Appendix

7.1 Contents of the DVD

- **aggregated_data** Folder containing the aggregated measurements from the test runs (was also used for the plots in the thesis)
- **CliqueSuite** Folder containing the software written for the thesis
  - **build** Folder containing a build of the software for Raspberry Pi OS (64-Bit, release from 11th December 2023)
  - **src** Folder containing the C++ source code
  - **.clang-format** File specifying how clang-format should format the C++ source code
  - **CMakeLists.txt** File specifying how to build the software using CMake
  - **raspi_bulk_run.py** Python script that was used to execute the test runs on the Raspberry Pi
  - **README.md** Markdown document describing the parts of the software, their requirements and how to use them
- **MA_sebpe.pdf** The written thesis as a PDF document
- **test_data.zip** An archive containing the test graphs and algorithm runs (unpacked size ∼9.1 GiB)
  - **iterative_runs_s2_raspi_1** Folder containing the results of the iterative runs with 2 % change probability of the graphs on the 1st Raspberry Pi
  - **iterative_runs_s2_raspi_2** Folder containing the results of the iterative runs with 2 % change probability of the graphs on the 2nd Raspberry Pi
  - **iterative_runs_s2_raspi_3** Folder containing the results of the iterative runs with 2 % change probability of the graphs on the 3rd Raspberry Pi
  - **iterative_runs_s5_raspi_1** Folder containing the results of the iterative runs with 5 % change probability of the graphs on the 1st Raspberry Pi
  - **iterative_runs_s5_raspi_2** Folder containing the results of the iterative runs with 5 % change probability of the graphs on the 2nd Raspberry Pi

- **iterative_runs_s5_raspi_3** Folder containing the results of the iterative runs with 5 % change probability of the graphs on the 3rd Raspberry Pi
- **similar_graphs_s2** Folder containing the test graphs with 2 % change probability for the first step of iterative runs
- **similar_graphs_s5** Folder containing the test graphs with 5 % change probability for the first step of iterative runs
- **standard_graphs** Folder containing the test graphs for the standalone runs
- **standard_runs_raspi_1** Folder containing the results of the standalone runs on the 1st Raspberry Pi
- **standard_runs_raspi_2** Folder containing the results of the standalone runs on the 2nd Raspberry Pi
- **standard_runs_raspi_3** Folder containing the results of the standalone runs on the 3rd Raspberry Pi

7.2 WC-MWC

Algorithm 27: WC-MWC

```
(C_0, O_0, G') := InitializeWLMC(G, 0);
Ĉ := C_0;
V' := vertices of G';
order V' w.r.t. O_0;
for i := |V'| to 1 do
    Candidates := N(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V'|}};
    if w(Candidates) + w(v_i) > w(Ĉ) then
        (C'_0, O'_0, G'') := InitializeWLMC(G[Candidates], w(Ĉ) − w(v_i));
        if w(C'_0) + w(v_i) > w(Ĉ) then
            Ĉ := C'_0 ∪ {v_i};
        C' := SearchMaxWCliqueWC-WMC(G'', Ĉ, {v_i}, O'_0);
        if w(C') > w(Ĉ) then
            Ĉ := C';
return Ĉ;
```

Algorithm 28: SearchMaxWCliqueWC-MWC($G, \hat{C}, C, O$)

if $V = \emptyset$ **then**
  return C;
$(A, B) := PartitionWC\text{-}MWC(G, w(\hat{C}) - w(C), O)$;
if $B = \emptyset$ **then**
  return $\hat{C}$;
Let $B = \{b_1, b_2, \ldots, b_{|B|}\}, b_1 < b_2 < \ldots < b_{|B|}$ w.r.t. O;
for $i := |B|$ *to* 1 **do**
  $Candidates := N(b_i) \cap (\{b_{i+1}, b_{i+2}, \ldots, b_{|B|}\} \cup A)$;
  if $w(C \cup \{b_i\}) + w(Candidates) > w(\hat{C})$ **then**
    $C' := SearchMaxWCliqueWC\text{-}MWC(G[Candidates], \hat{C}, C \cup \{b_i\}, O)$;
    if $w(C') > w(\hat{C})$ **then**
      $\hat{C} := C'$;
return $\hat{C}$;

7.3 TSM-MWC

Algorithm 29: TSM-MWC

$(C_0, O_0, G') := InitializeWLMC(G, 0)$;
$\hat{C} := C_0$;
$V' :=$ vertices of G';
order V' w.r.t. O_0;
for $i := |V'|$ *to* 1 **do**
  $Candidates := N(v_i) \cap \{v_{i+1}, v_{i+2}, \ldots, v_{|V'|}\}$;
  if $w(Candidates) + w(v_i) > w(\hat{C})$ **then**
    $(C'_0, O'_0, G'') := InitializeWLMC(G[Candidates], w(\hat{C}) - w(v_i))$;
    if $w(C'_0) + w(v_i) > w(\hat{C})$ **then**
      $\hat{C} := C'_0 \cup \{v_i\}$;
    $C' := SearchMaxWCliqueTSM\text{-}WMC(G'', \hat{C}, \{v_i\}, O'_0)$;
    if $w(C') > w(\hat{C})$ **then**
      $\hat{C} := C'$;
return $\hat{C}$;

Algorithm 30: SearchMaxWCliqueTSM-MWC($G, \hat{C}, C, O$)

```
if V = ∅ then
    return C;
B := GetBranchesTSM-MWC(G, w(Ĉ) − w(C), O);
if B = ∅ then
    return Ĉ;
A := V \ B;
Let B = {b_1, b_2, ..., b_|B|}, b_1 < b_2 < ... < b_|B| w.r.t. O;
for i := |B| to 1 do
    Candidates := N(b_i) ∩ ({b_{i+1}, b_{i+2}, ..., b_|B|} ∪ A);
    if w(C ∪ {b_i}) + w(Candidates) > w(Ĉ) then
        C' := SearchMaxWCliqueTSM-MWC(G[Candidates], Ĉ, C∪{b_i}, O);
        if w(C') > w(Ĉ) then
            Ĉ := C';
return Ĉ;
```

7.4 Results

7.4.1 Exact algorithms

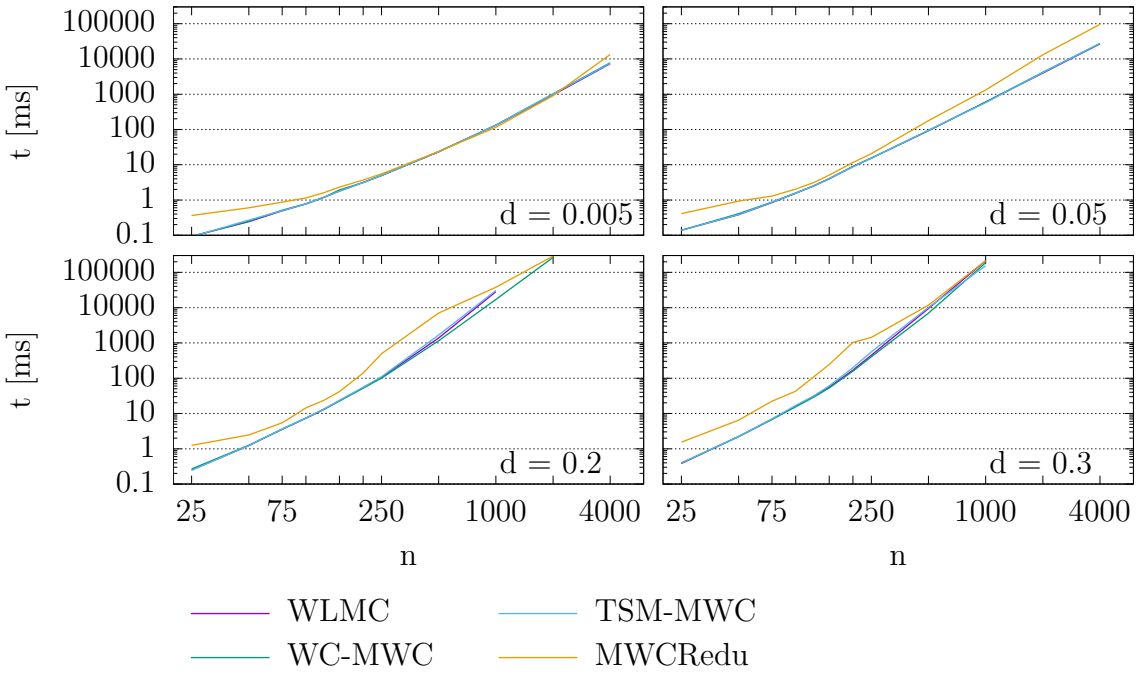

Figure 7.1: Runtimes of the exact algorithms (other densities)

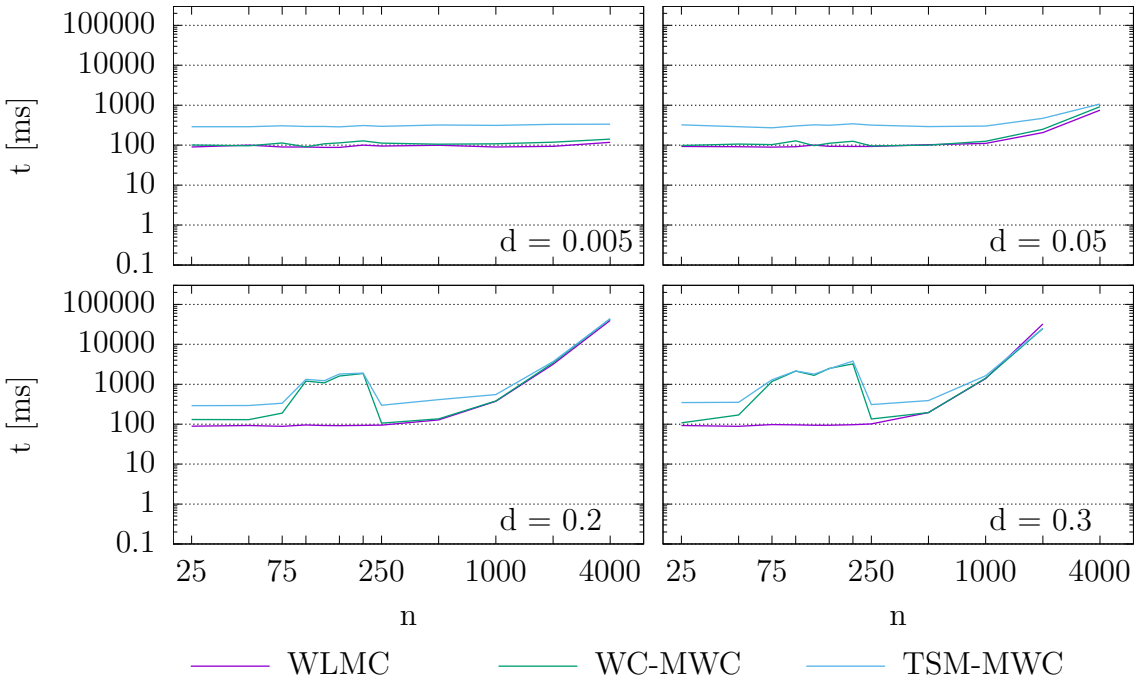

Figure 7.2: Runtimes of the exact algorithms (reference implementations, other densities)

n \ d	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5
25	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC
50	WL MC	WC- MWC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC
75	WL MC	WL MC	WC- MWC	WL MC	WL MC	WL MC	WL MC	WL MC
100	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC
125	WL MC	WL MC	WL MC	WC- MWC	WL MC	WL MC	WL MC	WL MC
150	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC
200	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC
250	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC
500	WL MC	WL MC	WL MC	WC- MWC	WC- MWC	WL MC	WC- MWC	WC- MWC
1000	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	TSM- MWC
2000	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	WC- MWC	-
4000	WL MC	WL MC	WL MC	WL MC	WL MC	WL MC	-	-

Table 7.1: Exact algorithms (reference implementations) with the best runtime

7.4.2 Heuristic algorithms

Own implementations

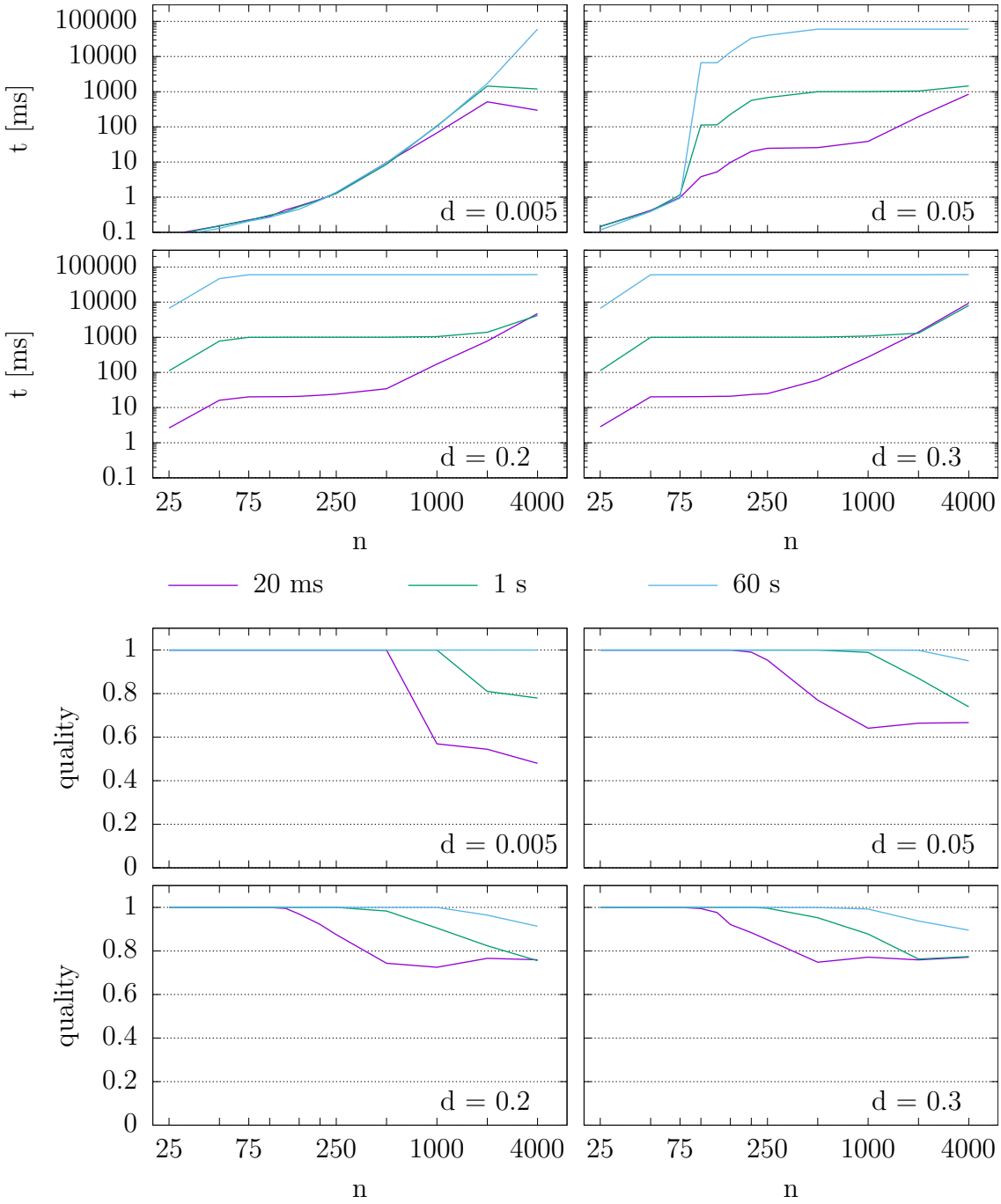

Figure 7.3: Measurements of FastWClq with different cutoff times (other densities)

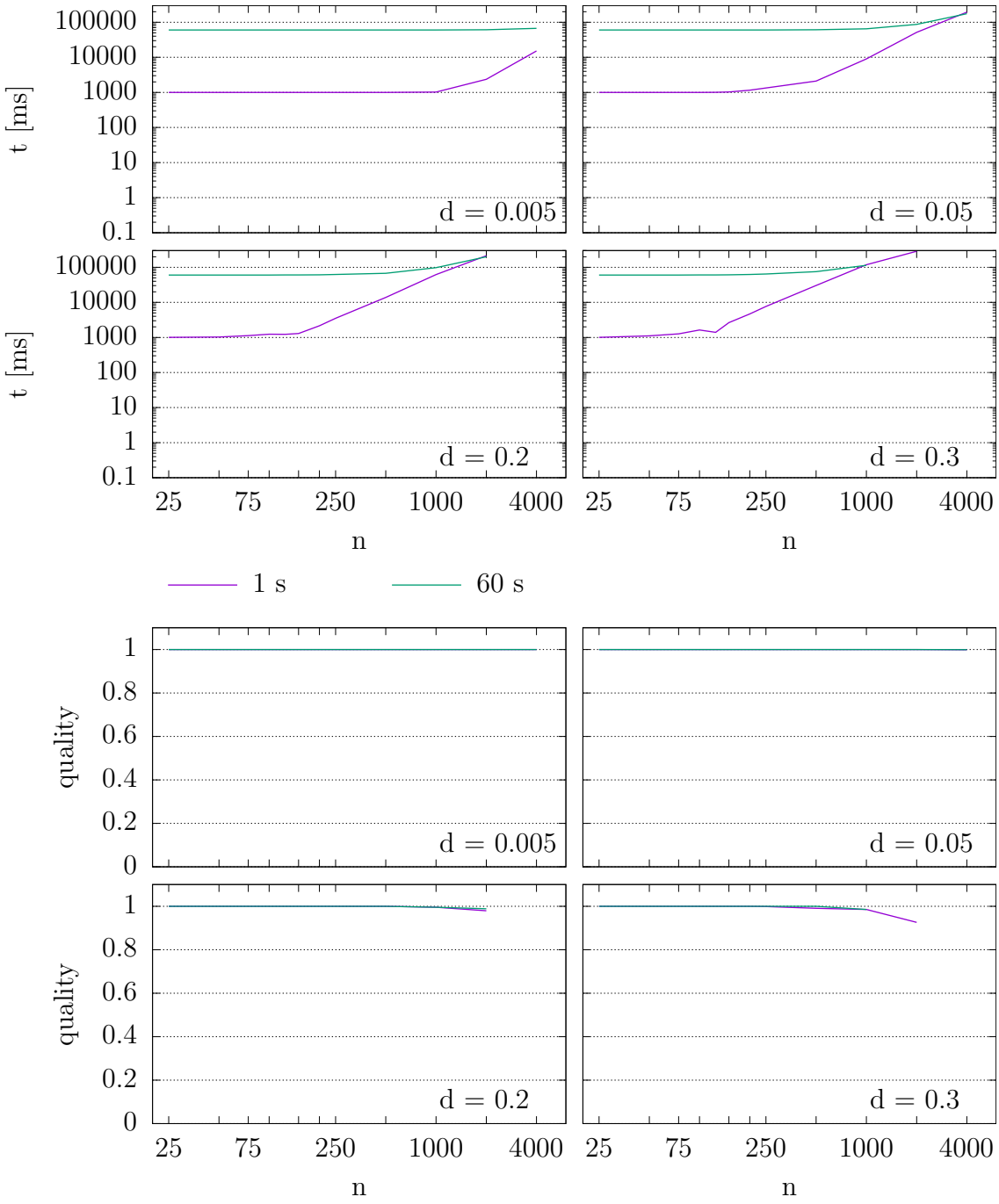

Figure 7.4: Measurements of SCCWalk with different cutoff times (other densities)

