UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

Solving the Maximum Weight Independent Set Problem

Application to Indirect Hex-Mesh Generation

Dissertation presented by
Kilian VERHETSEL

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Jean-François REMACLE, Amaury JOHNEN

Reader(s)
Jeanne PELLERIN, Yves DEVILLE , Julien HENDRICKX

Academic year 2016-2017

Acknowledgments

I would like to thank my supervisor, Jean-François Remacle, for believing in me, my co-supervisor Amaury Johnen, for his encouragements during this project, as well as Jeanne Pellerin, for her patience and helpful advice.

Contents

List of Notations

List of Mathematical Notations

G	A graph
$\overline{G}$	The complementary graph of G.
V	The set of vertices of a graph.
E	The set of edges of a graph.
u, v	A vertex.
v_i	A vertex whose index is i.
w_i	The weight of vertex v_i.
x_i	A variable related to vertex v_i.
S	An independent set.
C	A clique.
U, P, Q, X, Y	Some set of vertices.
$\mathcal{C}$	A clique cover of a graph.
$N_G(v)$	The set of vertices adjacent to v in graph G
$\deg(v)$	The degree of a vertex v, i.e. the number of vertices adjacent to v.
G_X	The induced subgraph of a subset X of the vertices of G. X is the vertex set of G_X and the edges that connect two vertices of X are its edge set.
W	The total weight of an independent set.
S^*	The best known independent set.
W^*	The total weight of the best known independent set.
$\mathcal{S}$	The set of all solutions to a problem
$\mathcal{N}$	A neighborhood function.
δ_A	The reward obtained by performing some action A.
T	In the context of a local search, a tabu list.
$\|S\|$	Cardinality of a set S.
$\neg x_i$	Logical negation of a variable
$x_i \vee x_j$	Logical disjunction of two Boolean expressions

List of Abbreviations and Initialisms

BMS	Best from multiple selections
CLS	Cooperating local search
LNS	Large Neighborhood Search
LP	Linear Programming
MaxSAT	Maximum satisfiability
MWCP	Maximum weight clique problem
MWIS	Maximum weight independent set
NP	Non-deterministic polynomial
OOM	Out of memory
RLS	Reactive local search

List of Figures

List of Tables

List of Algorithms

Abstract

We propose a heuristic method to find independent sets of large weight in graphs, used as part of an algorithm for indirect generation of hex-dominant meshes. Our method computes an initial solution using a local search algorithm. This algorithm was implemented using priority queues to select which action should be performed at each iteration in order to improve its performances. The initial solution is then iteratively improved by finding the optimal solution of the problem for subgraphs of up to a few hundred vertices. For this purpose, we use a branch and bound solver. This solver partitions the vertex set of the graph into cliques at each node of the search tree. This partition allows for the computation of an upper bound on the solution, which is used to reduce the number of solutions enumerated by the algorithm.

We ran our algorithm on graphs of more than one million vertices, and showed that it produces better results than the greedy methods used in existing indirect meshing algorithms.

Chapter 1

Introduction

Partial differential equations are used to model a variety of physical phenomena. In order to solve them, numerical methods such as finite elements methods are often used. These methods require a mesh to represent the domain on which the equation needs to be solved.

Meshes subdivide this domain in a set of discrete elements. They can be classified according to which type of geometric object is used to represent each cell. For example, *tetrahedral meshes* contain *tetrahedra*, *hexahedral meshes* are made up of hexahedra, and *hex-dominant meshes* primarily contain hexahedra but also a mixture of tetrahedra, prisms and pyramids.

Although hexahedral meshes are preferred for some applications, completely automated methods to generate them still need to deal with several open problems in mesh generation [36]. One approach, which has the benefit of being applicable to any geometry for which a tetrahedral mesh can be constructed, is to start from a tetrahedral mesh and to combine tetrahedra in order to form hexahedra. This is known as an *indirect approach* to mesh generation.

The sections below give some additional context about indirect mesh generation (section 1.1), and outline the contents of this dissertation (section 1.2).

1.1 Background on Indirect Mesh Generation

Indirect mesh generation algorithms usually include, and differ in their approaches to, the following steps:

1. finding all potential ways to recombine tetrahedra in order to generate hexahedra and other elements;

2. selecting a subset of these combinations in order to generate a new mesh;

3. combining other tetrahedra in order to insert elements such as prisms and pyramids, which must be used in order to meet conformity requirements (such as the fact that two elements cannot share only half of a face, meaning a tetrahedron cannot be adjacent to a hexahedron).

The algorithm proposed by Meshkat and Talmor [28] is a typical example of such methods. The algorithm first identifies tetrahedra that can be combined in other elements by looking for patterns in the dual graph of the original mesh, i.e. a graph whose vertices represent cells of the mesh and whose edges represent faces. It then selects which combinations should be applied. This is done by iteratively selecting a combination that produce an element of highest quality compatible with all previously applied combinations.

Other indirect approaches rely on similar greedy algorithms [46] [36] [6]. Botella, Lévy, and Caumon [6] also noted that this problem can be reformulated in terms of a graph whose vertices represent recombined elements such as tetrahedra or prisms, and whose edges represent an

incompatibility between two elements (e.g. because they are both produced using a common tetrahedra). Each element is also given a weight, based on its geometric quality and its type, so that hexahedra have a higher weight than prisms and pyramids. Finding the best set of combinations can then be identified as being equivalent to finding a maximum weight independent set of this graph (see definition in chapter 2).

This problem is NP-Hard, a difficulty already noted in [28]. In addition to being computationally difficult, meshes can be very large, containing several millions of elements. These two factors explain the type of greedy methods used in prior work.

The next section outlines the presentation of our approach, which considers more elaborate tools which can be applied to the graphs encountered as part of indirect mesh generation. These methods, although more expensive, will allow better meshes to be produced by our approach.

1.2 Outline

We first detail existing exact and heuristic algorithm used to solve the MWIS problem (chapter 2). This gives context for our method, which we describe in chapter 3. The presentation of the algorithm first details the algorithm used to generate a graph from the input mesh (section 3.1). The rest of the chapter details how our algorithm solves the MWIS problem on this graph.

Our method combines two major components:

1. An existing exact resolution method, applied to graphs of up to a few hundred vertices (section 3.2).

2. A heuristic method to quickly compute a good solution (section 3.3). This method enhances an existing local search approach (described in [47]) with data structures allowing it to perform iterations more quickly.