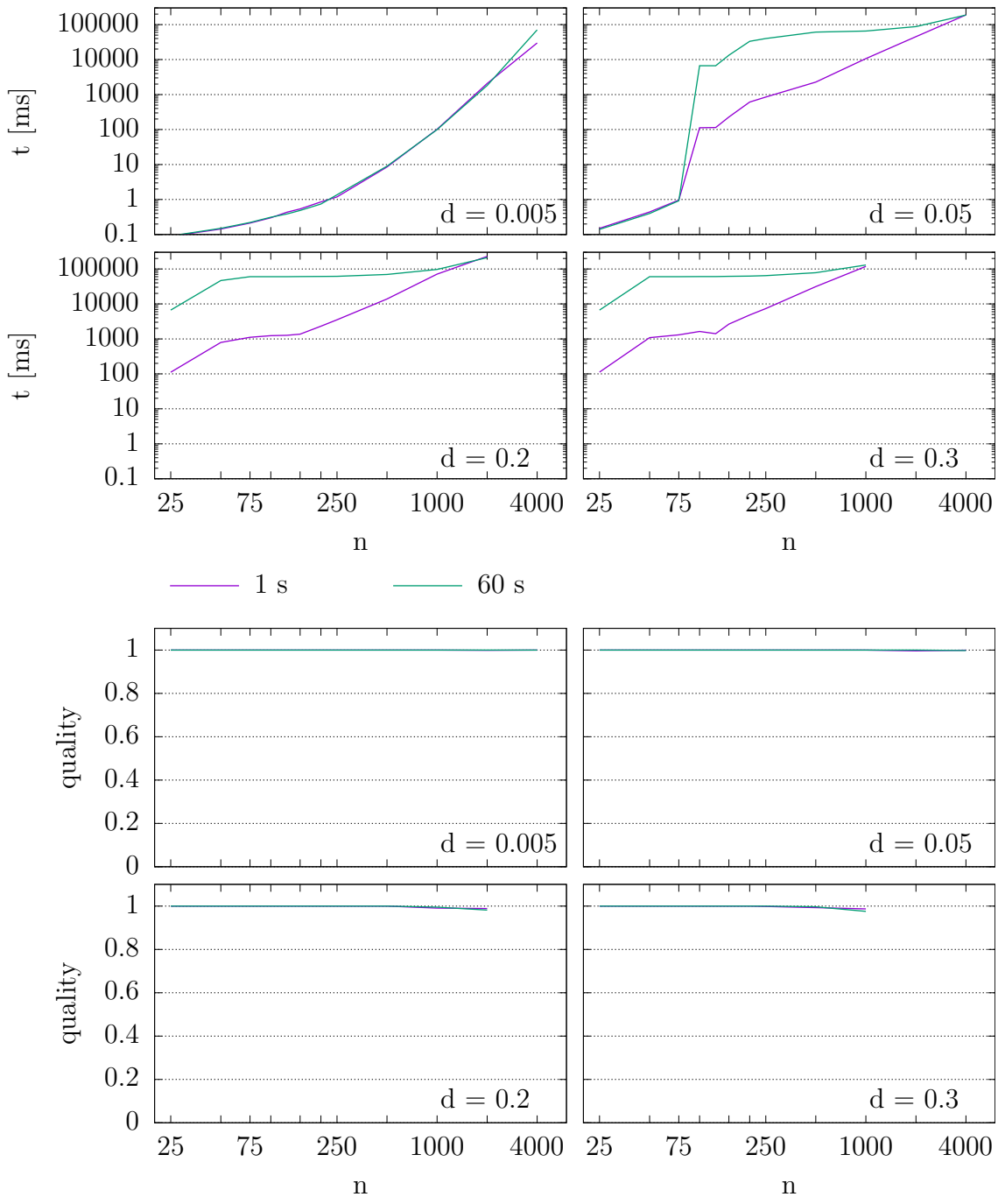

Figure 7.5: Measurements of SCCWalk4L with different cutoff times (other densities)

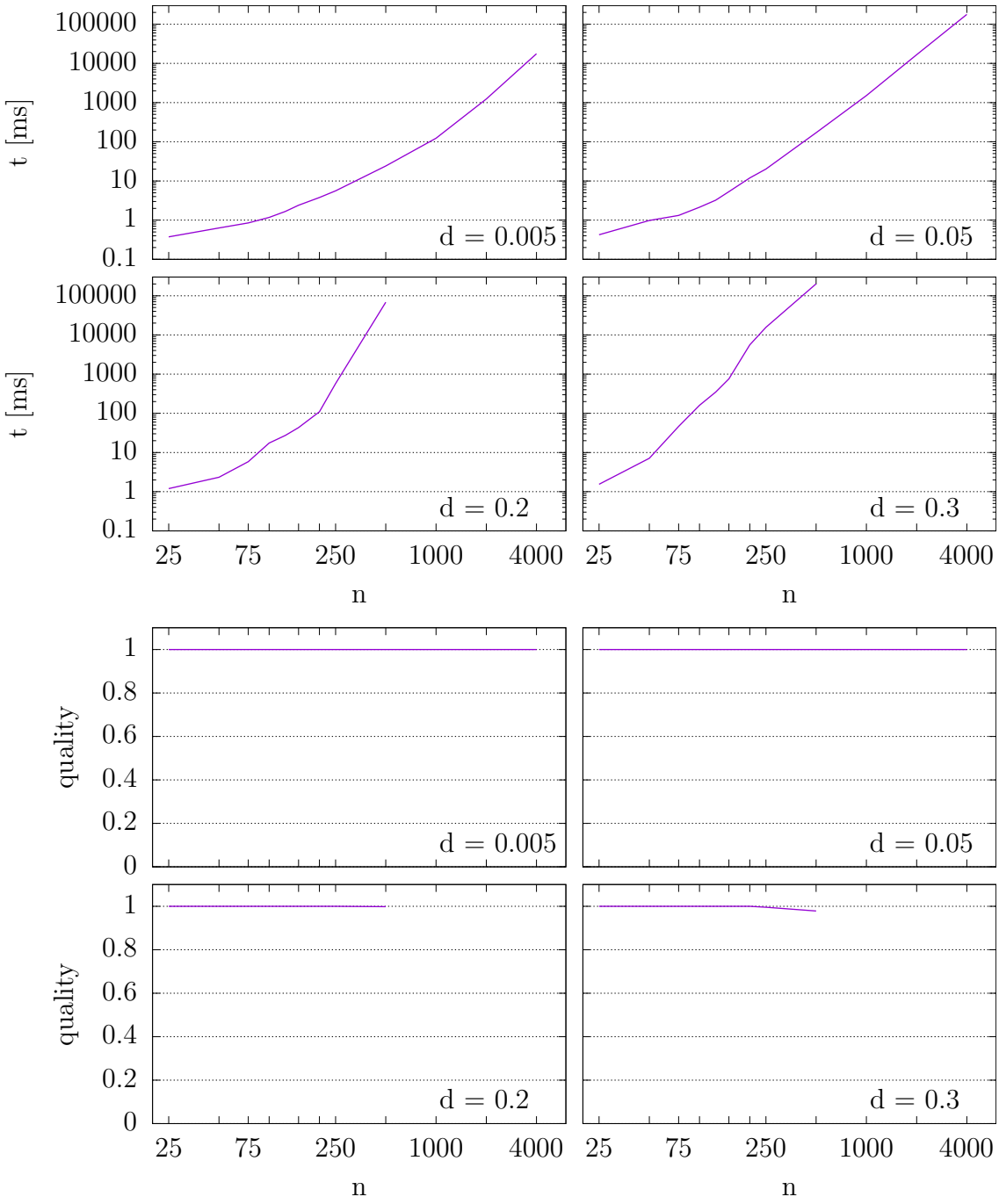

Figure 7.6: Measurements of MWCPeel (other densities)

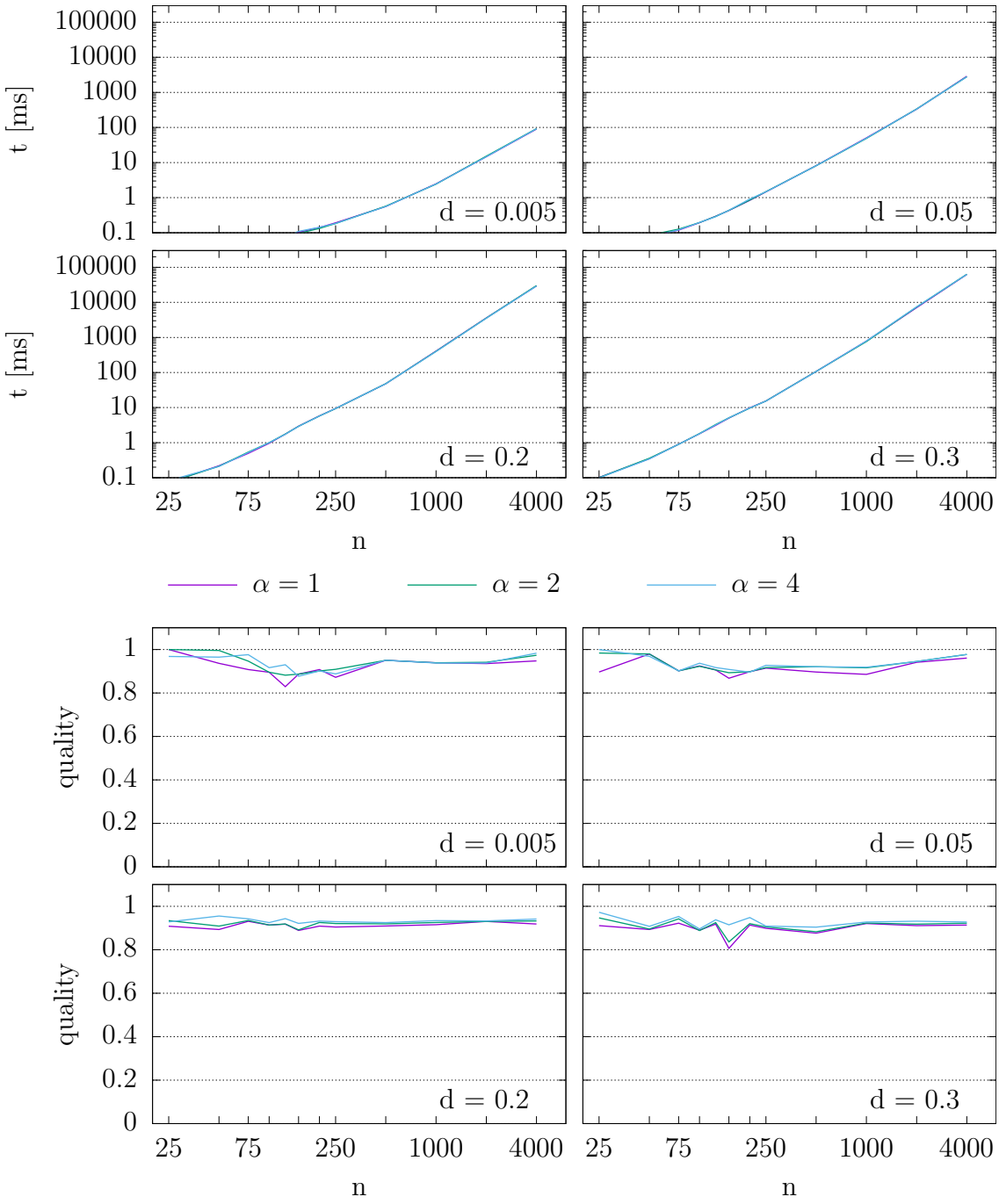

Figure 7.7: Measurements of UEW with different own weight priorities (other densities)

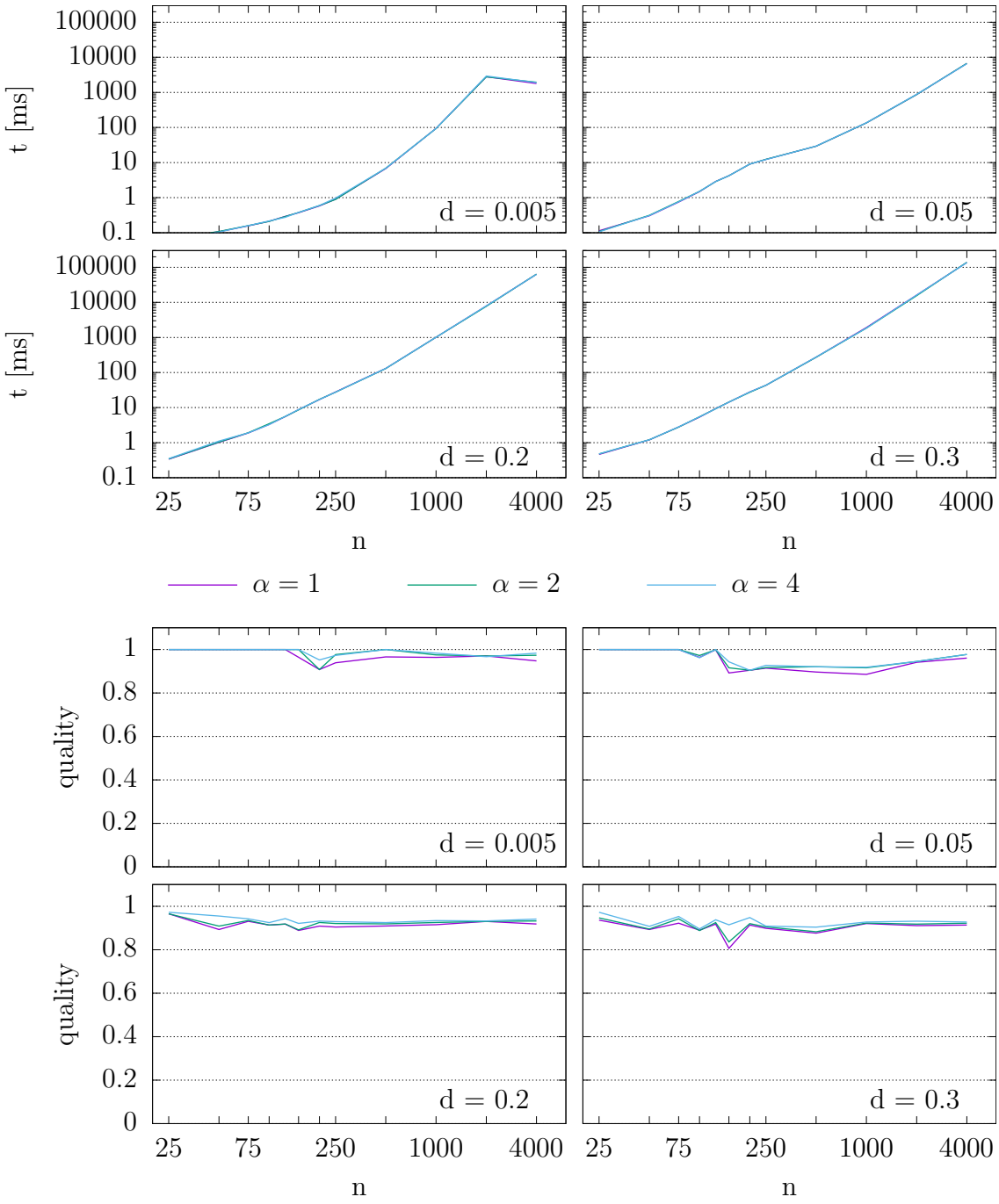

Figure 7.8: Measurements of UEW-R with different own weight priorities (other densities)

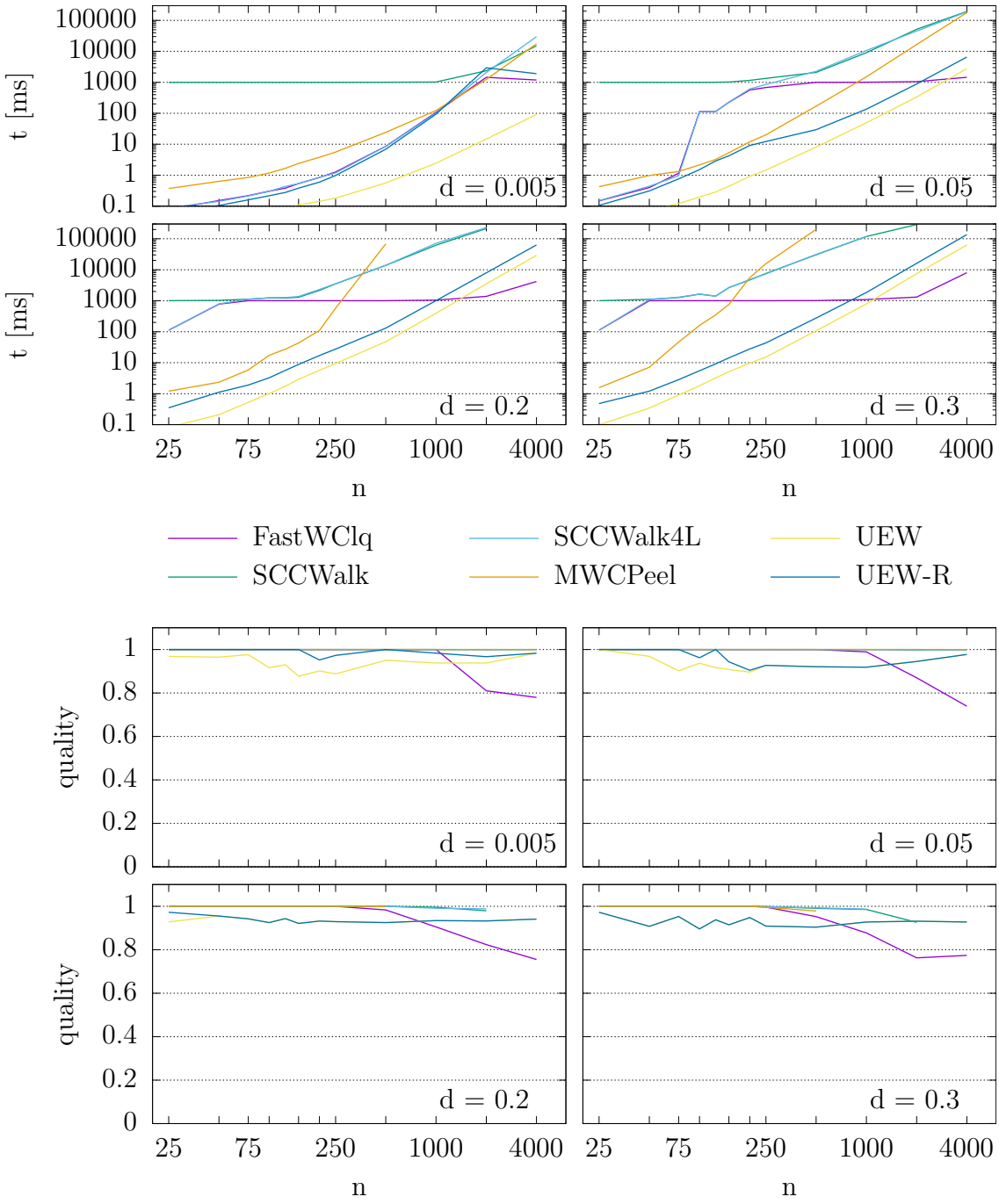

Figure 7.9: Measurements of the heuristic algorithms (other densities)

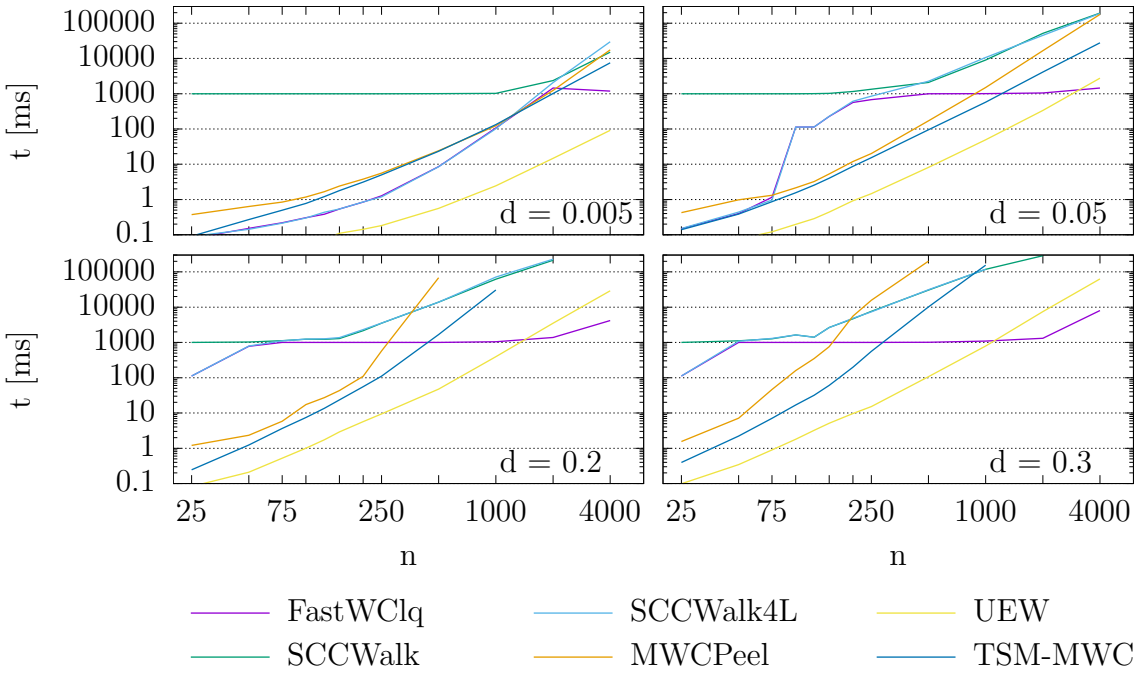

Figure 7.10: Measurements of the heuristic algorithms compared to TSM-MWC (other densities)

Reference implementations

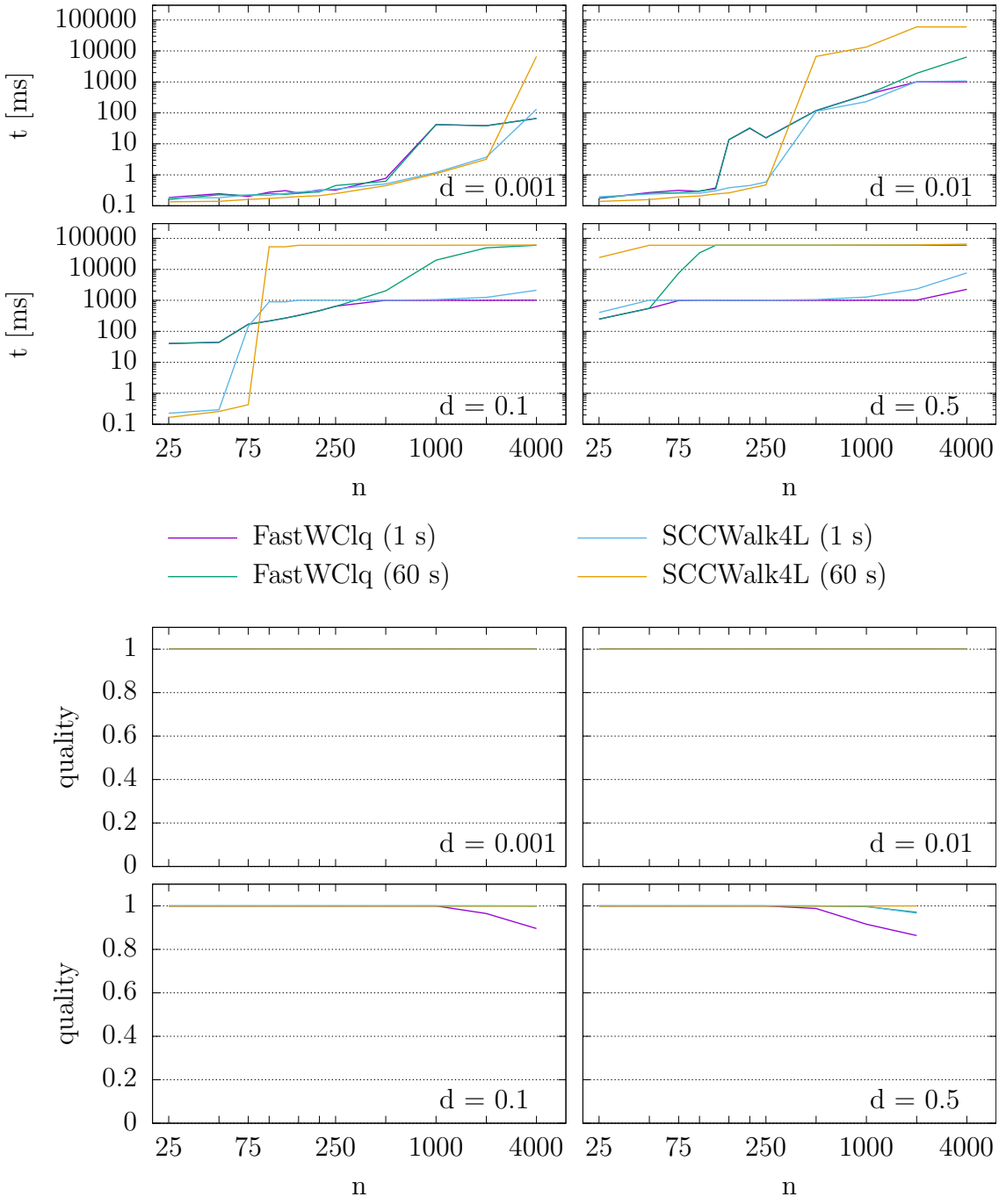

Figure 7.11: Measurements of the heuristic algorithms (reference implementations)

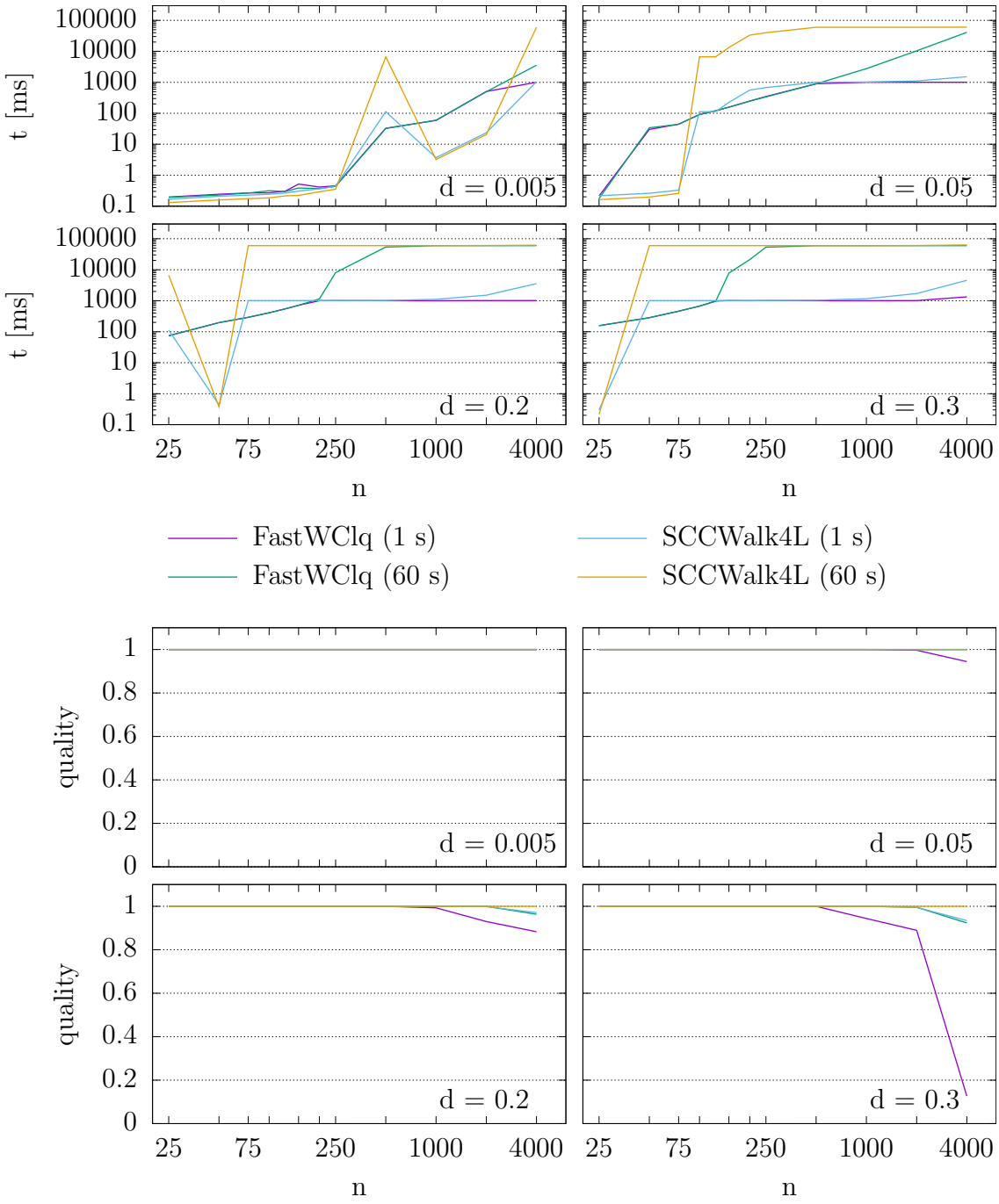

Figure 7.12: Measurements of the heuristic algorithms (reference implementations, other densities)

n \ d	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5
25	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	S4L 60 s	FWC 60 s
50	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	FWC 1 s	FWC 1 s
75	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	FWC 1 s	FWC 1 s	FWC 1 s
100	S4L 60 s	S4L 60 s	S4L 60 s	FWC 1 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s
125	S4L 60 s	S4L 60 s	S4L 60 s	S4L 1 s	FWC 1 s	FWC 60 s	FWC 60 s	S4L 1 s
150	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 60 s	FWC 60 s	S4L 1 s	FWC 1 s
200	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s
250	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s
500	S4L 60 s	FWC 60 s	S4L 1 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s
1000	S4L 60 s	S4L 60 s	S4L 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s
2000	S4L 60 s	S4L 60 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s
4000	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s

Table 7.2: Heuristic algorithms (reference implementations) with the best runtime

n \ d	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5
25	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 60 s	FWC 60 s
50	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 1 s	FWC 1 s
75	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s
100	S4L 60 s	S4L 60 s	S4L 60 s	FWC 1 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s
125	S4L 60 s	S4L 60 s	S4L 60 s	S4L 1 s	FWC 1 s	FWC 60 s	FWC 60 s	S4L 1 s
150	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 60 s	FWC 60 s	S4L 1 s	FWC 1 s
200	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s
250	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s
500	S4L 60 s	S4L 1 s	S4L 1 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s	S4L 1 s
1000	S4L 60 s	S4L 60 s	S4L 1 s	FWC 1 s	FWC 1 s	S4L 1 s	S4L 1 s	S4L 60 s
2000	S4L 60 s	S4L 60 s	FWC 1 s	S4L 1 s	S4L 1 s	FWC 60 s	S4L 60 s	S4L 60 s
4000	FWC 60 s	FWC 1 s	FWC 1 s	S4L 1 s	S4L 1 s	S4L 60 s	S4L 60 s	-

Table 7.3: Heuristic algorithms (reference implementations) with the best quality

n \ d	0.001	0.005	0.01	0.05	0.1	0.2	0.3	0.5
25	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	S4L 60 s	FWC 60 s
50	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	FWC 1 s	FWC 1 s
75	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	S4L 60 s	FWC 1 s	FWC 1 s	FWC 1 s
100	S4L 60 s	S4L 60 s	S4L 60 s	FWC 1 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s
125	S4L 60 s	S4L 60 s	S4L 60 s	S4L 1 s	FWC 1 s	FWC 60 s	FWC 60 s	S4L 1 s
150	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 60 s	FWC 60 s	S4L 1 s	FWC 1 s
200	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s
250	S4L 60 s	S4L 60 s	S4L 60 s	FWC 60 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s
500	S4L 60 s	FWC 60 s	S4L 1 s	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s
1000	S4L 60 s	S4L 60 s	S4L 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s
2000	S4L 60 s	S4L 60 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s
4000	FWC 60 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	FWC 1 s	S4L 1 s	-

Table 7.4: Heuristic algorithms (reference implementations) with the best quality to runtime ratio

7.4.3 Iterative runs on similar graphs

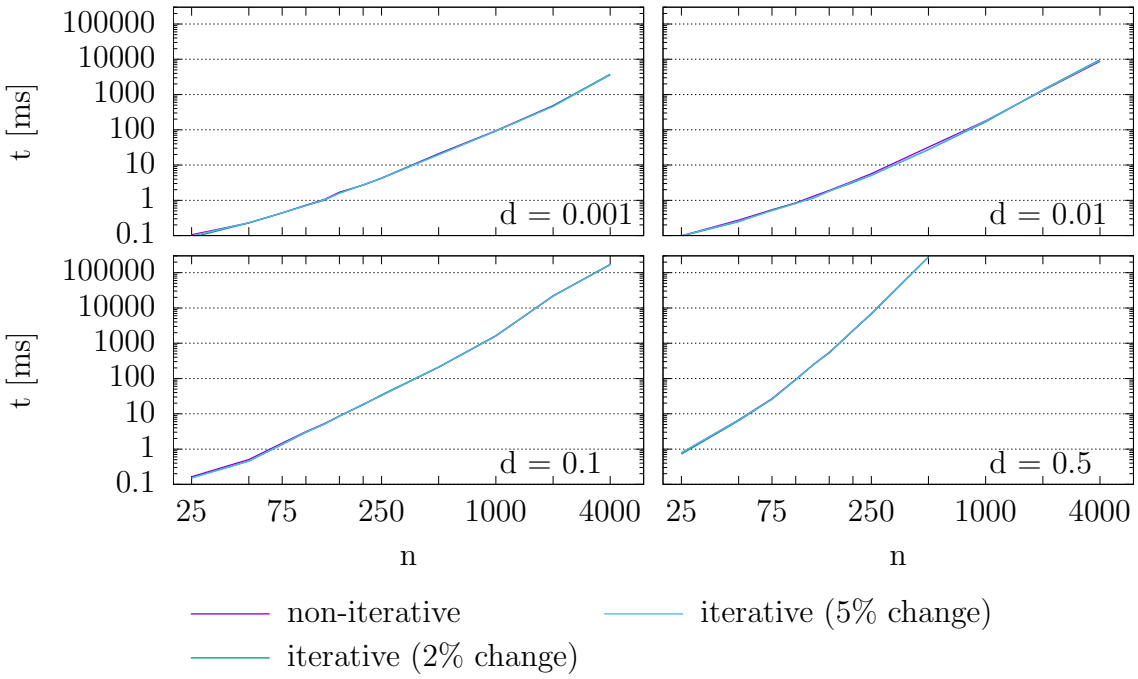

Figure 7.13: Measurements of iterative runs of WLMC

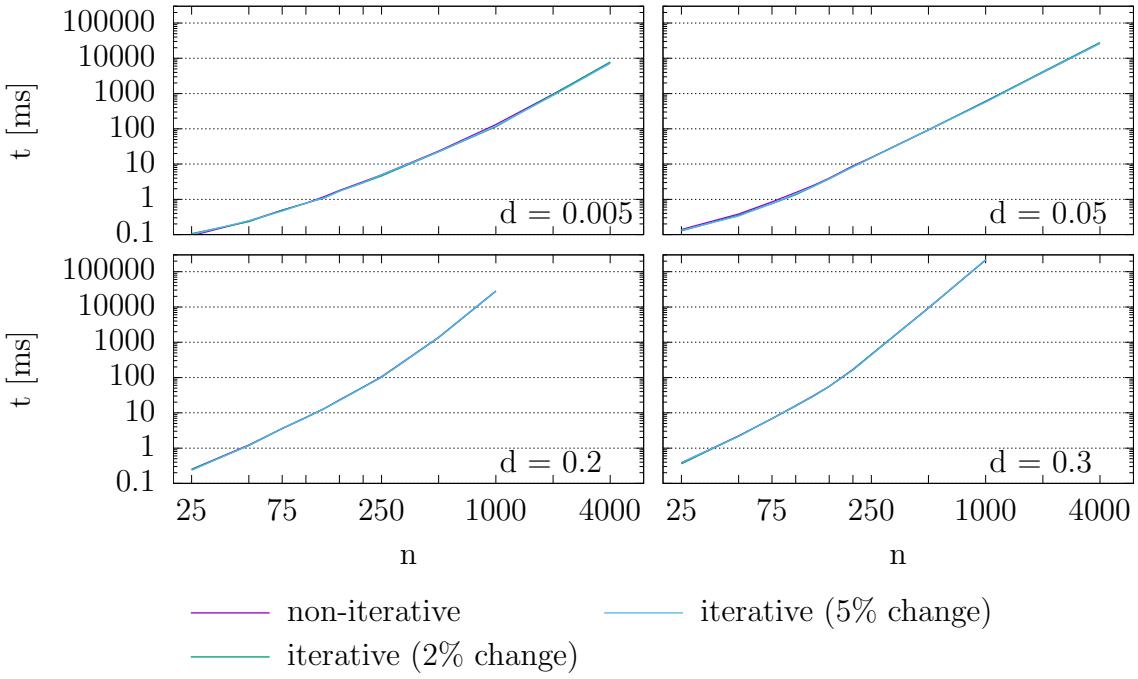

Figure 7.14: Measurements of iterative runs of WLMC (other densities)

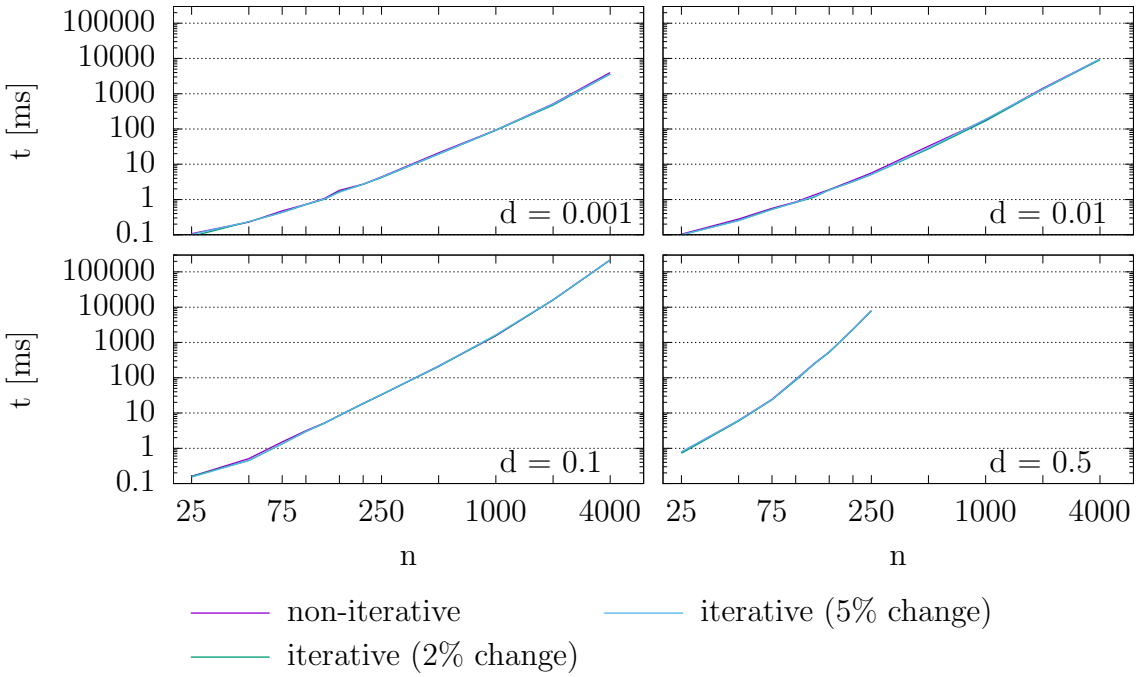

Figure 7.15: Measurements of iterative runs of WC-MWC

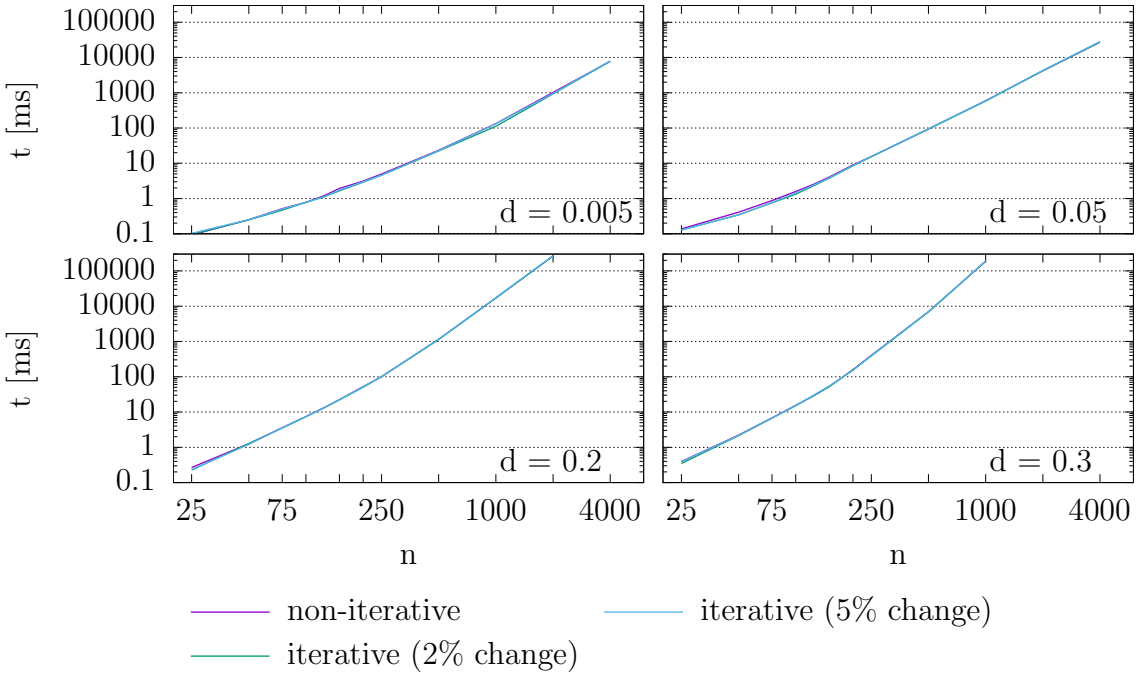

Figure 7.16: Measurements of iterative runs of WC-MWC (other densities)