Both of these algorithms are used as part of the algorithm we propose (section 3.4). This method improves a solution by iteratively solving the MWIS problem to optimality on small subgraphs.

The efficacy of this method and its components is evaluated in chapter 4. We show that we obtain better results than greedy methods within a few seconds even on graphs containing more than one million vertices.

Chapter 2

State of the Art

The problem of finding the best way to combine tetrahedra, as noted by Botella, Lévy, and Caumon [6], is equivalent to the maximum independent set problem, specifically its vertex-weighted variant. This problem can be formally stated as follows [42]: given a graph $G = (V, E)$ and for each vertex $v_i \in V$ a weight $w_i \geq 0$ find

$$\begin{aligned} &\max_{S \subseteq V} \sum_{v_i \in S} w_i \\ &\text{such that } \forall u, v \in S. (u, v) \notin E \end{aligned} \tag{MWIS}$$

This problem is closely related to the maximum weight clique problem (MWCP): given a graph $G = (V, E)$ and for each vertex $v_i \in V$ a weight $w_i \geq 0$ find

$$\begin{aligned} &\max_{C \subseteq V} \sum_{v_i \in C} w_i \\ &\text{such that } \forall u, v \in C. (u, v) \in E \end{aligned} \tag{MWCP}$$

The unweighted versions of these problems are equivalent to setting $w_i = 1$ for all vertices in the graph.

Finding a maximum independent set of a graph G is equivalent to finding a maximum clique of its complement graph $\overline{G}$, meaning that algorithms designed for either problem can be adapted to solve the other one.

Numerous algorithms have been proposed to solve these two problems. The next sections review the state of the art, both in terms of exact algorithms and heuristic algorithms.

2.1 Exact Resolution

The majority of exact algorithm developed to solve (MWIS) and (MWCP) are branch and bound algorithms. These algorithms iterate over potential solutions, in a way that guarantees that an optimal solution will always be found. They reduce the amount of visited solutions by computing an upper bound on subsets of the search space. When the upper bound determined by the algorithm is worse than the best known solution, solutions in the subset for which it was computed can be skipped by the solver [9]. Multiple families of upper bounding procedures and parallel exploration strategies have been developed to find the optimal solution more efficiently.

2.1.1 Approaches Based on Graph Coloring

Multiple strategies for the maximum clique problem have been developed based on the idea of coloring graphs. A graph coloring assigns a color to each vertex in the graph, such that two adjacent vertices always have a different color [17]. The set of all vertices of a given color is

known as a *color class*. A clique contains at most one vertex from each color class. Babel and Tinhofer [3] introduced a branch and bound algorithm which relies on this fact. DSATUR [7], a greedy algorithm that colors graphs, is used to compute an upper bound on the maximum size of a clique at every node of the search tree.

Tomita and Seki [38] improved this approach in an algorithm called MCQ. They used a greedy coloring algorithm which iterates over each vertex v, and tries to add it to the first color class (labeled 0), then the second one (labeled 1), etc. until it finds a color class to which v can be added. They noted that vertices whose color labels were higher were more likely to belong to a maximum clique, and use this to design a better branching strategy. MCR, introduced by Tomita and Kameda [37], improves this by sorting the vertices according to their degree. This sorting procedure is further improved in MaxCliqueDyn [21], using the fact vertices with low enough color labels cannot be added to the current clique and therefore do not need to be reordered. They also improved performances on dense graphs by running the sorting procedure again at nodes that are near the root of the search tree.

A more efficient implementation of the coloring procedure used in [38] was proposed by San Segundo and Tapia [34], who represent sets as bitsets, and rely on bit-parallel instructions to compute some common operations used while branching and computing the coloring heuristic, such as set intersection.

More complex coloring algorithms have also been used. Tomita et al. [40] introduce MCS, which improves on MCR by using a renumbering procedure after the coloring algorithm to improve the upper bound. MCT, introduced by Tomita et al. [39], additionally uses a k-opt local search to compute an initial lower bound on the solution, and achieves a better trade-off between the effectiveness of the used heuristics and the time spent computing them by using less expensive heuristics near the leaves of the search tree.

The principle of using a graph coloring to compute an upper bound on the solution can be adapted to the vertex weighted version of the problem, as shown by Kumlander [22].

Our algorithm, which uses an exact solver as one of its components, uses such a generalization of MCQ, using an efficient coloring procedure based on a bitset encoding, as described in San Segundo and Tapia [34]. This allows the algorithm to spend little time computing the upper bound at each node of the search tree. The more sophisticated approaches could be more efficient for some instances, as they were shown to be more effective in particular for dense graphs. We also leverage the idea of using local search in order to compute an initial lower bound for the exact solver, as in [39]. This is an effective way of reducing the overall cost of finding an optimal solution.

2.1.2 Approaches Based on MaxSAT

Tighter upper bounds on the maximum size of a clique have been obtained using other approaches than graph coloring. Li and Quan [24] introduced MaxCLQ, which computes a coloring, and then encodes the problem in an instance of partial MaxSAT (Maximum satisifiability) [2]. For each vertex v_i a Boolean variable x_i is defined, determining whether or not a vertex is part of the solution. These variables are used within two types of clauses:

- Hard clauses, which must be satisfied in any solution. There is one hard clause for any pair of non-adjacent vertices, guaranteeing that the solution is a clique. This can be written as the logical disjunction of the negation of the variables corresponding to non-adjacent vertices:

$$\neg x_i \vee \neg x_j \, \forall (v_i, v_j) \notin E$$

- Soft clauses, which may or may not be satisfied in a solution. A graph coloring is computed at each node as in [38], and a soft clause is added for each color class. A clause is satisfied iff at least one vertex belonging to its corresponding color class was selected. This can

be expressed as a logical disjunction of the variables corresponding to the vertices which belong to that color class C:

$$\bigvee_{v_i \in C} x_i$$

The goal of partial MaxSAT is to find an assignment to all variables so that all hard clauses and as many soft clauses as possible are satisfied. In MaxCLQ, inference rules are used on the MaxSAT instance to find inconsistent sets of soft clauses, i.e. soft clauses which cannot all be simultaneously satisfied. This allows the initial upper bound to be lowered, since finding such a sets of soft clauses means that, in any solution, at least one of them will not be satisfied.