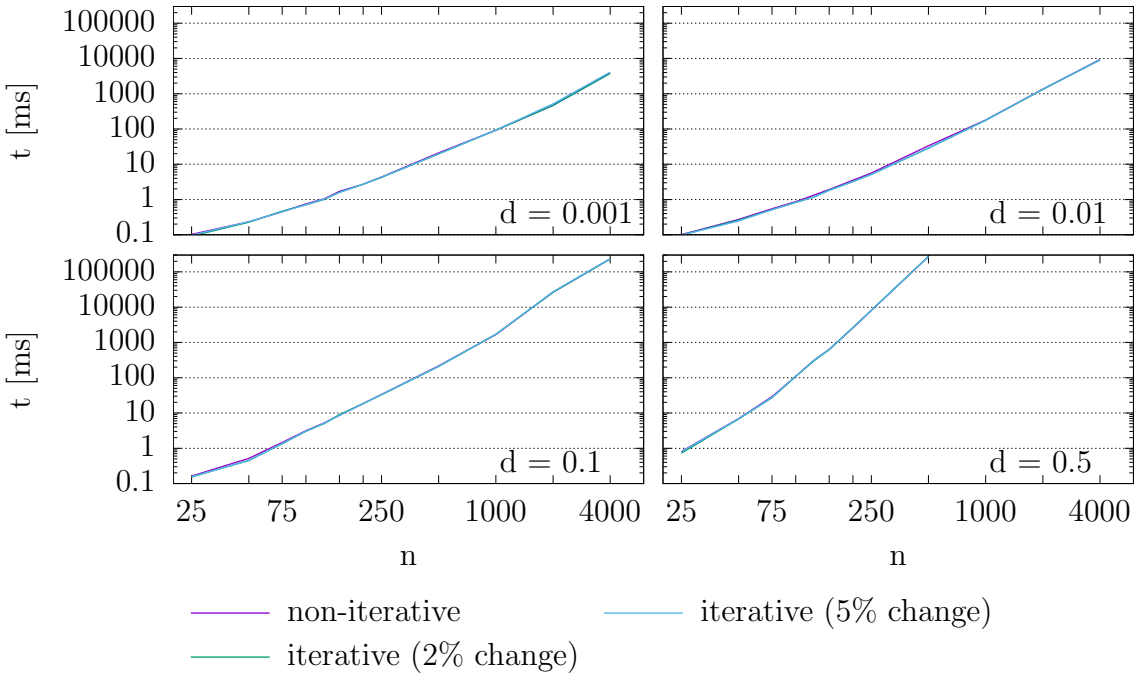

Figure 7.17: Measurements of iterative runs of TSM-MWC

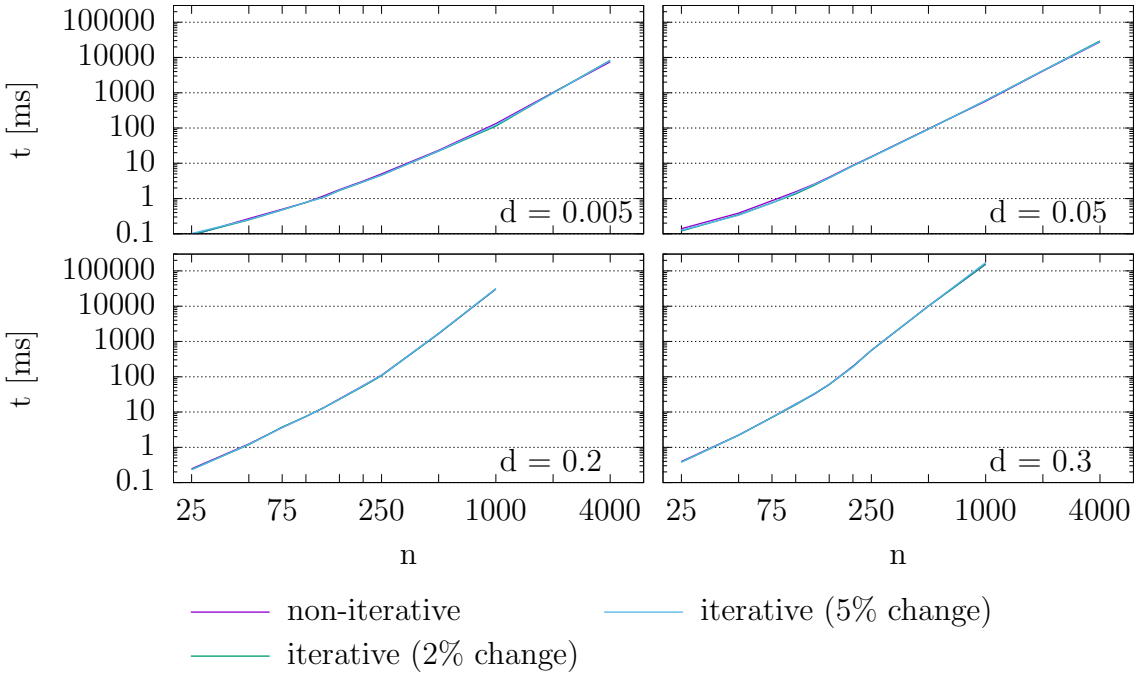

Figure 7.18: Measurements of iterative runs of TSM-MWC (other densities)

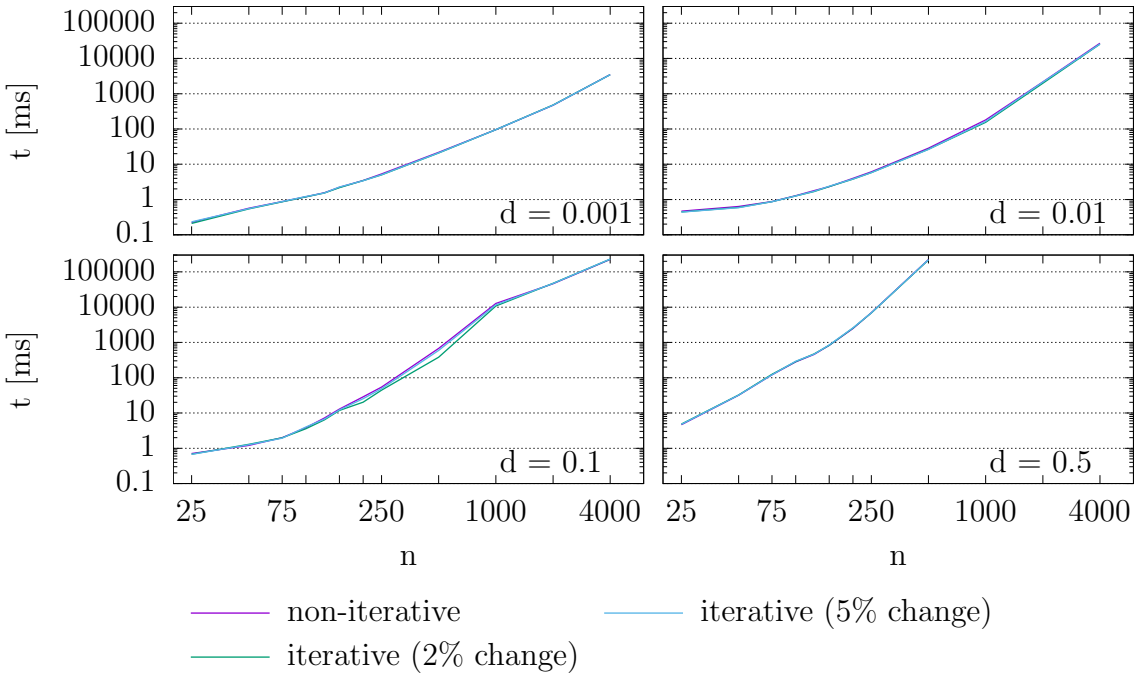

Figure 7.19: Measurements of iterative runs of MWCRedu

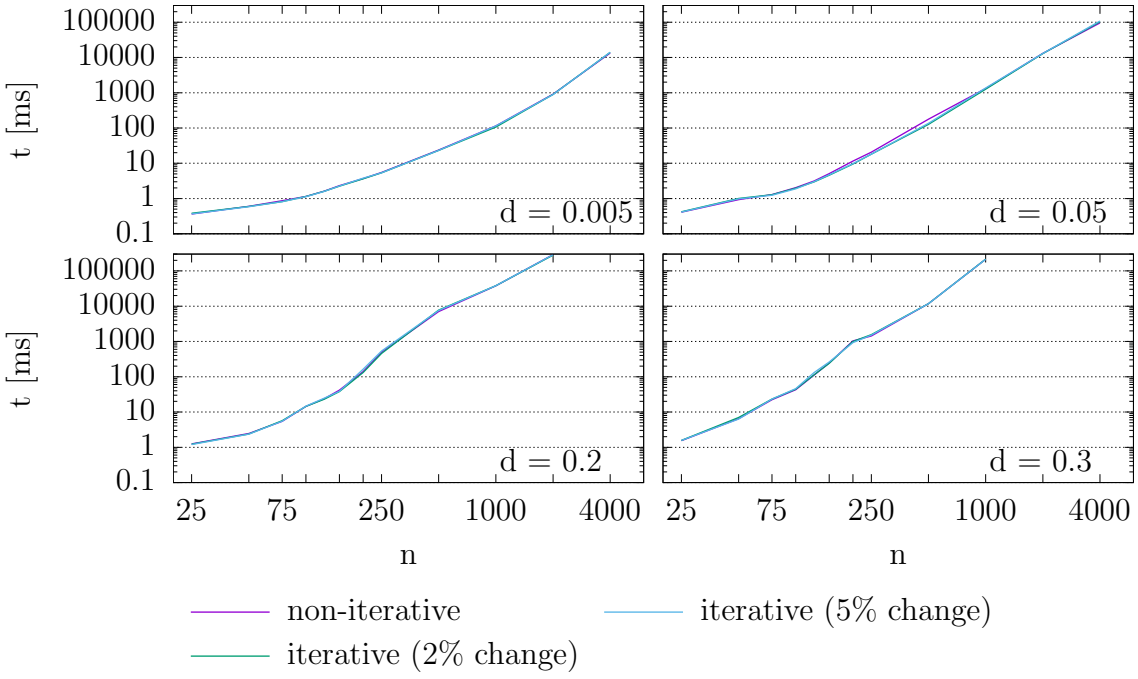

Figure 7.20: Measurements of iterative runs of MWCRedu (other densities)

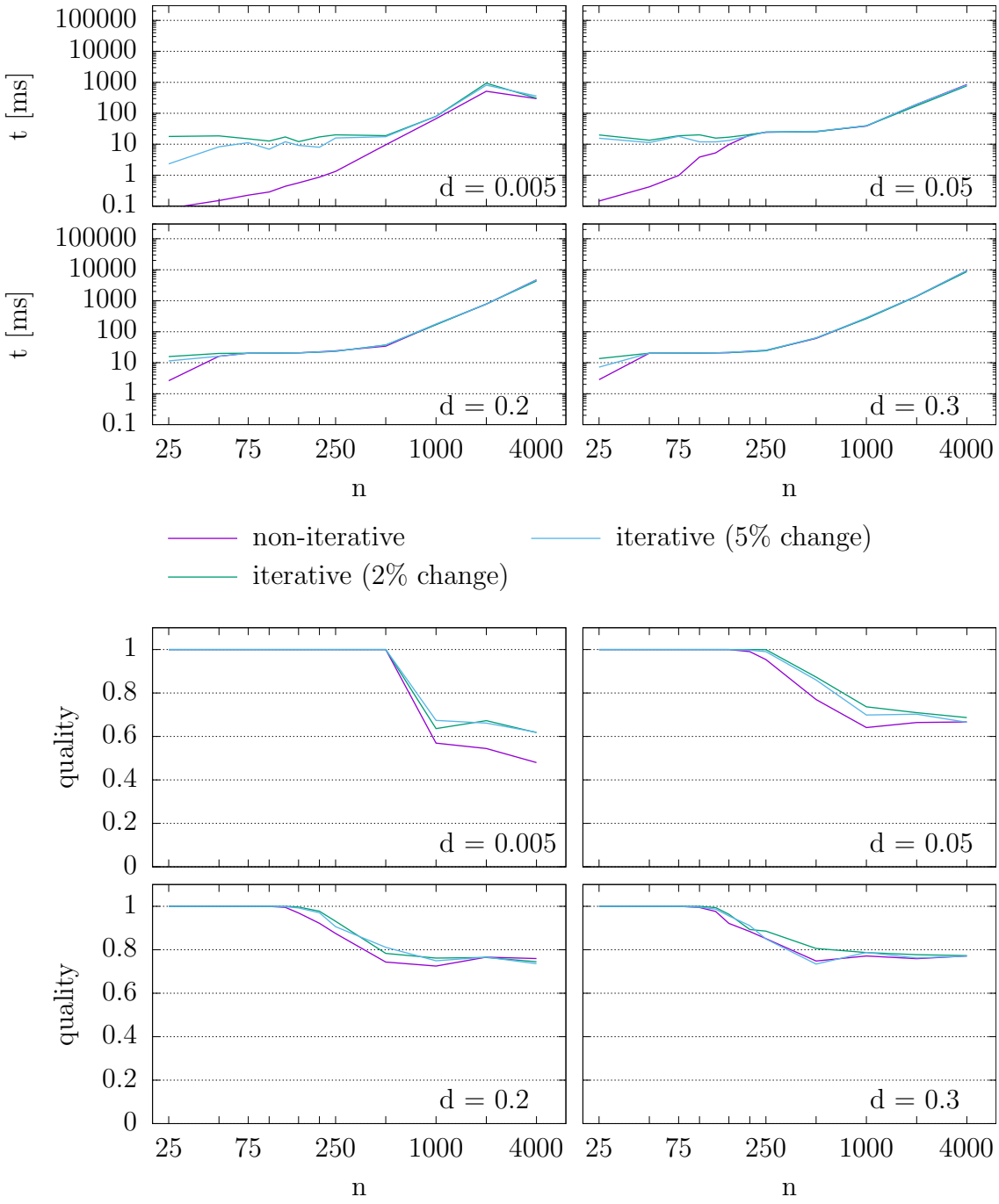

Figure 7.21: Measurements of iterative runs of FastWClq (20 ms cutoff time, other densities)

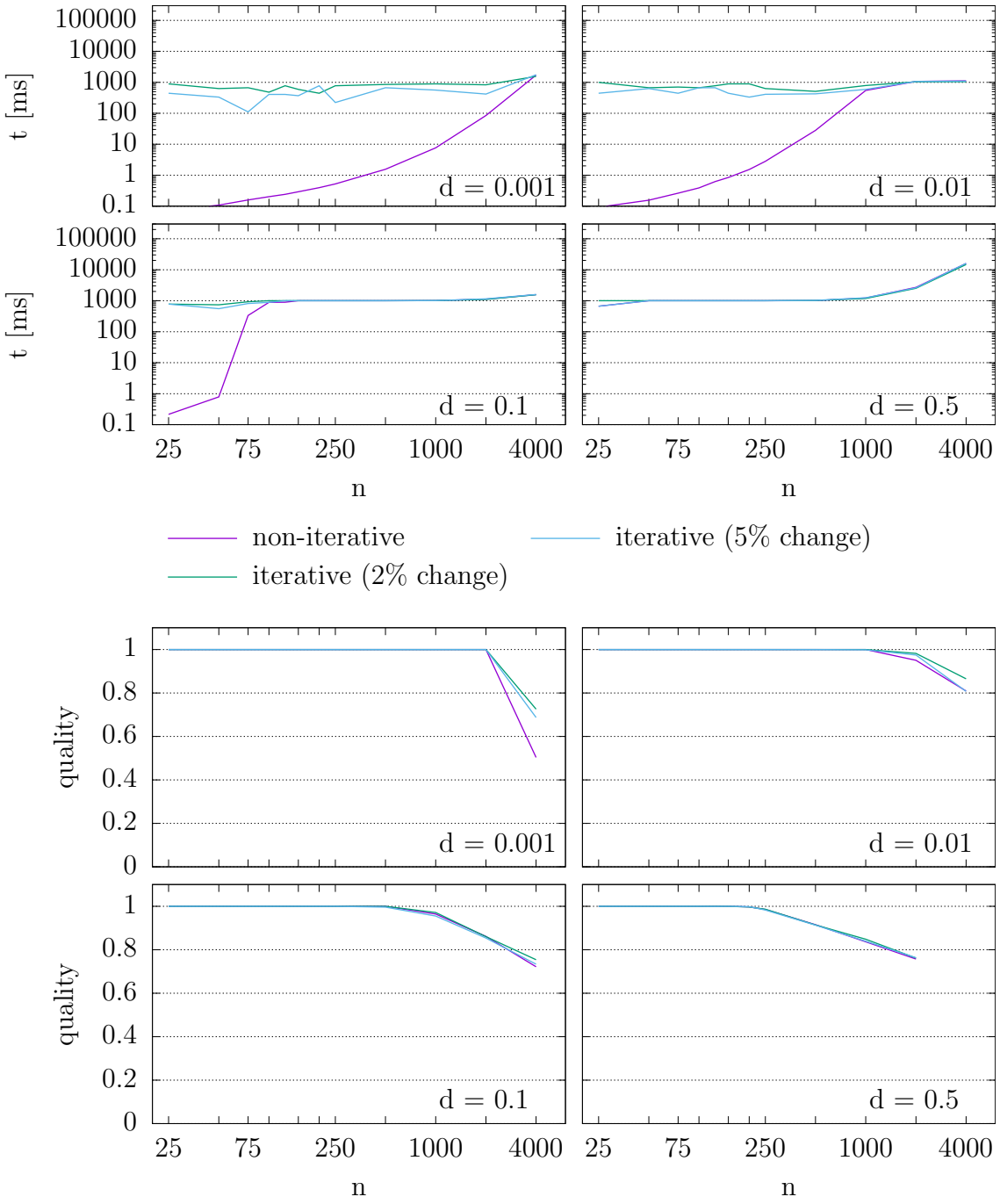

Figure 7.22: Measurements of iterative runs of FastWClq (1 s cutoff time)

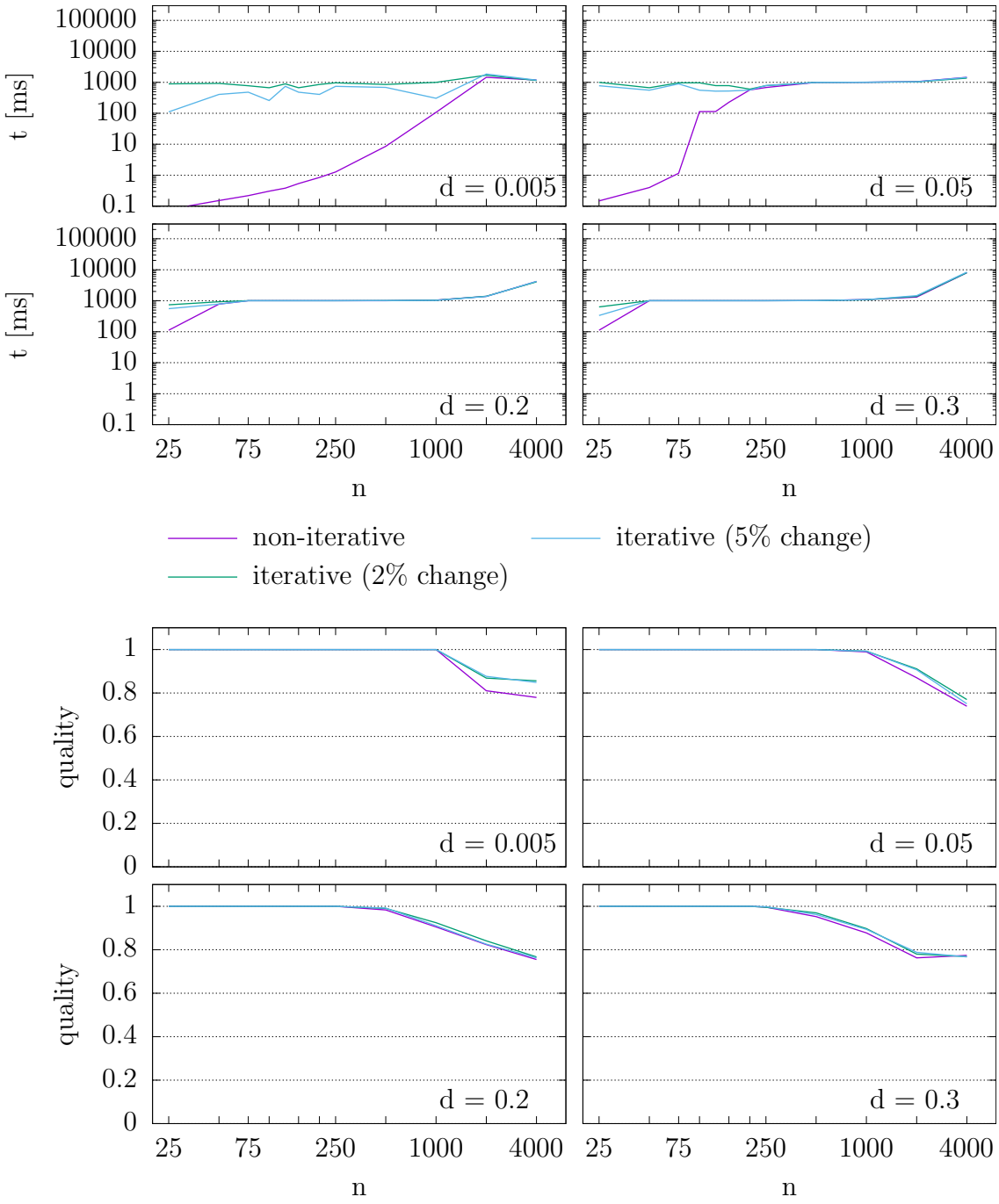

Figure 7.23: Measurements of iterative runs of FastWClq (1 s cutoff time, other densities)