This approach was adapted to the maximum weight clique problem in Fang et al. [13], using literal-weighted partial MaxSAT. This is done by giving a weight to each literal (i.e. a variable or its negation), and accounting for this weight when transformation rules are applied.

WLMC [18] extends this approach by not restarting the upper bound computation completely from scratch at each node of the search tree. Instead, it computes a set of vertices which could potentially be added to the current clique, as well as a MaxSAT encoding of the problem into a set of clauses Π. It then iteratively considers each vertex v in this set, and applies transformation rules to Π, with an added clause stating that v should belong to the solution. Based on this, the algorithm may detect that it is not possible to obtain an optimal solution by adding v to the current clique. When a vertex has been processed, in addition to the upper bound, a new set of clauses Π' is obtained. WLMC reduces the total cost of computing upper bounds by using this new set of clauses as a starting point when reasoning about other vertices in the set.

In our algorithm, we opted for the simpler algorithms based on graph coloring. MaxSAT-based upper bounds have however been shown to be tighter than these simple approaches. They could be used in our algorithm in order to allow the optimal solution to be found for larger subgraphs.

2.1.3 Approaches Based on Integer Programming

The maximum clique problem can also been formulated as an integer program, a few formulations have been described in [25] and [4], where linear relaxations of these formulations are used as part of their solvers. The use of Lagrangian relaxations has also been proposed in Campelo and Correa [8].

We consider the use of such methods as part of algorithms, as they are usually tighter bounds than those computed with graph coloring. The drawback is that such upper bounds on the solution are more costly to compute, and thus lead to slower algorithms for easier instances.

2.1.4 Parallel Implementations

Parallel and distributed algorithms have been used to process larger graphs. McCreesh and Prosser [26] use a work stealing parallel algorithm to speed up the execution of a branch and bound algorithm: worker threads take subproblems from a shared queue when they have no more work to do, and donate some unexplored subproblems to the queue when other threads are idle or when the queue is empty. McCreesh and Prosser [27] also present a study of some properties of this search tree which are relevant for a parallel implementation of branch and bound. They show that work-splitting strategies which attempt to diversify the search are more effective even if they do so at the expense of balance.

Alternatively, it is possible to subdivide the graph into smaller subgraphs, and have each thread run a traditional branch and bound algorithm on a different subgraph. This is the approach used by Xiang, Guo, and Aboulnaga [45]. In order to find the exact solution, the branch and bound procedure must not only be executed on each subgraph of the partition, but also on the subgraphs containing the edges that connect different elements of the partition, making it important for a good partitioning strategy to be used.

Our algorithm does not use a parallel algorithm to find the optimal solution of a given graph. Instead, our approach to parallelization is to have multiple threads independently run exact solvers on different graphs, without attempting to obtain the optimal solution. While this approach requires little communication between threads, it cannot produce the type of super-linear scaling found in [26].

2.2 Heuristic Approach

When dealing with graphs containing thousands of vertices, finding the maximum weight independent set and proving the optimality of the solution becomes intractable. Heuristic approaches are used so that a good-enough solution can be found in a reasonable amount of time.

Most of these approaches rely on some form of local search. In this framework, a solution is modified iteratively by perturbing it in order to improve it. So-called metaheuristics are used to deal with local optima. Polynomial time approximation schemes have been proposed for some special cases, such as intersection graphs of convex objects [1], but their complexity makes them impractical for the large graphs which occur in indirect meshing algorithms, as those can containing millions of nodes. Other methods, such as evolutionary algorithms or ant colony optimization, have been used, though they are not currently competitive with traditional local search strategies [42], hence the rest of this section will focus on different local search formulations of the problem.

STABULUS [15] is an early local search algorithm which looks for independent sets by fixing the cardinality of the solution to some value k, and modifies this set with the Swap operator, which adds a vertex to the solution while removing another one from it. The goal is to minimize the amount of edges between vertices in the solution, until there are no such edges — meaning that an independent set has been reached. k is decreased until the algorithm manages to find an independent set of the requested cardinality. It uses the tabu search metaheuristic, which maintains a list of tabu moves that can no longer be used for a certain amount of iterations, so as to mitigate the problem of cycling back to previously explored areas of the search space.

Wu and Hao [43] also use a tabu search for the maximum clique problem, but start by looking for a clique of a small size k, and then gradually increase k until the algorithm is no longer able to find a clique. They also use a restart strategy which allows the algorithm to move to another part of the search space when the solution is no longer improving.

Another family of local search approaches ensure that the current solution remain a valid independent set (or clique) at any time, and directly try to optimize the size of the solution.

Katayama, Hamamoto, and Narihisa [20] proposed a k-opt local search, which considers a sequence of k actions containing Add moves, which add a vertex to the solution, and Drop moves, which remove a vertex from it, where the length of the sequence is dynamically selected at each iteration. A restart strategy is also used when the search seems to be stalling.

Because some algorithms contain several parameters that need to be tuned for them to be effective (e.g. the prohibition duration of a move in a tabu search), it can be difficult to evaluate them. To deal with this problem, Battiti and Protasi [5] introduced a reactive local search (RLS) for the maximum clique problem. The algorithm automatically adjusts the value of such parameters during the search.

A parallel cooperating local search (CLS) was proposed by Pullan, Mascia, and Brunato [32]. In this approach multiple heuristics execute in a parallel. Different heuristic randomly selects vertices with a different bias, allowing them to explore different parts of the search space.

Wu, Hao, and Glover [44] proposed an algorithm to solve the maximum weight clique problem using a tabu search algorithm with restarts. It considers a neighborhood defined by the Add, Swap, and Drop operators, but tries to find the best action overall, instead of only considering

swaps when the solution cannot be improved using the ADD operator, which is the approach used in RLS and CLS.

Wang, Cai, and Yin [41] proposed a local search algorithm which uses the configuration checking metaheuristic, whereby adding a vertex to the solution can only be done if the state of at least one of its neighbors has changed since that vertex was last removed from the solution. In order to decide which swap move should be selected without iterating over a potentially large set, they describe the BMS (Best from Multiple Selections) heuristic, which evaluates a certain amount of randomly selected swaps and returns the best element of this smaller subset.

Zhou, Hao, and Goëffon [47] introduced an operator called PUSH, which adds a vertex v to the solution, and removes from it all vertices which do not share an edge with v. They describe how this operator can be used within a tabu search.

Our algorithm uses such a local search solver, built on top of the method presented in [47]. Instead of using BMS, it uses a priority queue to select vertices, allowing iterations to be performed more quickly.

The solution obtained by this local search solver is used as an initial solution of a more complex algorithm based on a Large Neighborhood Search [31]. This approach converges to a better local optimum thanks to the larger neighborhood it considers, which is similar to the reason the k-opt local search used in [20] is effective. Whereas k-opt local search considers successive applications of small perturbations, we improve the solution by iteratively replacing some part of it with the optimal solution found for a small subgraph.

Chapter 3

Hybrid Approach to Combining Tetrahedra into Hexahedra

Given a tetrahedral mesh of a model, we attempt to generate a hex-dominant mesh of the same geometry by combining elements of the tetrahedral mesh into new elements. The newly generated elements should have a good geometric quality, and be as numerous as possible. This problem can be formulated as an optimization problem known as the maximum weight independent set problem [6]. Given a graph $G = (V, E)$ and for each vertex $v_i \in V$, a weight $w_i \geq 0$ find [42]

$$\max_{S \subseteq V} \sum_{v_i \in S} w_i$$
$$\text{such that } \forall u, v \in S.\ (u, v) \notin E$$

Section 3.1 describes how to generate an instance of the maximum weight independent set, i.e. a weighted graph, from a tetrahedral mesh.

The rest of the chapter describes an algorithm to find a good enough independent set of this graph, as it is usually too large for the optimal solution to be computed. The two major components of this algorithm are a method to find the maximum weight independent set of small graphs (section 3.2), and a heuristic approach used to find a good solution quickly (section 3.3). These are combined in the algorithm described in section 3.4, which is a hybrid method between heuristic approaches and exact resolution methods. This algorithm reaches better results than traditional local search approaches for our instances, while running in a reasonable amount of time even on graphs containing millions of vertices. The efficacy of this approach is analyzed in chapter 4.

3.1 Computation of the Incompatibility Graph

The first step of the algorithm is to compute the set of all potential hexahedra from the input tetrahedral mesh. The graphs used for the rest of the method are produced using the algorithm presented in [30]. This algorithm considers each vertex of the tetrahedral mesh a starting point, and successively adds new vertices such that each newly added vertex is connected by the edges of the tetrahedral mesh to some of the previously selected ones in a pattern which will allow a hexahedron to be generated (Figure 3.1). The algorithm terminates early when it detects that it will not be possible to generate a valid hexahedron. For example, this can happen if it is not possible to find a triangulation of one of the quadrangular faces of the hexahedron such that both triangles are present in the tetrahedral mesh.

To avoid finding the same hexahedron multiple times, constraints on the ordering of the indices of the vertices are added.

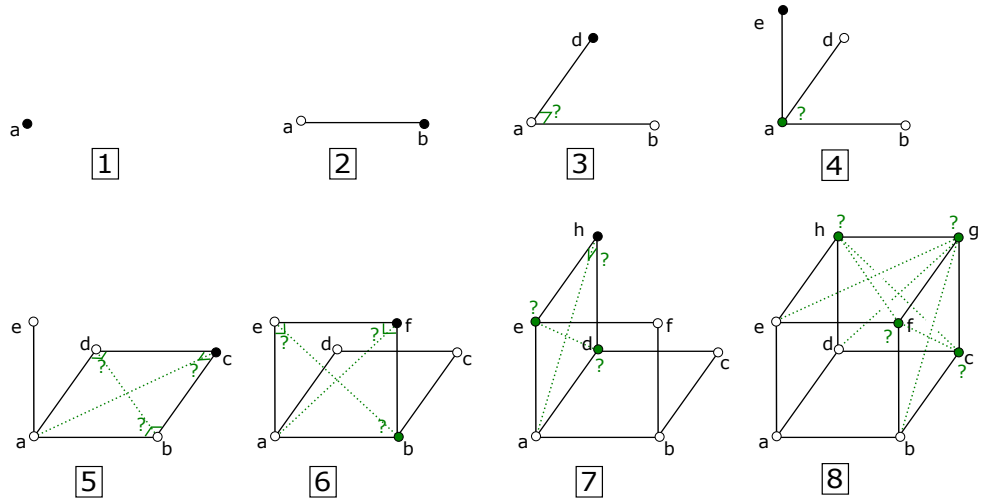


Figure 3.1: Finding a hexahedron in a mesh. The algorithm starts from a vertex and adds vertices connected to previously selected vertices by edges of the tetrahedral mesh. This is repeated until a hexahedron is obtained. Image from [30].

This step is further sped up by the fact that the algorithm computes an upper bound on the quality of the hexahedron being generated, and ends the generation early if this bound becomes lower than some fixed threshold.

This set of elements is the vertex set of the incompatibility graph whose maximum weight independent set needs to be computed. The weight of each vertex corresponds to the geometric quality of the corresponding hexahedron. Maximizing the sum of the qualities of selected elements thus attempts to maximize the quality of the generated mesh.

For the computation of the maximum weight independent set, the edges of the graph also need to be computed. Since we are looking for an independent set, there should be an edge between two incompatible combinations. This is determined by noting that two hexahedra can only be incompatible if they share at least one vertex. Therefore, the algorithm considers all pairs of hexahedra that share at least one vertex, and checks if they are compatible to determine whether or not an edge should be added. Two hexahedra are incompatible if one of the following conditions is satisfied:

1. they share more than 4 vertices;
2. they share exactly 4 vertices which do not correspond to a face shared by both hexahedra;
3. they share exactly 3 vertices;
4. they share exactly 2 vertices which do not correspond to an edge shared by both hexahedra;
5. they both share a tetrahedron, with the exception of flat tetrahedra completely contained within a face of the hexahedron (Figure 3.2).

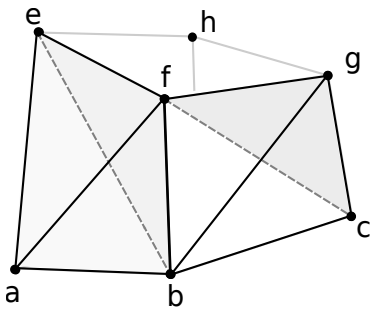


Figure 3.2: Flat tetrahedra on a hexahedron Boundary. These can be shared by two hexahedra in the same mesh. Image from [30].

3.2 Exact Resolution for Small Graphs

Consider the problem of finding the maximum weight independent set of a graph. When the graph G contains up to a few hundreds of vertices, it is generally possible to find the optimal solution. Algorithm 1 accomplishes this by enumerating all independent sets and comparing their total weight against the best known solution [42].