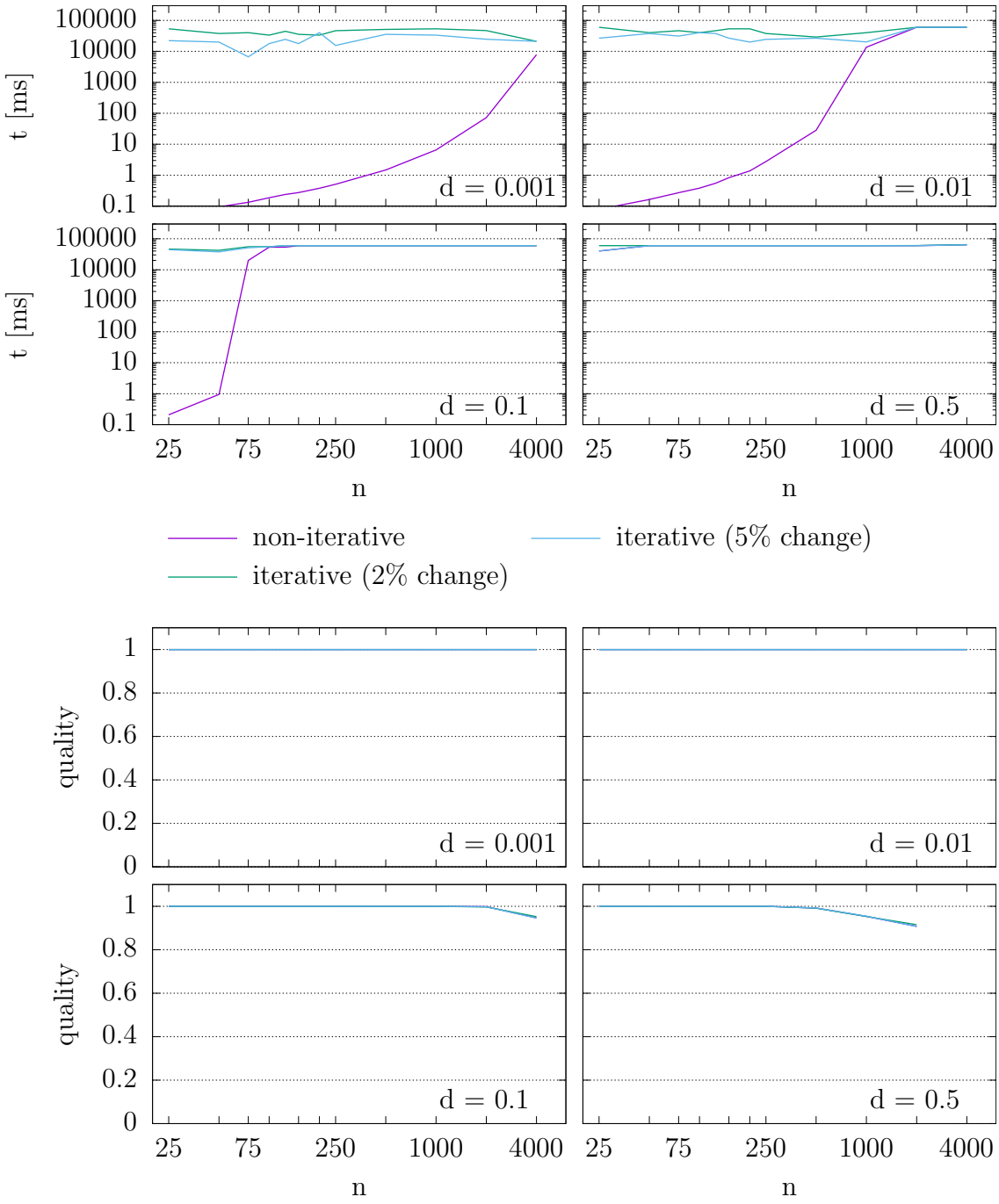

Figure 7.24: Measurements of iterative runs of FastWClq (60 s cutoff time)

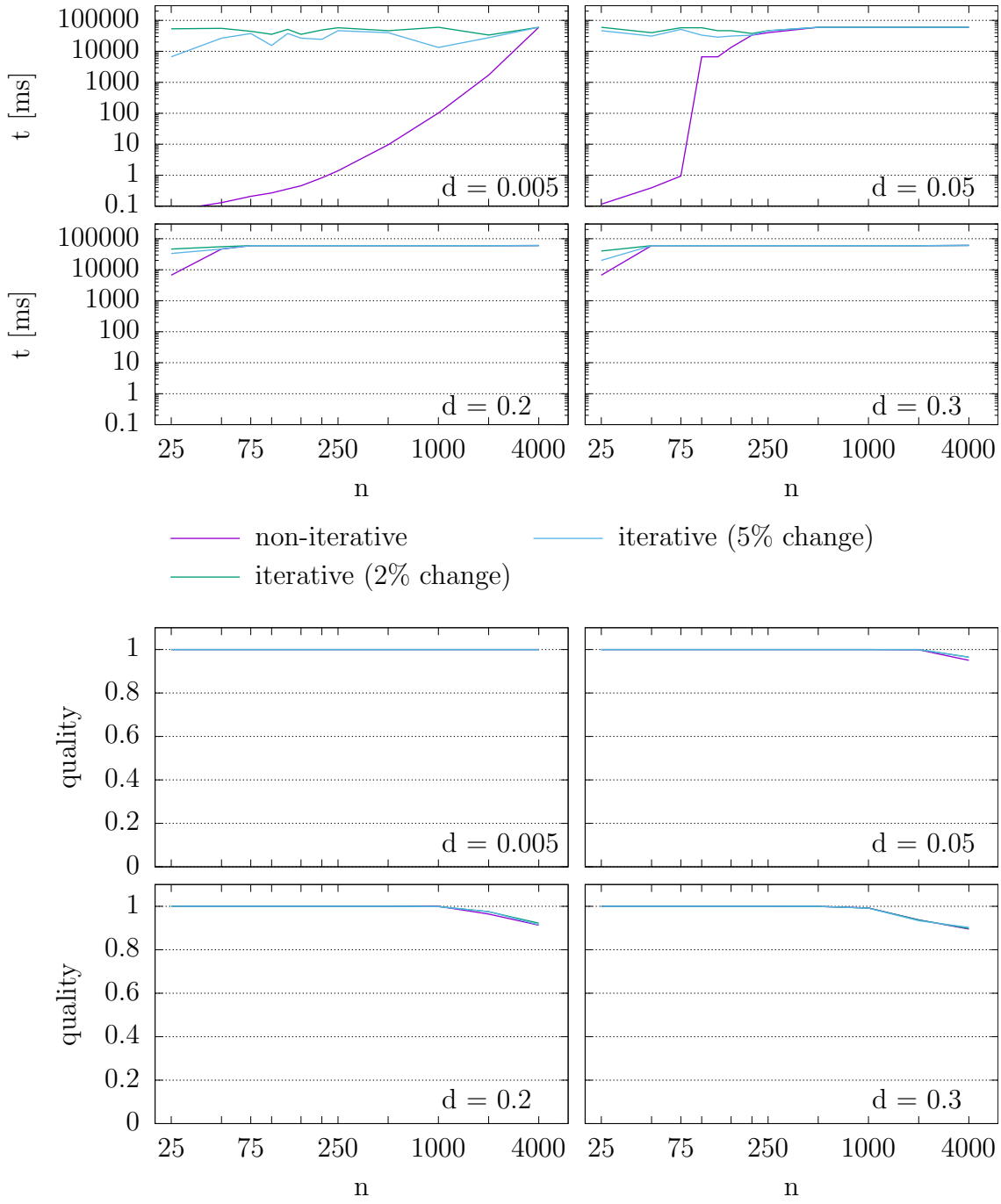

Figure 7.25: Measurements of iterative runs of FastWClq (60 s cutoff time, other densities)

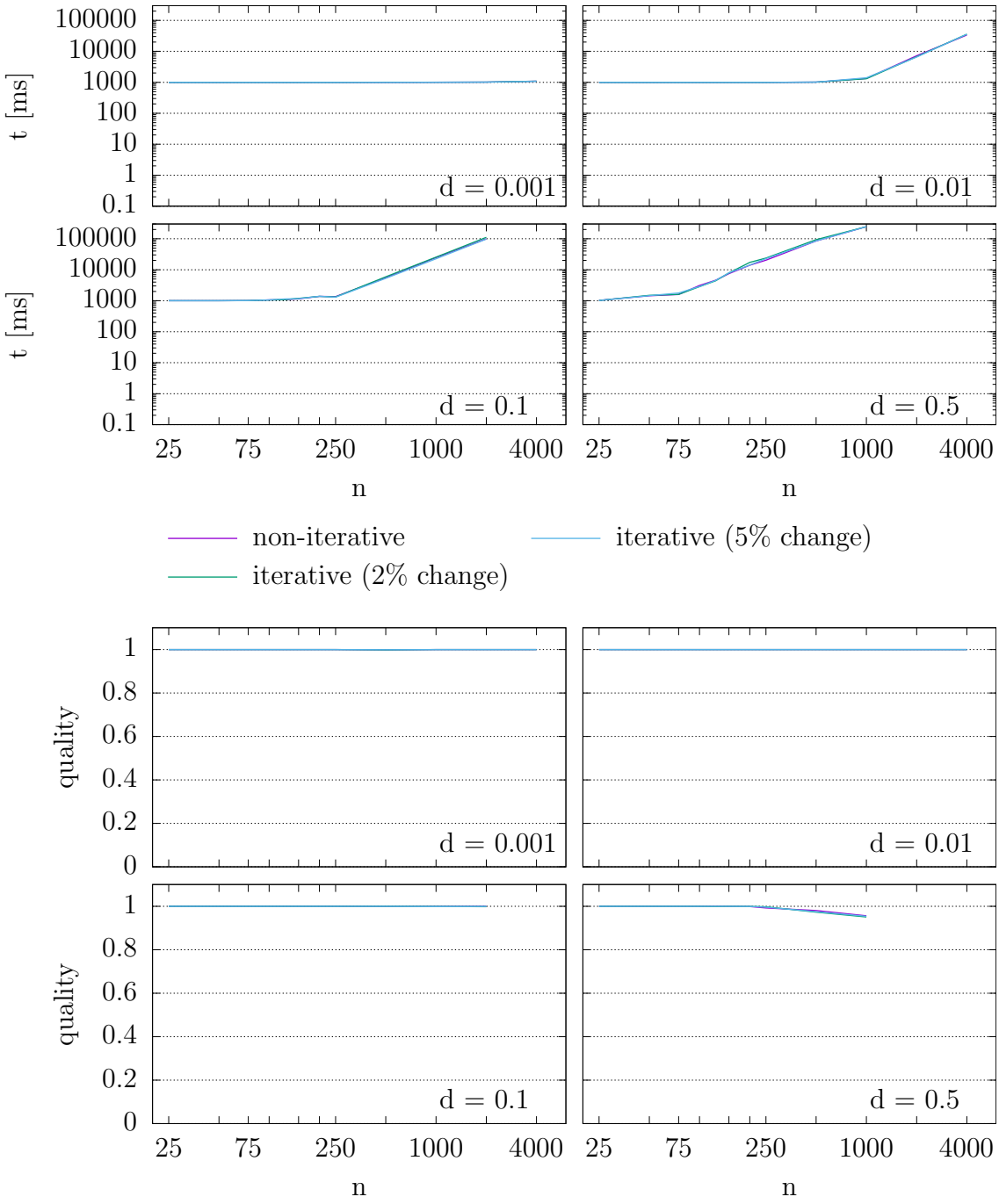

Figure 7.26: Measurements of iterative runs of SCCWalk (1 s cutoff time)

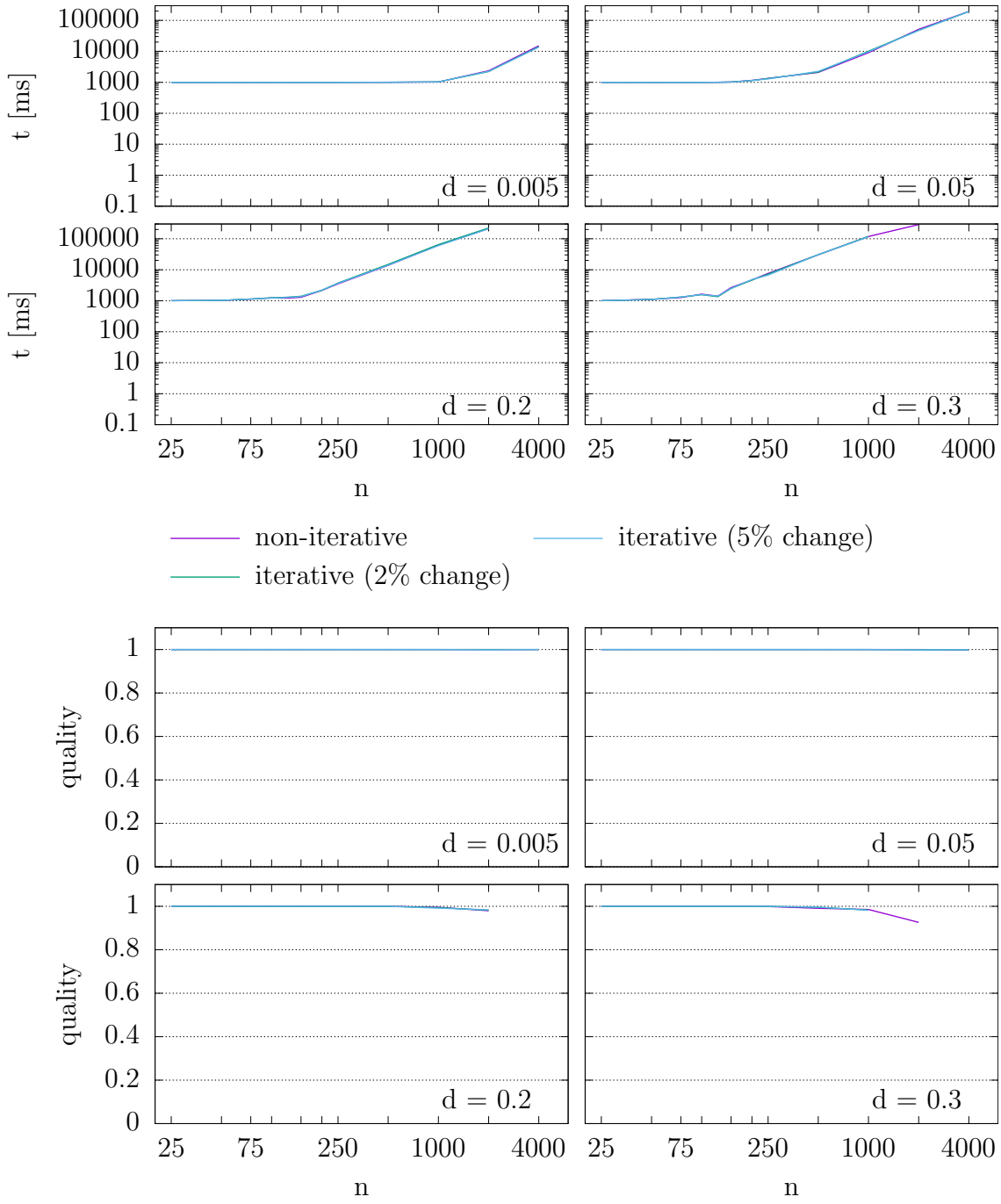

Figure 7.27: Measurements of iterative runs of SCCWalk (1 s cutoff time, other densities)

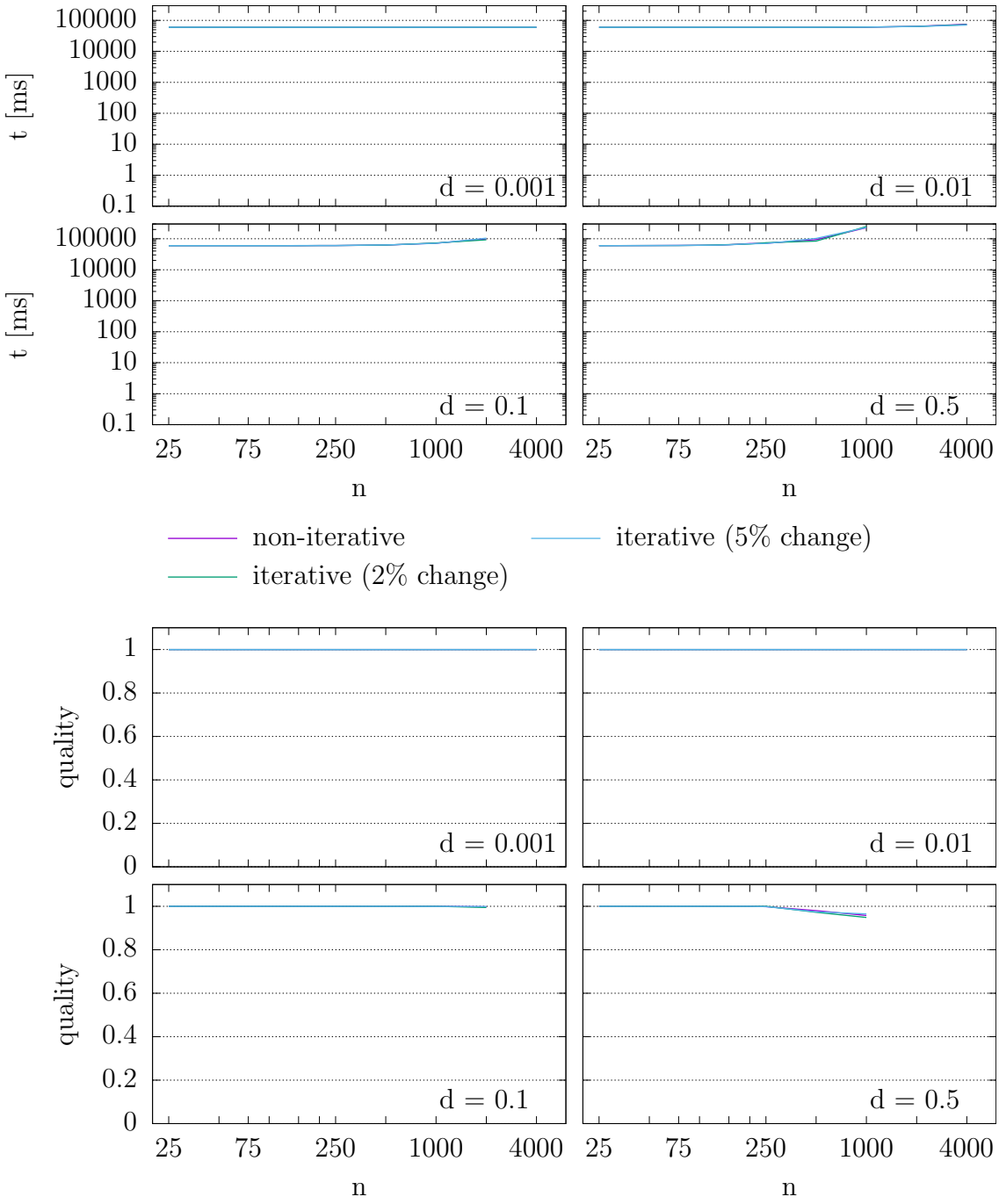

Figure 7.28: Measurements of iterative runs of SCCWalk (60 s cutoff time)