A complete search algorithm cannot be expected to terminate in a reasonable amount of time for the large incompatibility graphs obtained from meshes, even if those only contain thousands of elements. However, it remains useful to consider how to perform a complete search efficiently because this can be used as a building block in algorithms suited for larger graphs (see section 3.4).

The drawback of the basic complete search algorithm is that it very quickly becomes intractable. In this section, we consider how a complete search can be improved so as to allow the optimal solution to be found for larger graphs by using the branch and bound framework (subsection 3.2.2). The major component of this approach is an upper bounding procedure, for which some possible approaches are described in subsections 3.2.3 and 3.2.4.

3.2.1 Complete Search

Algorithm 1 COMPLETE-SEARCH: An exact algorithm for the maximum weight independent set

Input: A graph $G = (V, E)$, a set of candidate vertices P, a partial solution S of weight W, and the best solution so far S^* of weight W^*.
Output: The improved best solution S^* and its weight W^*

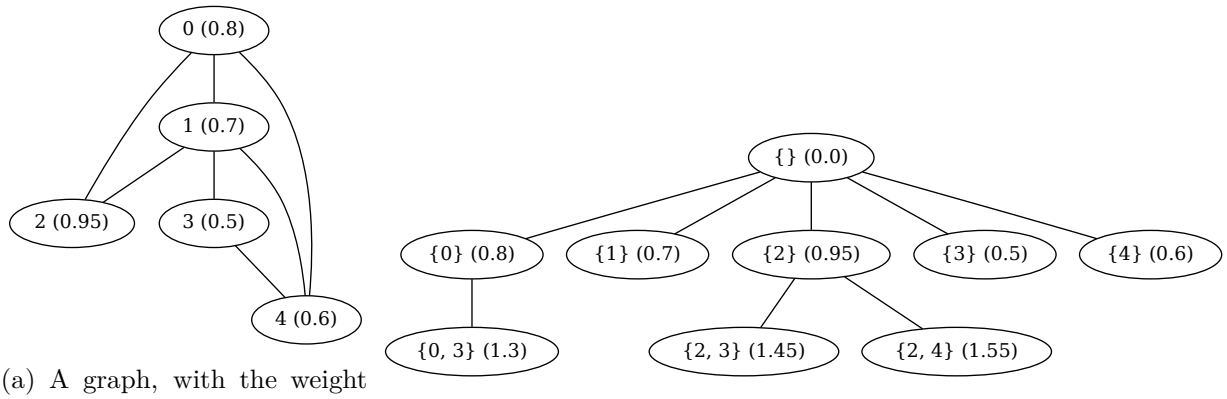
```
1: if W > W* then
2:     S* ← S
3:     W* ← W
4: end if
5: for each vertex v_i in P do
6:     P ← P \ {v_i}
7:     (S*, W*) ← COMPLETE-SEARCH(G, P \ N_G(v_i), S ∪ {v_i}, W + w_i, S*, W*)
8: end for
9: return (S*, W*)
```

Consider Algorithm 1. The algorithm looks for the maximum weight independent set of G. To do so, it solves a new subproblem for each vertex v_i in P: "What is the maximum weight independent set in G that contains v_i and none of the previously considered vertices?".

This is illustrated in the *search tree* represented on Figure 3.3b. Each node of this tree represents one of the subproblems explored by the algorithm, the root being the original problem. Each node has edges going down from itself to all of its subproblems: the leftmost child of the root assumes that vertex 0 is part of the solution, the one to its right assumes that vertex 1 is part of the solution but vertex 0 is not, the next one assumes that vertex 2 is part of the solution but vertices 0 and 1 are not, and so on.

Let us define the *solution space* of a problem as the set of all elements that respect the problem's constraints. In this context, the solution space of the original problem is the set of all independent sets of G.

Each subproblem corresponds to a subset of the solution space. When they have all been explored, the entire solution space has also been explored. The algorithm finds the best solution in the entire space by finding the best solution of each subproblem, i.e. the one whose total weight is the largest, and keeping the best solution overall.



(a) A graph, with the weight of each vertex specified between parentheses.

(b) Corresponding search tree. Nodes are labeled with their current solution S followed by its total weight W between parentheses.

Figure 3.3: A graph and its search tree

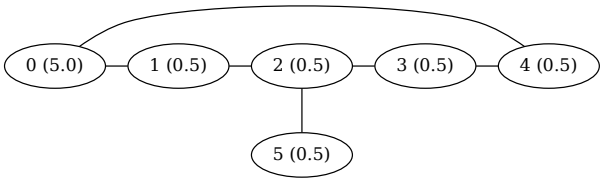


Figure 3.4: A graph, with the weight of each vertex specified between parentheses.

3.2.2 Branch and Bound

The algorithm presented above visits all independent sets once, and eventually finds the one whose weight is the largest. It is however possible to significantly improve the performances of the algorithm by not visiting certain parts of the search tree, while still guaranteeing that the solution returned by the algorithm is optimal.

Consider the graph shown in Figure 3.4. The algorithm starts by looking for the maximum weight independent set that contains vertex 0. After exploring the corresponding subtree, it will find:

$$S = \{0, 3, 5\}$$

whose total weight is 6.0. The next step would be to consider all independent sets which contain vertex 1 but not vertex 0. After adding vertex 1 to the solution, its weight is increased to 0.5. The remaining candidate vertices are then:

$$P = \{3, 4, 6\}$$

Since the set of vertices to add to the solution is a subset of P, let us consider the case where all the elements of P are added to the solution. Even in this very optimistic case, the total weight of the solution would be 2.0, less than the best known solution of weight 6.0. This makes it clear that there is no point in visiting the children of this node.

This approach is known as branch and bound. In general, a branch and bound algorithm will find the optimal solution by subdividing the problem into subproblems, and tracks the best known solution so far (e.g. variable W in Algorithm 1). Since the optimal solution is at least as good as any solution that has been found by the algorithm, this is a lower bound on the optimal solution. The difference with Algorithm 1 is that, when visiting a node of the search tree, an upper bound on the best solution within the corresponding subset of the search space is computed.

If this upper bound is less than the lower bound maintained by the algorithm, then it is not possible for the optimal solution of the original problem to be found by solving that subproblem. There is therefore no point in exploring the children of this node.

A good upper bounding procedure needs to be tight, i.e. close to the optimal solution, and it also should be possible to evaluate it efficiently, as this procedure needs to be evaluated for every node of the search tree. There can arise a compromise between tighter upper bounds, which allow larger portions of the search tree to be pruned, and more efficient upper bounds, which can be evaluated at a lower cost.

For a more detailed explanation of the branch and bound approach the reader is referred to [9].

In the context of Algorithm 1, the optimal solution of a subproblem is given by:

$$\max_{X \subseteq P} \sum_{v_i \in S} w_i + \sum_{v_i \in X} w_i$$
$$\text{such that } S \cup X \text{ is an independent set}$$

where P is the set of candidate vertices and S is the current solution.

Since the algorithm guarantees that S is an independent set (by only adding vertices which have no neighbor in the current solution), and there is no edge between any element of P and elements of S (as neighbors of vertices which have been added to the solution are removed from the candidate set), the constraint that $S \cup X$ must be an independent set is equivalent to there being no edge between the elements of X. In other words, X needs to be an independent set of the induced subgraph G_P.

This means that an upper bound on the optimal solution of a subproblem can be described in terms of an upper bound on the maximum weight independent set of the induced subgraph of the set of candidate vertices. Algorithm 2 uses an Upper-Bound procedure which computes a value greater than or equal to the maximum weight independent set of a graph.

In the example at the beginning of this subsection, the upper bound which was used was simply the sum of the weights of all vertices:

$$\sum_{v_i \in V} w_i$$

While this upper bound can be computed very efficiently, it is not a tight bound. The next subsections describe procedures which process the graph in order to find tighter bounds.

Algorithm 2 A branch and bound algorithm to compute the maximum weight independent set

Input: A graph $G = (V, E)$, a set of candidate vertices P, a partial solution S of weight W, and the best solution so far S^* of weight W^*.
Output: The improved best solution S^* and its weight W^*

```
1: if W + Upper-Bound(G_P) ≤ W* then
2:     return (S*, W*)
3: end if
4: if W > W* then
5:     S* ← S
6:     W* ← W
7: end if
8: for each vertex v_i in P do
9:     P ← P \ {v_i}
10:    (S*, W*) ← Branch-And-Bound(G, P \ N_G(v_i), S ∪ {v_i}, W + w_i, S*, W*)
11: end for
12: return (S*, W*)
```

3.2.3 Upper Bounds Based on Clique Partitions

Consider a clique C and an independent set S. For any two vertices u and v in C, there is an edge $(u, v) \in E$. However, for vertices which belong to the independent set S no such edges can exist. This means that at most one vertex of C can belong to S.

This property can be used to compute an upper bound on the size of any independent set $|S|$: given a partition of the vertices of a graph into cliques $\mathcal{C}$, in the best case scenario, any clique $C \in \mathcal{C}$ will have one vertex in S. This implies that [3]

$$|S| \leq |\mathcal{C}|$$

This is easily adapted into an upper bound for the maximum weight independent set by considering that in the best possible case, S will contain one vertex of each clique $C \in \mathcal{C}$, and that vertex will be the vertex of maximum weight within that clique [22]. This leads to the upper bound

$$\sum_{C \in \mathcal{C}} \max_{v_i \in C} w_i$$

Computing a minimal clique partition leads to tighter bounds, but is equivalent to computing a minimum coloring of the complementary graph $\overline{G}$, and is thus NP-Complete [19]; greedy algorithms such as Algorithm 3 can be used to compute a partition efficiently [33].

Algorithm 3 A greedy algorithm to partition the vertices of a graph into cliques

Input: A graph $G = (V, E)$
Output: A vector **l** containing the index of the clique that each vertex was partitioned into.

```
 1: Q ← V {The set of vertices that have not been assigned to a clique}
 2: k ← 0
 3: while Q ≠ ∅ do
 4:    U ← Q {The set of vertices that can be added to the current clique}
 5:    while U ≠ ∅ do
 6:       v_i ←  the vertex with minimal index in U
 7:       l_i ← k
 8:       U ← U \ {v}
 9:       U ← U ∩ N_G(v)
10:       Q ← Q \ {v}
11:    end while
12:    k ← k + 1
13: end while
14: return l
```

There are two notable ways this can be used to compute upper bounds [42]: either by running the partitioning algorithm once at the beginning of the search, or by recomputing a new partition into cliques for G_P at every node of the search tree.

The first approach allows the upper bound to be maintained very efficiently: every time the last vertex of a clique C is removed from the set of candidates P, the upper bound is lowered by $\max_{v_i \in C} w_i$.

The second approach is more computationally expensive, but will generally lead to tighter bounds, making it worthwhile overall. Since the partitioning algorithm now needs to be executed at every node the search tree, an efficient implementation of the clique partition algorithm is required.

If the graph is small enough that its complete adjacency matrix can be stored, representing sets as arrays of bits leads to an implementation that computes the intersection in Algorithm 3, statement 9 using a bitwise AND operation [33]. This can be done using the regular registers of the CPU used for the computation, or, on some platforms, using larger vector registers [11].

In addition to being useful as a means to compute an upper bound, the clique partition can be used to improve the branching strategy: Tomita and Seki [38] note that vertices which have been assigned to a clique with a higher index by the greedy algorithm are more likely to be part of a maximum independent set. Therefore, in Algorithm 2, in statement 8, the vertices can be iterated on in descending order of clique index. This branching order helps the algorithm find a good lower bound early, and consequently allows it to visit fewer nodes within the search tree.

3.2.4 Upper Bounds based on Linear Programming

Another way to compute an upper bound on the weight of the optimal solution is to consider formulations of the maximum weight independent set as an integer linear program [29]. For instance, this edge based formulation:

$$\begin{aligned} &\max_{\mathbf{x}\in\{0,1\}^{|V|}} \sum_{v_i\in V} w_i x_i \\ &\text{subject to } x_i + x_j \leq 1 \ \forall (v_i, v_j) \in E \end{aligned} \tag{LP-Edge}$$

In this formulation, x_i is set to 1 iff v_i belongs to the independent set. The constraints guarantee that there is no edge between selected vertices.