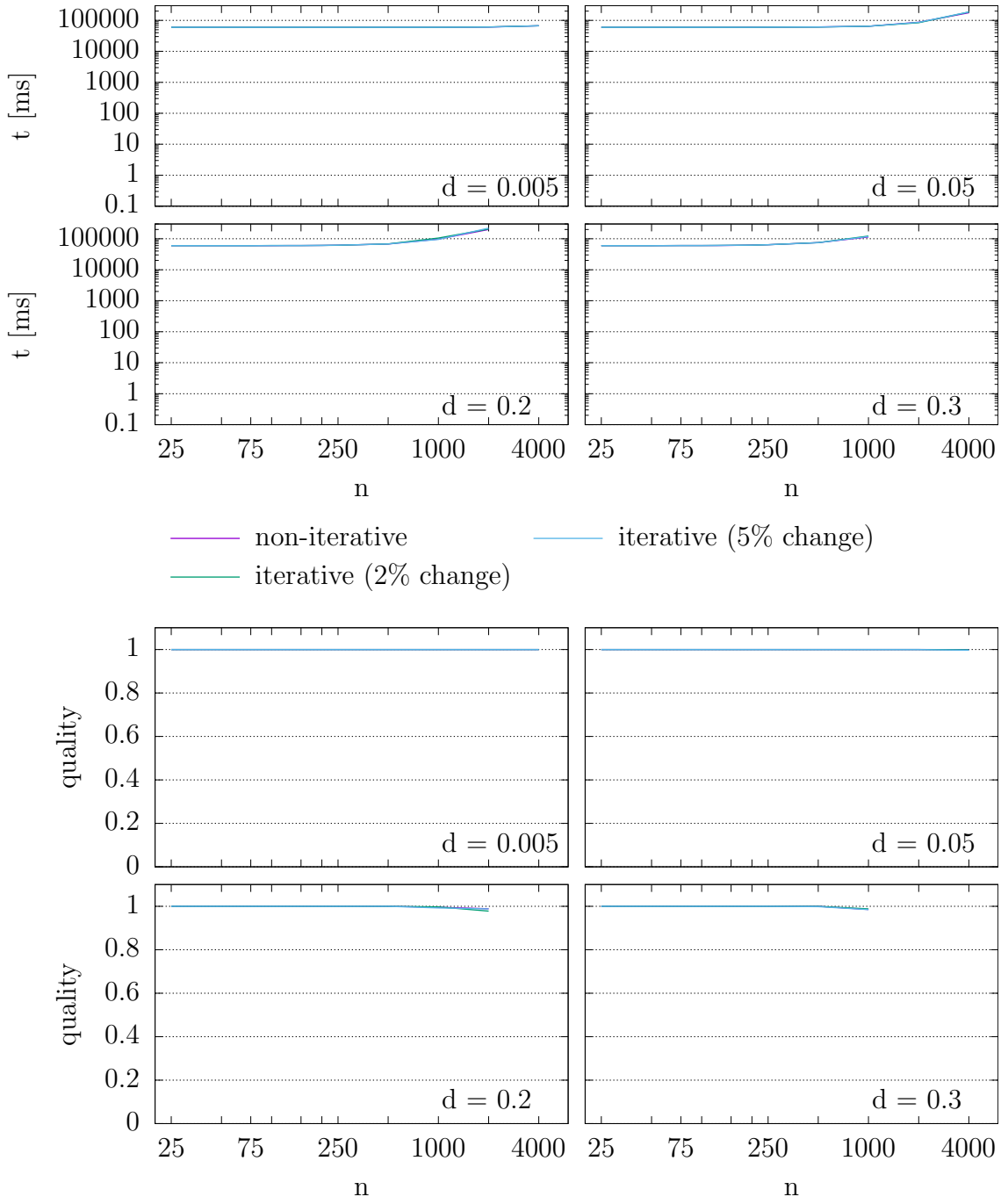

Figure 7.29: Measurements of iterative runs of SCCWalk (60 s cutoff time, other densities)

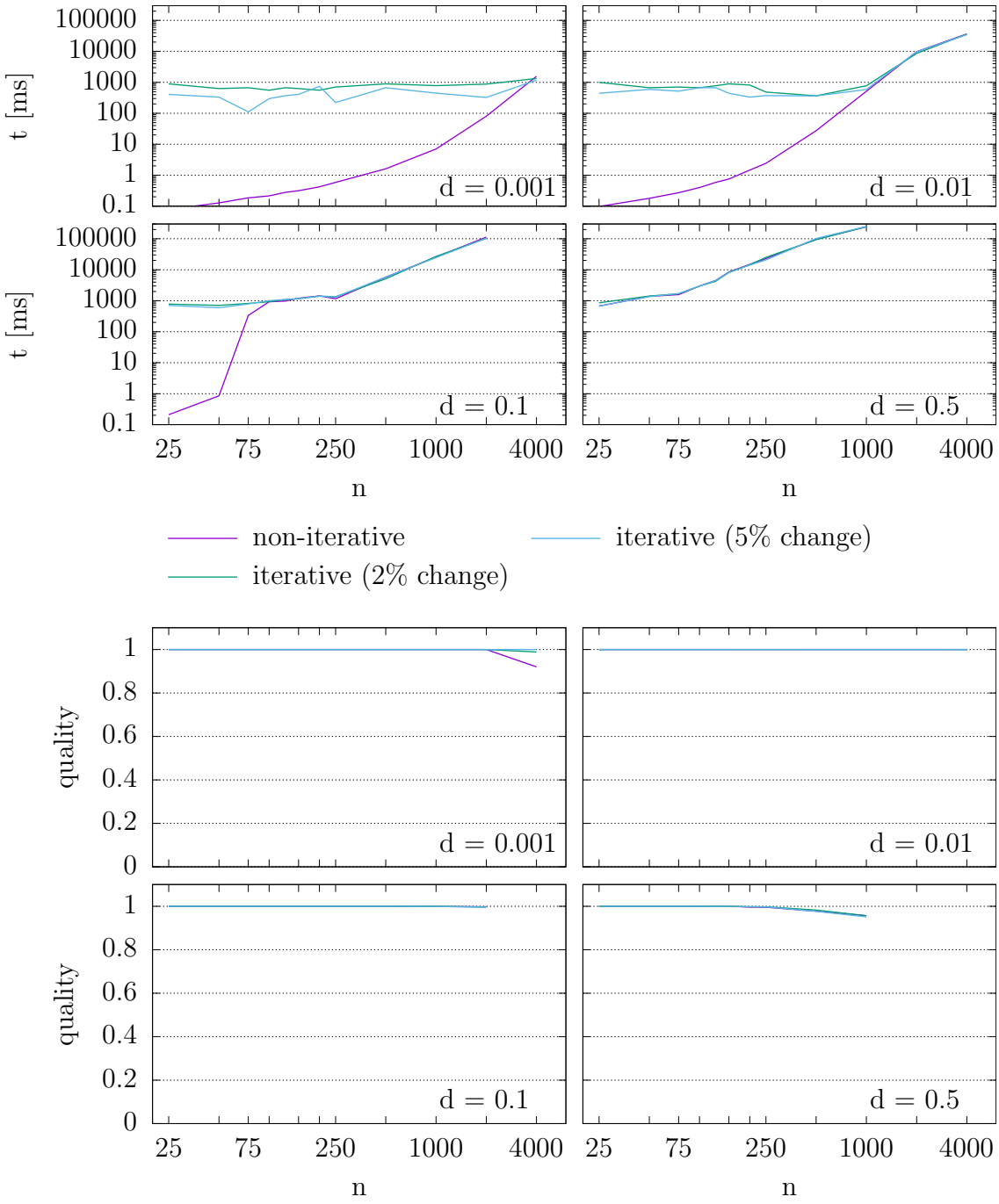

Figure 7.30: Measurements of iterative runs of SCCWalk4L (1 s cutoff time)

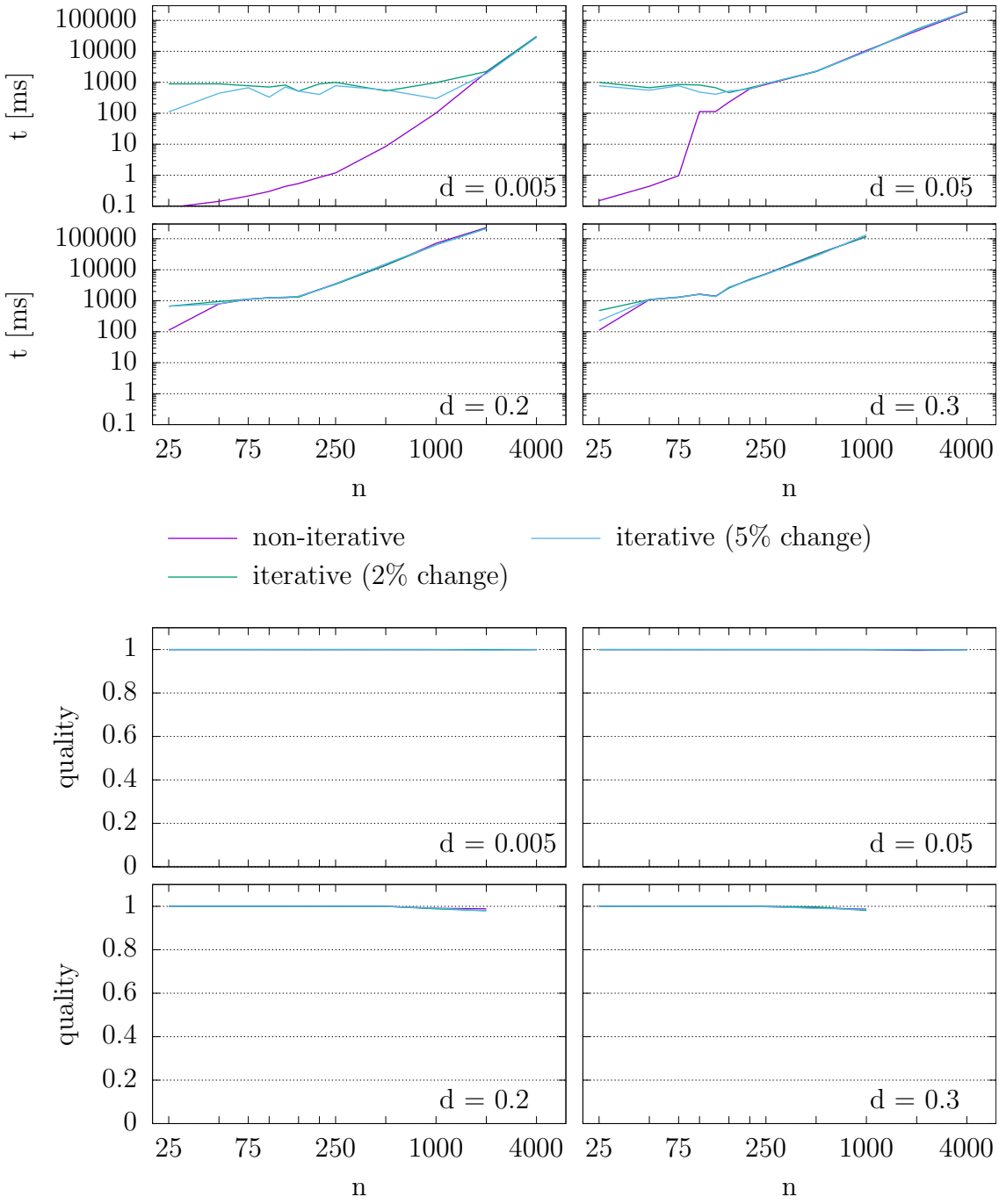

Figure 7.31: Measurements of iterative runs of SCCWalk4L (1 s cutoff time, other densities)

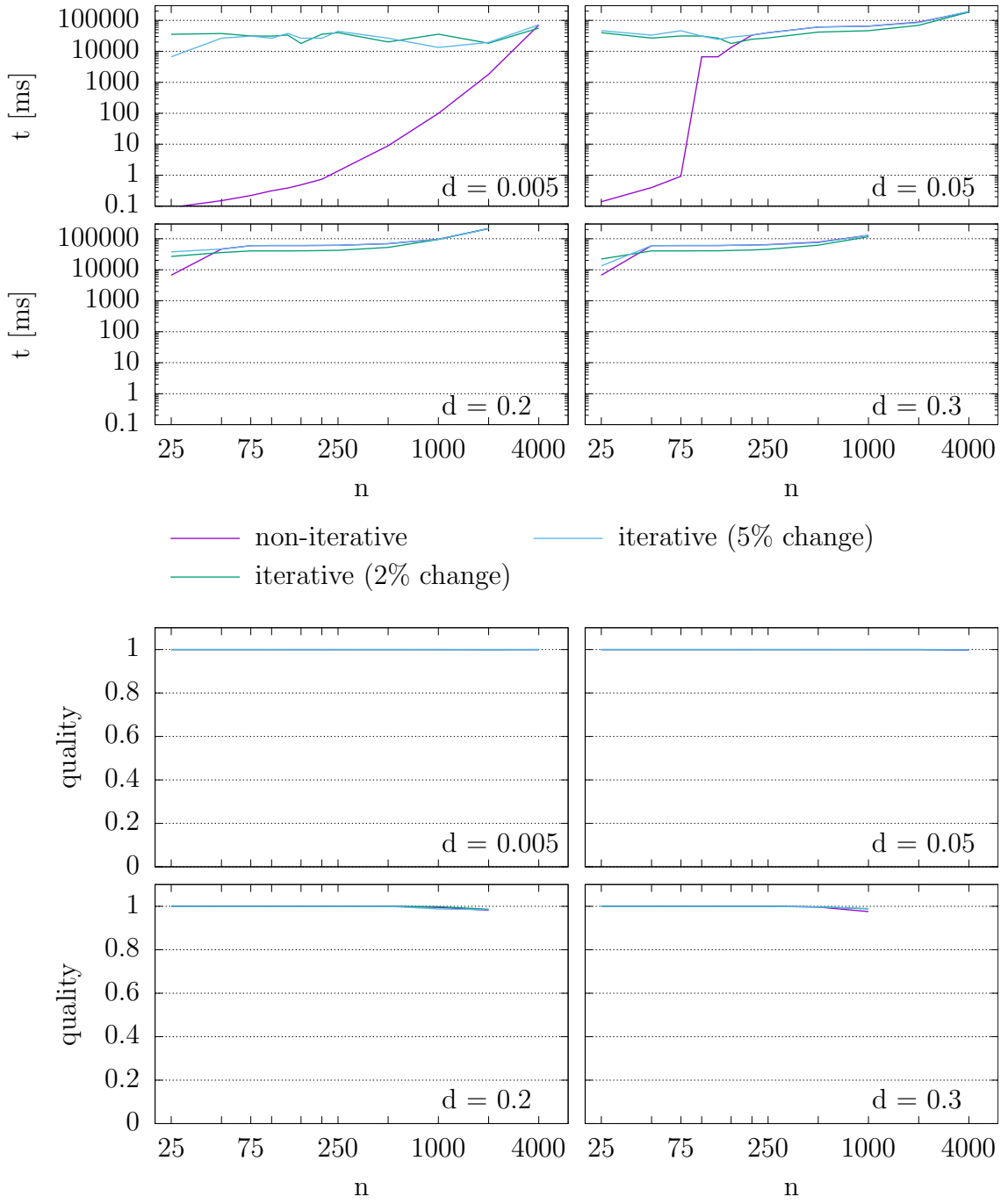

Figure 7.32: Measurements of iterative runs of SCCWalk4L (60 s cutoff time, other densities)

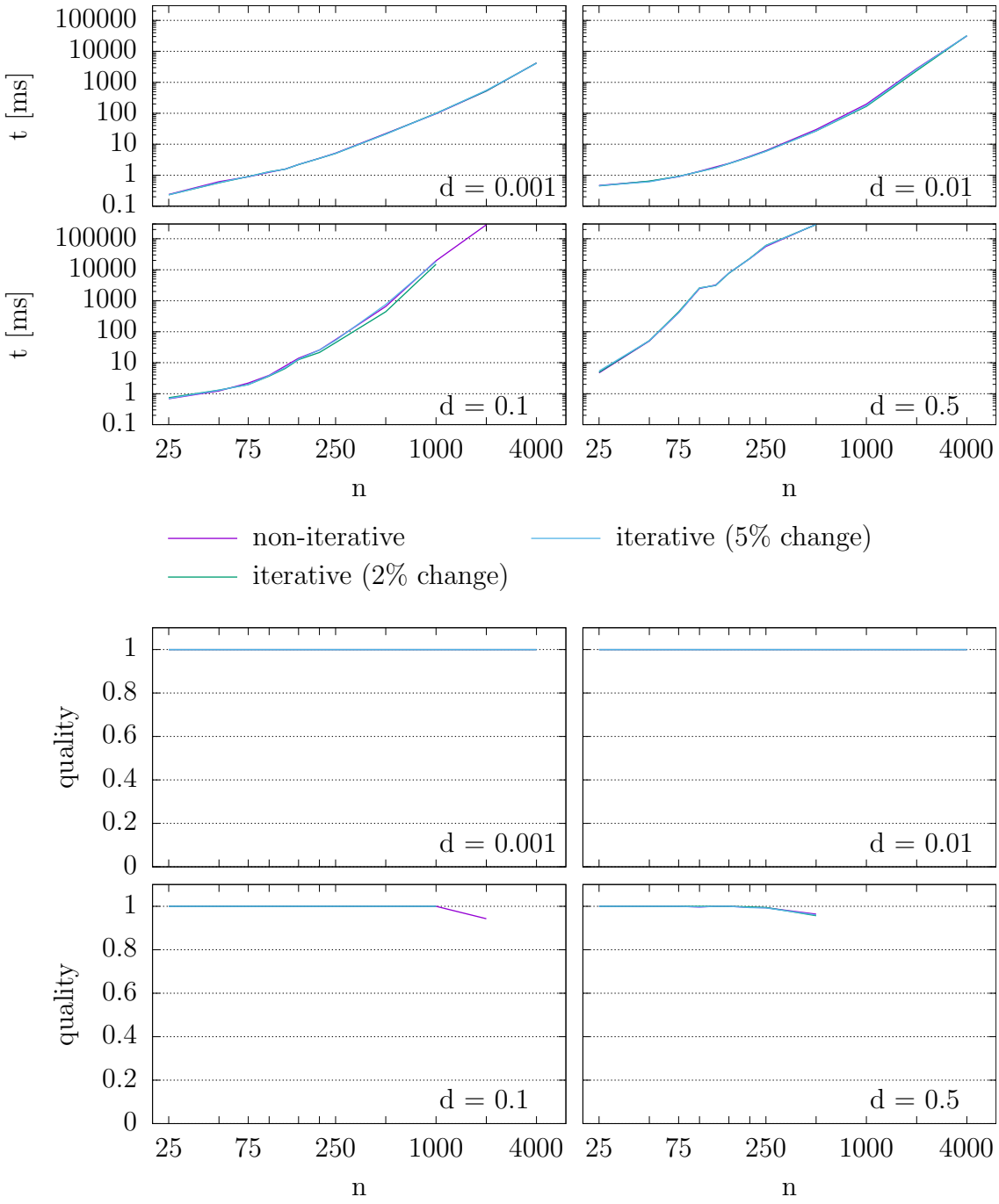

Figure 7.33: Measurements of iterative runs of MWCPeel

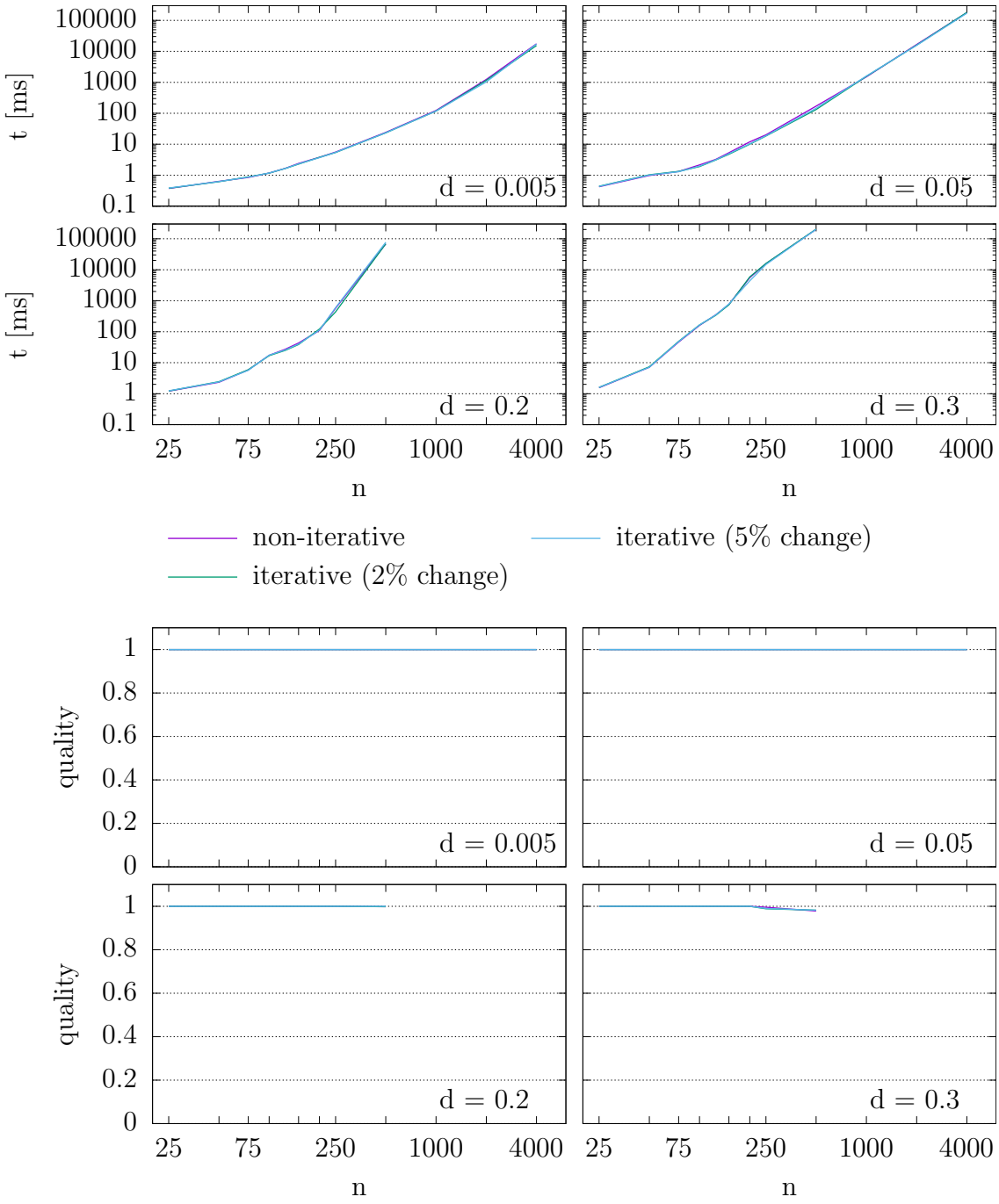

Figure 7.34: Measurements of iterative runs of MWCPeel (other densities)