An equivalent formulation can be obtained by writing constraints that guarantee that that at most one vertex of any clique is selected. Given $\mathcal{C}$ the set of all cliques in G, we have the following problem [25]:

$$\begin{aligned} &\max_{\mathbf{x}\in\{0,1\}^{|V|}} \sum_{v_i\in V} w_i x_i \\ &\text{subject to } \sum_{v_i\in C} x_i \leq 1 \ \forall C \in \mathcal{C} \end{aligned} \tag{LP-Clique}$$

The drawback of this second formulation is that it may contain an exponential amount of constraints. However, it is not necessary to include all cliques of G: it is sufficient for $\mathcal{C}$ to be an edge clique cover, i.e. a set of cliques such that for any edge $(u, v) \in E$, there is a clique $C \in \mathcal{C}$ which contains both u and v [4]. Algorithm 4 shows a greedy algorithm to compute such a set $\mathcal{C}$.

Algorithm 4 Greedy computation of an edge clique cover

Input: A graph $G = (V, E)$
Output: An edge clique cover $\mathcal{C}$
1: $\mathcal{C} \leftarrow \emptyset$
2: **while** there is an unexplored edge $(u, v) \in E$ **do**
3: $\quad Q \leftarrow (N_G(u) \cap N_G(v))$
4: $\quad C \leftarrow \{u, v\}$
5: $\quad$ **while** $Q \neq \emptyset$ **do**
6: $\quad\quad x \leftarrow$ some vertex in Q
7: $\quad\quad C \leftarrow C \cup \{x\}$
8: $\quad\quad Q \leftarrow (Q \setminus \{x\}) \cap N_G(x)$
9: $\quad$ **end while**
10: $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{C\}$
11: $\quad$ **for** each pair of two vertices x and y belonging to C **do**
12: $\quad\quad$ Mark the edge (x, y) as explored
13: $\quad$ **end for**
14: **end while**
15: **return** $\mathcal{C}$

The equivalence between the edge formulation and this formulation can be found by considering the fact that each constraint of the second problem can be split into several constraints of the form:

$$x_i + x_j \leq 1 \; \forall v_i \in C. \; \forall v_j \in C \setminus \{v_i\}$$

Any such constraint is also a constraint of the first problem because (v_i, v_j) is an edge (since v_i and v_j belong to one same clique). In addition to this, all constraints of the first problem are present in such a decomposition because $\mathcal{C}$ is an edge clique cover.

These integer linear programs are strictly equivalent to the original problem, and are therefore no easier to solve. They can however be relaxed into easier problems whose solution can be computed in polynomial time, most notably linear and Lagrangian relaxations.

The linear relaxation considers a linear program where the constraints that $x_i \in \{0, 1\}$ is replaced by $x_i \in [0, 1]$. The optimal solution to this problem, which can be computed using a generic algorithm to solve linear programs, is an upper bound to the solution of the integer linear program.

The Lagrangian relaxation of the second formulation of the problem is obtained by observing that for any feasible solution and any clique $C \in \mathcal{C}$, given a non-negative value λ_C:

$$\lambda_C \left(-1 + \sum_{v_i \in C} x_i \right) \leq 0$$

Hence, simply subtracting such terms from the objective function gives an upper bound. Removing the constraints of the original problem only adds new elements to the search space, which cannot possible lower the optimal value. Furthermore, while any non-negative values of the Lagrange multipliers λ_C give an upper bound, the tightest upper bound is found by minimizing the upper bound over all possible values of the multipliers, giving the following expression:

$$\begin{aligned}
\mathcal{U}^* &= \min_{\lambda_C \geq 0 \; \forall C \in \mathcal{C}} \max_{\mathbf{x} \in \{0,1\}^{|V|}} \left(\sum_{v_i \in V} w_i x_i \right) - \left(\sum_{C \in \mathcal{C}} \lambda_C \left(-1 + \sum_{v_i \in C} x_i \right) \right) \\
&= \min_{\lambda_C \geq 0 \; \forall C \in \mathcal{C}} \max_{\mathbf{x} \in \{0,1\}^{|V|}} \left(\sum_{v_i \in V} (w_i - \sum_{C \in \mathcal{C} \text{ s.t. } v_i \in C} \lambda_C) x_i \right) + \sum_{C \in \mathcal{C}} \lambda_C
\end{aligned}$$

The maximization problem can be solved by setting x_i to 1 for any vertex v_i such that

$$w_i - \sum_{C \in \mathcal{C} \text{ s.t. } v_i \in C} \lambda_C \tag{3.1}$$

is non-negative. The minimization problem can be solved using a subgradient method, as shown in Algorithm 5 [14].

Algorithm 5 Subgradient method for LP-Clique

Input: A edge clique cover $\mathcal{C}$ and initial values for each the Lagrangian multipliers λ_C
Output: An upper bound $\mathcal{U}$

$k \leftarrow 0$ {k is the iteration number}
while stopping criterion not met **do**
  $k \leftarrow k + 1$
  $\mathbf{x} \leftarrow \max_{\mathbf{x} \in \{0,1\}^{|V|}} \left(\sum_{v_i \in V} (w_i - \sum_{C \in \mathcal{C} \text{ s.t. } v_i \in C} \lambda_C) x_i \right) + \sum_{C \in \mathcal{C}} \lambda_C$
  for each clique $C \in \mathcal{C}$ **do**
    {μ_k is the step size, e.g. a constant or $\frac{1}{k}$}
    $\lambda_C \leftarrow \lambda_C - \mu_k \sum_{v_i \in C} x_i$
  end for
end while
return $\left(\sum_{v_i \in V} (w_i - \sum_{C \in \mathcal{C} \text{ s.t. } v_i \in C} \lambda_C) x_i \right) + \sum_{C \in \mathcal{C}} \lambda_C$

As there are two possible relaxations of the LP-Clique problem, it is interesting to analyze how they relate to each other. Notably, the linear and Lagrangian relaxation presented above have the same optimal solution. This is the consequence of LP-Clique having a property known as *integrality* [14]: removing from the Lagrangian relaxation the constraints that the variables x_i be integers does not decrease its value, since the objective function is still maximized by setting x_i to 1 when (3.1) is positive, and to 0 when (3.1) is negative.

The benefit of the Lagrangian relaxation, therefore, does not come from how tight the upper bound obtained in this manner is, but from the fact that it is not necessary to compute the best possible upper bound: any upper bound could in principle be returned. Therefore, tuning the amount of iterations done in the subgradient method allows the trade-off between the tightness of the upper bound and the time spent computing it to be adjusted. In particular, a successful approach can be to perform relatively few iterations (e.g. 100) so as to compute a good-enough bound more quickly [12].