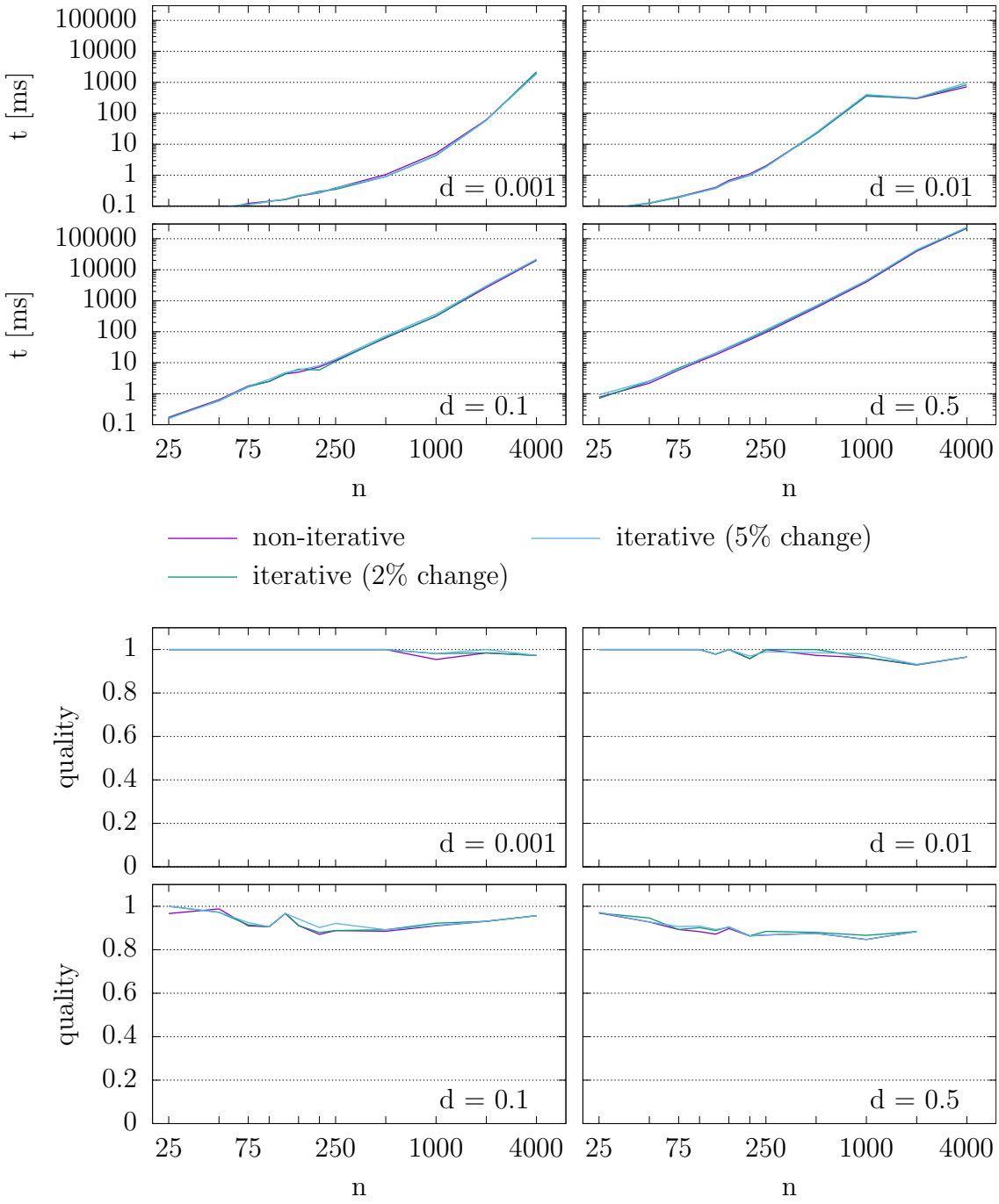

Figure 7.35: Measurements of iterative runs of UEW-R ($\alpha = 1$)

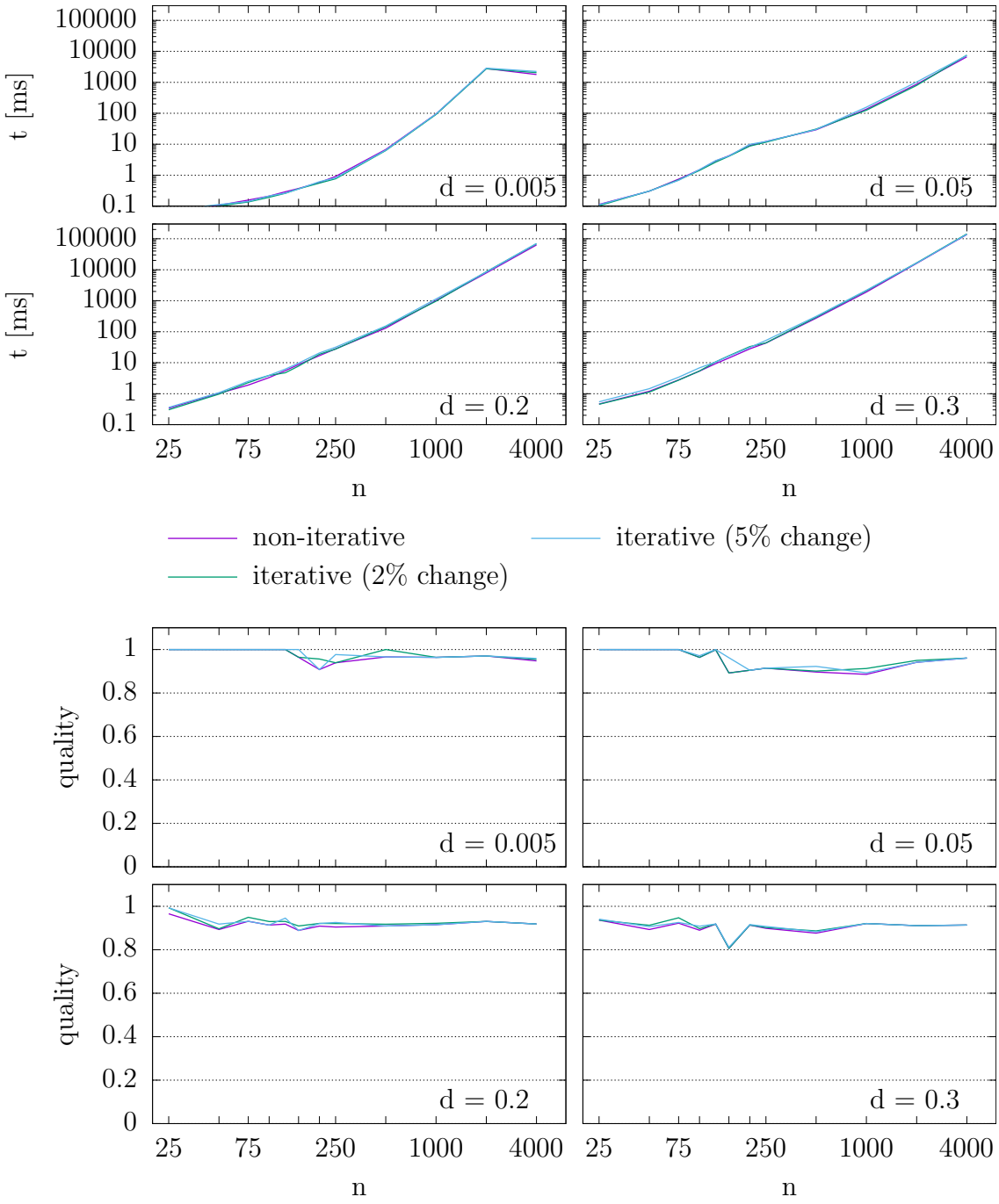

Figure 7.36: Measurements of iterative runs of UEW-R ($\alpha = 1$, other densities)

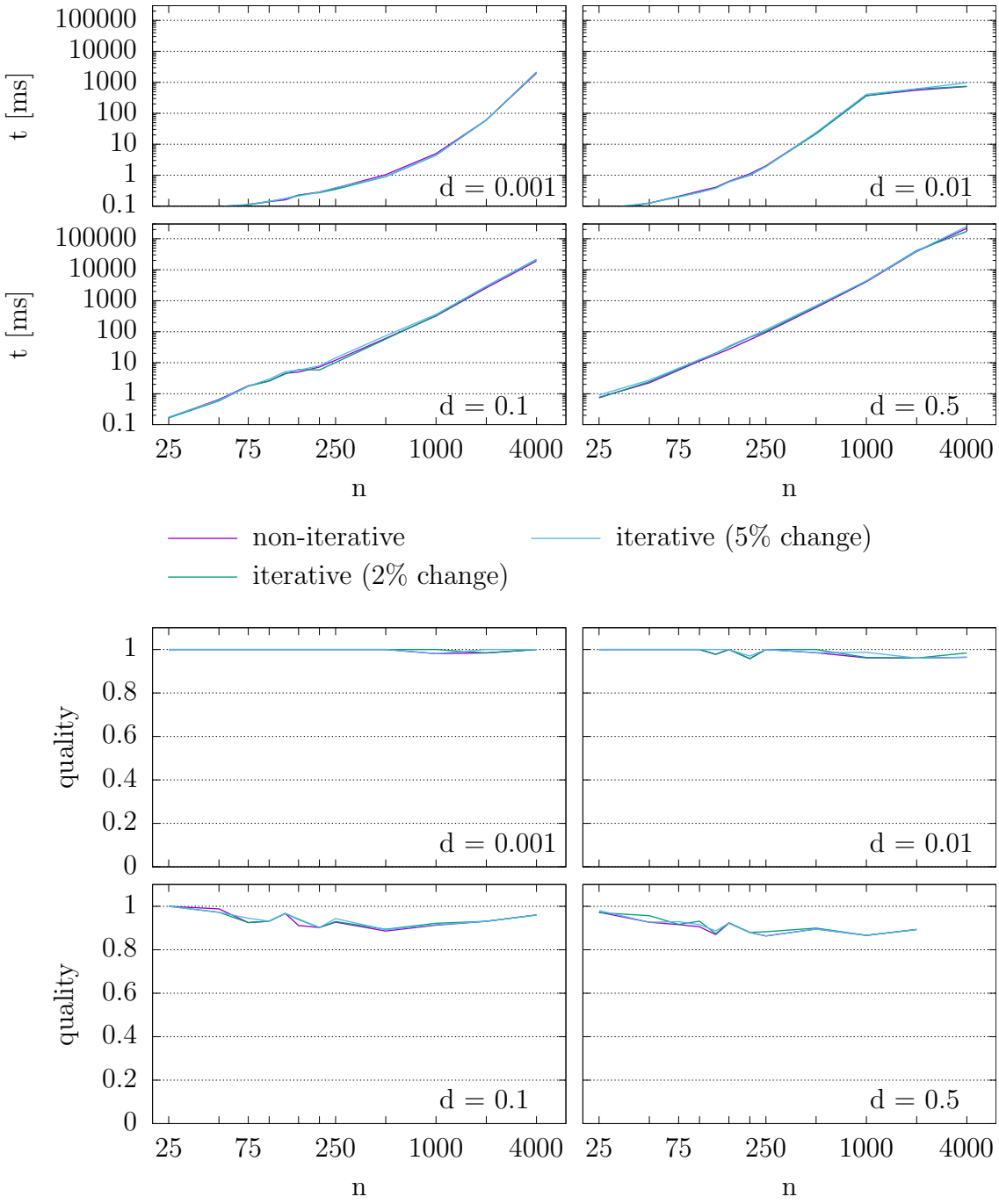

Figure 7.37: Measurements of iterative runs of UEW-R ($\alpha = 2$)

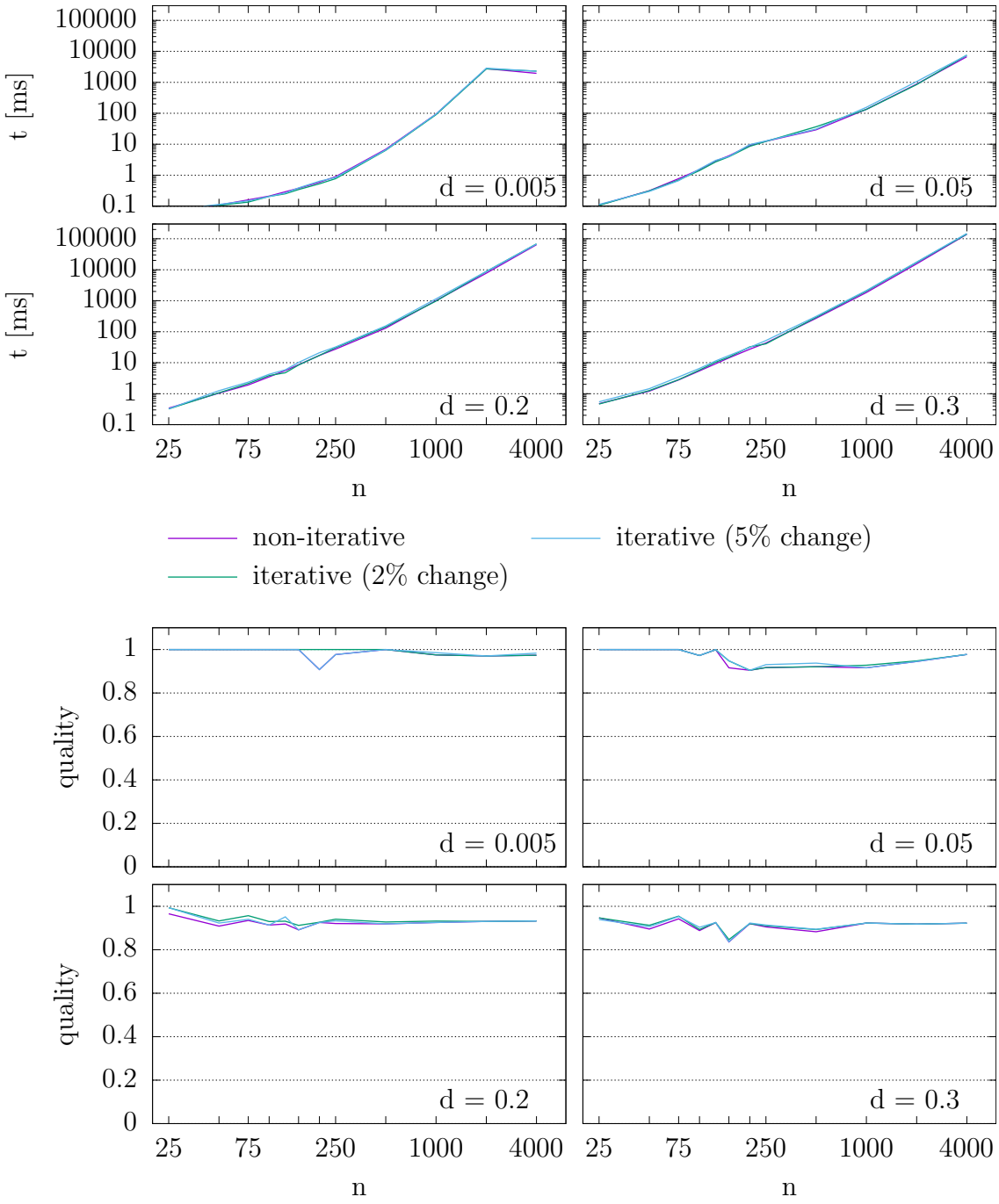

Figure 7.38: Measurements of iterative runs of UEW-R ($\alpha = 2$, other densities)

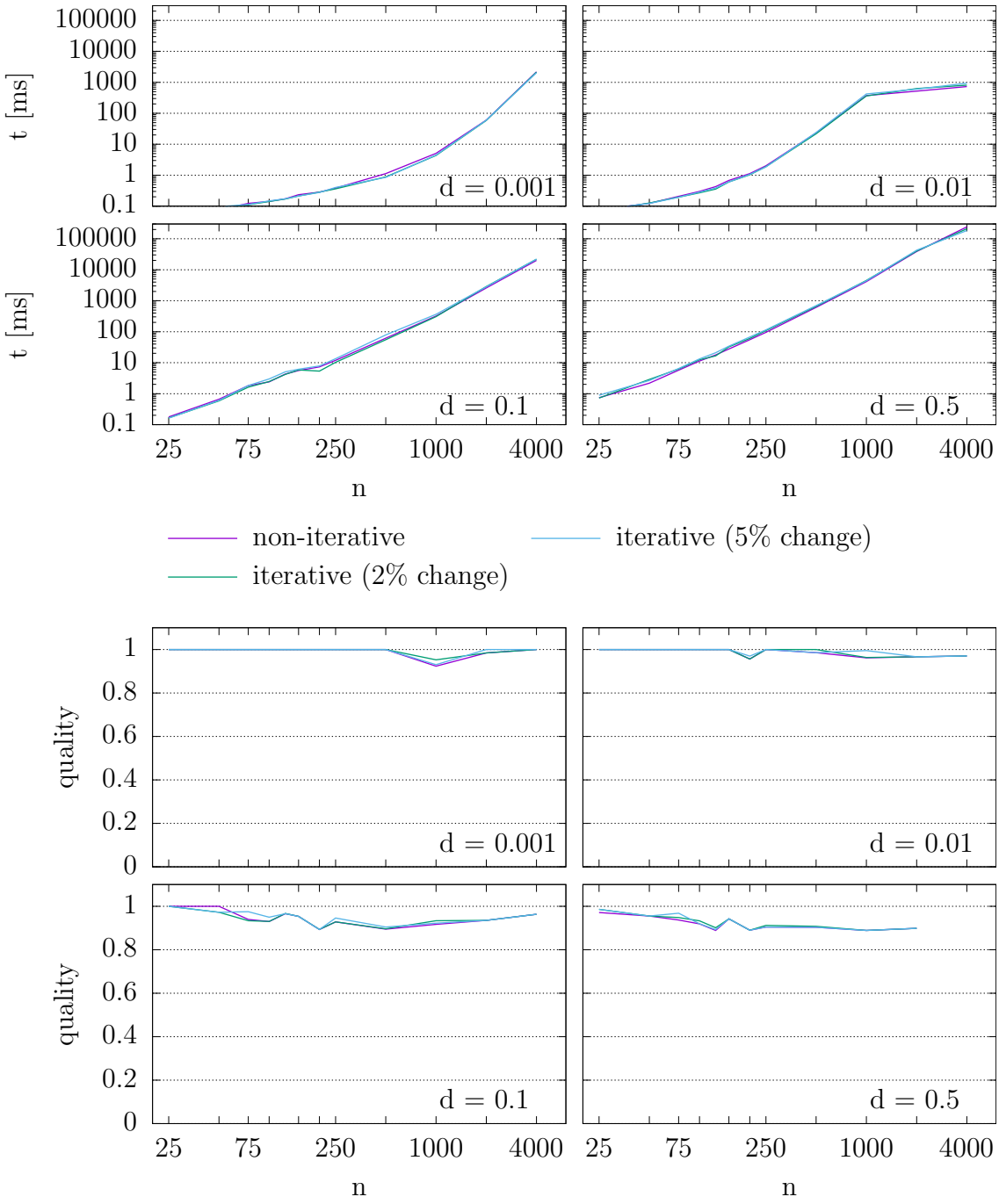

Figure 7.39: Measurements of iterative runs of UEW-R ($\alpha = 4$)

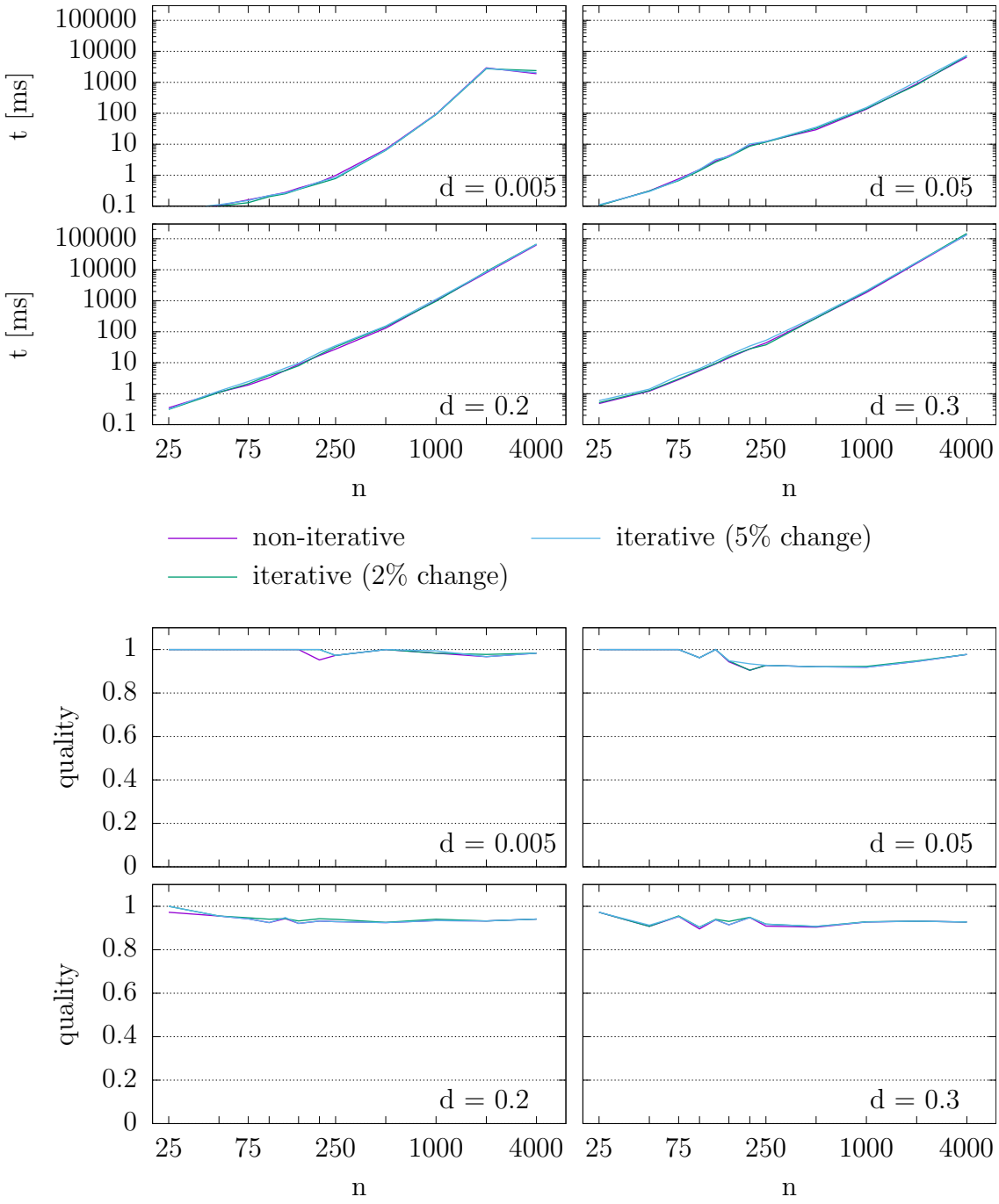

Figure 7.40: Measurements of iterative runs of UEW-R ($\alpha = 4$, other densities)

Internal sources

[10] M. Vodel and W. Hardt, "Energy-efficient communication in distributed, embedded systems," in *Proceedings of the WiOpt 13 - RAWNET/WNC3*, Tokio, Japan: IEEE Computer Society, May 2013, pp. 641–647, ISBN: 978-3-901882-54-8.

[11] Vodel, M. and Hardt, W. (Hrsg.), *Energieeffiziente Kommunikation in verteilten, eingebetteten Systemen* (Wissenschaftliche Schriftenreihe Eingebettete Selbstorganisierende Systeme), 13th ed. Universitätsverlag Chemnitz, Jan. 2014, ISBN: 978-3-944640-05-1. [Online]. Available: `http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-133410`.

[12] D. Reißner, M. Caspar, and W. Hardt, "Energy-efficient adaptive communication by preference-based routing and forecasting," ser. Multi-Conference on Systems, Signals & Devices (SSD), Technische Universität Chemnitz, Fakultät für Informatik Professur Technische Informatik, IEEE Computer Society, Feb. 2014, pp. 1–6. [Online]. Available: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6808912&isnumber=6808745`.

[13] R. Bergelt and W. Hardt, "An event-based local action paradigm to improve energy efficiency in queriable wireless sensor actuator networks," *Journal of Communications Software and Systems*, vol. 16, no. 1, pp. 66–74, Mar. 2020, ISSN: 1845-6421. [Online]. Available: `https://doi.org/10.24138/jcomss.v16i1.1021`.

[14] W. Hardt and Camposano, "Specification analysis for hw/sw-partitioning," in *3. GI/ITG Workshop: Anwendung formaler Methoden für den Hardware-Entwurf*, Passau: Shaker Verlag, Mar. 1995, pp. 1–10.

[15] R. Schmidt, S. Blokzyl, and W. Hardt, "Hardware acceleration for beamforming algorithms based on optimized hardware-/software partitioning," in *European Test and Telemetry Conference (2018 : Nürnberg)*, 2018, ISBN: 978-3-9816876-8-2.

[16] T. Meier, M. Ernst, A. Frey, and W. Hardt, "Enhancing task assignment in many-core systems by a situation aware scheduler," in *embedded world Conference 2017, 14-16.03.2017, Nürnberg. - WEKA FACHMEDIEN GmbH, 2017*, TU Chemnitz, Mar. 2017. [Online]. Available: `http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-227009`.

Bibliography

[1] D. Zhang, O. Javed, and M. Shah, "Video object co-segmentation by regulated maximum weight cliques," Sep. 2014, pp. 551–566, ISBN: 978-3-319-10583-3. DOI: 10.1007/978-3-319-10584-0_36.

[2] J. Pattillo, N. Youssef, and S. Butenko, "On clique relaxation models in network analysis," *European Journal of Operational Research*, vol. 226, no. 1, pp. 9–18, 2013, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2012.10.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221712007679.

[3] B. Balasundaram and S. Butenko, "Graph domination, coloring and cliques in telecommunications," in *Handbook of Optimization in Telecommunications*. Boston, MA: Springer US, 2006, pp. 865–890, ISBN: 978-0-387-30165-5. DOI: 10.1007/978-0-387-30165-5_30. [Online]. Available: https://doi.org/10.1007/978-0-387-30165-5_30.

[4] F. Mascia, E. Cilia, M. Brunato, and A. Passerini, "Predicting structural and functional sites in proteins by searching for maximum-weight cliques," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, pp. 1274–1279, Jul. 2010. DOI: 10.1609/aaai.v24i1.7495. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/7495.

[5] S. Butenko and W. Wilhelm, "Clique-detection models in computational biochemistry and genomics," *European Journal of Operational Research*, vol. 173, no. 1, pp. 1–17, 2006, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2005.05.026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221705005266.

[6] Y. Wang, S. Cai, J. Chen, and M. Yin, "Sccwalk: An efficient local search algorithm and its improvements for maximum weight clique problem," *Artificial Intelligence*, vol. 280, p. 103 230, 2020, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2019.103230. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370219302164.

[7] S. Cai and J. Lin, "Fast solving maximum weight clique problem in massive graphs," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, IJCAI/AAAI Press, 2016, pp. 568–574. [Online]. Available: http://www.ijcai.org/Abstract/16/087.

[8] H. Jiang, C.-M. Li, Y. Liu, and F. Manyà, "A two-stage maxsat reasoning approach for the maximum weight clique problem," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, Apr. 2018. DOI: 10.1609/aaai.v32i1.11527. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/11527.

[9] C. McCreesh, P. Prosser, K. A. Simpson, and J. Trimble, "On maximum weight clique algorithms, and how they are evaluated," in *International Conference on Principles and Practice of Constraint Programming*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:26546945.

[10] M. Vodel and W. Hardt, "Energy-efficient communication in distributed, embedded systems," in *Proceedings of the WiOpt 13 - RAWNET/WNC3*, Tokio, Japan: IEEE Computer Society, May 2013, pp. 641–647, ISBN: 978-3-901882-54-8.

[11] Vodel, M. and Hardt, W. (Hrsg.), *Energieeffiziente Kommunikation in verteilten, eingebetteten Systemen* (Wissenschaftliche Schriftenreihe Eingebettete Selbstorganisierende Systeme), 13th ed. Universitätsverlag Chemnitz, Jan. 2014, ISBN: 978-3-944640-05-1. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-133410.

[12] D. Reißner, M. Caspar, and W. Hardt, "Energy-efficient adaptive communication by preference-based routing and forecasting," ser. Multi-Conference on Systems, Signals & Devices (SSD), Technische Universität Chemnitz, Fakultät für Informatik Professur Technische Informatik, IEEE Computer Society, Feb. 2014, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6808912&isnumber=6808745.

[13] R. Bergelt and W. Hardt, "An event-based local action paradigm to improve energy efficiency in queriable wireless sensor actuator networks," *Journal of Communications Software and Systems*, vol. 16, no. 1, pp. 66–74, Mar. 2020, ISSN: 1845-6421. [Online]. Available: https://doi.org/10.24138/jcomss.v16i1.1021.

[14] W. Hardt and Camposano, "Specification analysis for hw/sw-partitioning," in *3. GI/ITG Workshop: Anwendung formaler Methoden für den Hardware-Entwurf*, Passau: Shaker Verlag, Mar. 1995, pp. 1–10.

[15] R. Schmidt, S. Blokzyl, and W. Hardt, "Hardware acceleration for beamforming algorithms based on optimized hardware-/software partitioning," in *European Test and Telemetry Conference (2018 : Nürnberg)*, 2018, ISBN: 978-3-9816876-8-2.

[16] T. Meier, M. Ernst, A. Frey, and W. Hardt, "Enhancing task assignment in many-core systems by a situation aware scheduler," in *embedded world Conference 2017, 14-16.03.2017, Nürnberg. - WEKA FACHMEDIEN GmbH, 2017*, TU Chemnitz, Mar. 2017. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-227009.

[17] R. Erhardt, K. Hanauer, N. Kriege, C. Schulz, and D. Strash, "Improved exact and heuristic algorithms for maximum weight clique," Feb. 2023. DOI: 10.48550/arXiv.2302.00458.

[18] C.-M. Li, Y. Liu, H. Jiang, F. Manyà, and Y. Li, "A new upper bound for the maximum weight clique problem," *European Journal of Operational Research*, vol. 270, no. 1, pp. 66–77, 2018, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2018.03.020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221718302297.

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*, First Edition. W. H. Freeman, 1979, ISBN: 0716710455. [Online]. Available: http://www.amazon.com/Computers-Intractability-NP-Completeness-Mathematical-Sciences/dp/0716710455.

[20] H. Jiang, C.-M. Li, and F. Manyà, "An exact algorithm for the maximum weight clique problem in large graphs," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, Feb. 2017. DOI: 10.1609/aaai.v31i1.10648. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/10648.

[21] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960, ISSN: 00129682, 14680262. [Online]. Available: http://www.jstor.org/stable/1910129.

[22] S. Cai, J. Lin, Y. Wang, and D. Strash, "A semi-exact algorithm for quickly computing a maximum weight clique in large sparse graphs," *J. Artif. Int. Res.*, vol. 72, pp. 39–67, Jan. 2022, ISSN: 1076-9757. DOI: 10.1613/jair.1.12327. [Online]. Available: https://doi.org/10.1613/jair.1.12327.

[23] R. Erhardt, "Engineering algorithms for the weighted maximum clique problem," M.S. thesis, Heidelberg University, 2022.

[24] P. R. J. Östergård, "A new algorithm for the maximum-weight clique problem," *Nordic J. of Computing*, vol. 8, no. 4, pp. 424–436, Dec. 2001, ISSN: 1236-6064.

[25] D. Kumlander, "A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search," *Proceedings of the Fourth International Conference on Engineering Computational Technology*, Jan. 2004. DOI: 10.4203/ccp.80.60.

[26] Z. Fang, C.-M. Li, K. Qiao, X. Feng, and K. Xu, "Solving maximum weight clique using maximum satisfiability reasoning," *Frontiers in Artificial Intelligence and Applications*, vol. 263, Jan. 2014. DOI: 10.3233/978-1-61499-419-0-303.

[27] S. Shimizu, K. Yamaguchi, T. Saitoh, and S. Masuda, "Fast maximum weight clique extraction algorithm: Optimal tables for branch-and-bound," *Discrete Applied Mathematics*, vol. 223, pp. 120–134, 2017, ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2017.01.026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166218X1730063X.

[28] Y. Sun, X. Li, and A. Ernst, "Using statistical measures and machine learning for graph reduction to solve maximum weight clique problems," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 5, pp. 1746–1760, 2021. DOI: 10.1109/TPAMI.2019.2954827.

[29] C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *Commun. ACM*, vol. 16, no. 9, pp. 575–577, Sep. 1973, ISSN: 0001-0782. DOI: 10.1145/362342.362367. [Online]. Available: https://doi.org/10.1145/362342.362367.

[30] The Qt Company, *Qt — tools for each stage of software development lifecylce*, https://www.qt.io, Accessed: 2024-02-18.

[31] S. C. North *et al.*, *Graphviz*, https://graphviz.org, Accessed: 2024-02-18.

[32] A. Kapoulkine, *Pugixml, light-weight, simple and fast xml parser for c++ with xpath support*, https://pugixml.org, Accessed: 2024-02-18.

[33] C.-M. Li, *Englishpage*, https://home.mis.u-picardie.fr/~cli/EnglishPage.html, Accessed: 2024-01-14.

[34] S. Cai, *Maximum (weight) clique problem solving*, https://lcs.ios.ac.cn/~caisw/CLQ.html, Accessed: 2024-01-14.

[35] Raspberry Pi Ltd., *Raspberry pi 4 model b specifications - raspberry pi*, https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/, Accessed: 2024-02-16.

[36] Raspberry Pi Ltd., *Operating system images - raspberry pi*, https://www.raspberrypi.com/software/operating-systems/, Accessed: 2024-02-16.

Chemnitzer Informatik-Berichte

In der Reihe der Chemnitzer Informatik-Berichte sind folgende Berichte erschienen:

CSR-20-01 Danny Kowerko, Chemnitzer Linux-Tage 2019 - LocalizeIT Workshop, Januar 2020, Chemnitz

CSR-20-02 Robert Manthey, Tom Kretzschmar, Falk Schmidsberger, Hussein Hussein, René Erler, Tobias Schlosser, Frederik Beuth, Marcel Heinz, Thomas Kronfeld, Maximilian Eibl, Marc Ritter, Danny Kowerko, Schlussbericht zum InnoProfile-Transfer Begleitprojekt localizeI, Januar 2020, Chemnitz

CSR-20-03 Jörn Roth, Reda Harradi und Wolfram Hardt, Indoor Lokalisierung auf Basis von Ultra Wideband Modulen zur Emulation von GPS Positionen, Februar 2020, Chemnitz

CSR-20-04 Christian Graf, Reda Harradi, René Schmidt, Wolfram Hardt, Automatisierte Kameraausrichtung für Micro Air Vehicle basierte Inspektion, März 2020, Chemnitz

CSR-20-05 Julius Lochbaum, René Bergelt, Time Pech, Wolfram Hardt, Erzeugung von Testdaten für automatisiertes Fahren auf Basis eines Open Source Fahrsimulators, März 2020, Chemnitz

CSR-20-06 Narankhuu Natsagdorj, Uranchimeg Tudevdagva, Jiantao Zhou, Logical Structure of Structure Oriented Evaluation for E-Learning, April 2020, Chemnitz

CSR-20-07 Batbayar Battseren, Reda Harradi, Fatih Kilic, Wolfram Hardt, Automated Power Line Inspection, September 2020, Chemnitz

CSR-21-01 Marco Stephan, Batbayar Battseren, Wolfram Hardt, UAV Flight using a Monocular Camera, März 2021, Chemnitz

CSR-21-02 Hasan Aljzaere, Owes Khan, Wolfram Hardt, Adaptive User Interface for Automotive Demonstrator, Juli 2021, Chemnitz

CSR-21-03 Chibundu Ogbonnia, René Bergelt, Wolfram Hardt, Embedded System Optimization of Radar Post-processing in an ARM CPU Core, Dezember 2021, Chemnitz

CSR-21-04 Julius Lochbaum, René Bergelt, Wolfram Hardt, Entwicklung und Bewertung von Algorithmen zur Umfeldmodellierung mithilfe von Radarsensoren im Automotive Umfeld, Dezember 2021, Chemnitz

Chemnitzer Informatik-Berichte

CSR-22-01 Henrik Zant, Reda Harradi, Wolfram Hardt, Expert System-based Embedded Software Module and Ruleset for Adaptive Flight Missions, September 2022, Chemnitz

CSR-23-01 Stephan Lede, René Schmidt, Wolfram Hardt, Analyse des Ressourcenverbrauchs von Deep Learning Methoden zur Einschlagslokalisierung auf eingebetteten Systemen, Januar 2023, Chemnitz

CSR-23-02 André Böhle, René Schmidt, Wolfram Hardt, Schnittstelle zur Datenakquise von Daten des Lernmanagementsystems unter Berücksichtigung bestehender Datenschutzrichtlinien, Januar 2023, Chemnitz

CSR-23-03 Falk Zaumseil, Sabrina, Bräuer, Thomas L. Milani, Guido Brunnett, Gender Dissimilarities in Body Gait Kinematics at Different Speeds, März 2023, Chemnitz

CSR-23-04 Tom Uhlmann, Sabrina Bräuer, Falk Zaumseil, Guido Brunnett, A Novel Inexpensive Camera-based Photoelectric Barrier System for Accurate Flying Sprint Time Measurement, März 2023, Chemnitz

CSR-23-05 Samer Salamah, Guido Brunnett, Sabrina Bräuer, Tom Uhlmann, Oliver Rehren, Katharina Jahn, Thomas L. Milani, Güunter Daniel Rey, NaturalWalk: An Anatomy-based Synthesizer for Human Walking Motions, März 2023, Chemnitz

CSR-24-01 Seyhmus Akaslan, Ariane Heller, Wolfram Hardt, Hardware-Supported Test Environment Analysis for CAN Message Communication, Juni 2024, Chemnitz

CSR-24-02 S. M. Rizwanur Rahman, Wolfram Hardt, Image Classification for Drone Propeller Inspection using Deep Learning, August 2024, Chemnitz

CSR-24-03 Sebastian Pettke, Wolfram Hardt, Ariane Heller, Comparison of maximum weight clique algorithms, August 2024, Chemnitz

Chemnitzer Informatik-Berichte

ISSN 0947-5125

Herausgeber: Fakultät für Informatik, TU Chemnitz
Straße der Nationen 62, D-09111 Chemnitz