3.3 Construction of an Initial Solution

In section 3.2, several algorithms used to find the maximum weight independent set have been described. These algorithms come with the guarantee of finding the optimal solution, but have exponential worst case complexity, making them ill-suited when dealing with graphs containing thousands of vertices.

In practice, a mesh can contain several millions of tetrahedra, and the set of potential hexahedra obtained by combining them is even larger. This means that other approaches need to be considered in order to find a solution in a timely fashion. For the application to mesh generation, prior work has focused on using greedy algorithms [6].

In this section, local search algorithms for the maximum weight independent set are considered as an alternative way to quickly obtain a reasonable, though not necessarily optimal, solution.

The next subsections present the general structure of a local search algorithm (subsection 3.3.1), and how to implement its different components in order to solve the maximum weight independent set problem: a formulation of the solution space and of actions to modify the solution (subsection 3.3.2), heuristics which can be used to escape local optima, which are an important concern in local search algorithms (subsection 3.3.3), and data structures allowing an efficient implementation to be achieved (subsection 3.3.4).

3.3.1 Local Search Algorithms

Consider the problem of finding the element S of a solution space $\mathcal{S}$, such that S maximizes an objective function f. A local search algorithm starts from some initial solution $S \in \mathcal{S}$, and iteratively perturbs it in order to improve it. This is accomplished by, at each iteration, replacing the current solution S found within a set $\mathcal{N}(S)$ known as the *neighborhood* of the current solution. Algorithm 6 illustrates the general principle behind a local search algorithm.

Notice that in statement 3, it is not always necessary to explicitly compute the new value of the objective function for each state of the neighborhood. Efficient local search algorithms benefit from the fact that the neighborhood can be explored with a lower time complexity than in the naive approach. This concept is illustrated in subsection 3.3.4.

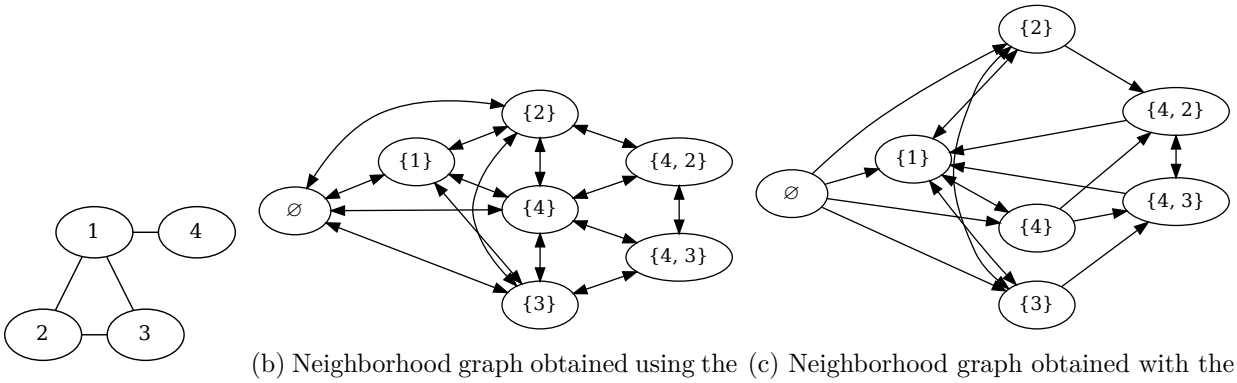
Algorithm 6 Simple Local Search Algorithm

Input: An objective function f and an initial solution S
Output: The best solution found S^*
1: $S^* \leftarrow S$
2: **while** stopping criterion not met **do**
3: $\quad S \leftarrow \arg\max_{S' \in \mathcal{N}(S)} f(S')$
4: $\quad$ **if** $f(S) > f(S^*)$ **then**
5: $\quad\quad S^* \leftarrow S$
6: $\quad$ **end if**
7: **end while**
8: **return** S^*

The neighborhood function $\mathcal{N}$ is a function assigning a neighborhood to each state. Each element of the neighborhood is generally obtained by applying a small perturbation to the current solution. In subsection 3.3.2, a set of operators are defined and used for this purpose.

The neighborhood function can also be described using the neighborhood graph. This graph has one vertex for each feasible solution $S \in \mathcal{S}$, and an edge from S to S' iff $S' \in \mathcal{N}(S)$. Figure 3.5 illustrates such a graph for the maximum weight independent set, using the neighborhoods defined in subsection 3.3.2.

A desirable property of the neighborhood graph is for it to be *weakly optimally connected*, meaning that, for any possible state, there exists a path leading to some optimal solution. Not



(a) Input graph

(b) Neighborhood graph obtained using the ADD, SWAP, and DROP operators

(c) Neighborhood graph obtained with the PUSH operator

Figure 3.5: A graph and possible neighborhood graphs. Each node is a solution, and each edge is an application of an operator.

having this property implies the optimal solution can never be reached once certain states have been reached.

An important issue with the naive formulation shown in Algorithm 6 comes from *local optima*. A (strict) local maximum is a solution S whose neighborhood does not contain any solution that achieves a better value of the objective function:

$$\forall S' \in \mathcal{N}(S).\ f(S') < f(S)$$

When encountering such a state, the algorithm will not be able to improve the solution. This motivates the use of metaheuristics such as tabu search, configuration checking or restarts. The use of these for the maximum weight independent set problem is described in subsection 3.3.3.

A more detailed description of local search algorithms can be found in [23].

3.3.2 Local Search Formulation of the Maximum Weight Independent Set

The maximum weight independent set problem can be described in the framework presented in the previous subsection. The solution space $\mathcal{S}$ is defined as the set of all independent sets. The objective function is the total quality of the solution, defined as

$$f(S) = W = \sum_{v \in S} w_v$$

The neighborhood of a solution S can be defined as the set of independent sets obtained by a single application of one of the following operators to S (Figure 3.6):

- ADD(v) adds a vertex v to the current independent set. Performing such an action increases the objective value by
$$\delta_{\text{ADD}(v)} = w_v$$
- SWAP(u, v) removes a vertex u from the independent set and adds a vertex v. Performing such an action increases the objective value by
$$\delta_{\text{SWAP}(u,v)} = w_v - w_u$$
- DROP(v) removes a vertex v from the current independent set. Performing such an action increases the objective value by
$$\delta_{\text{DROP}(v)} = -w_v$$

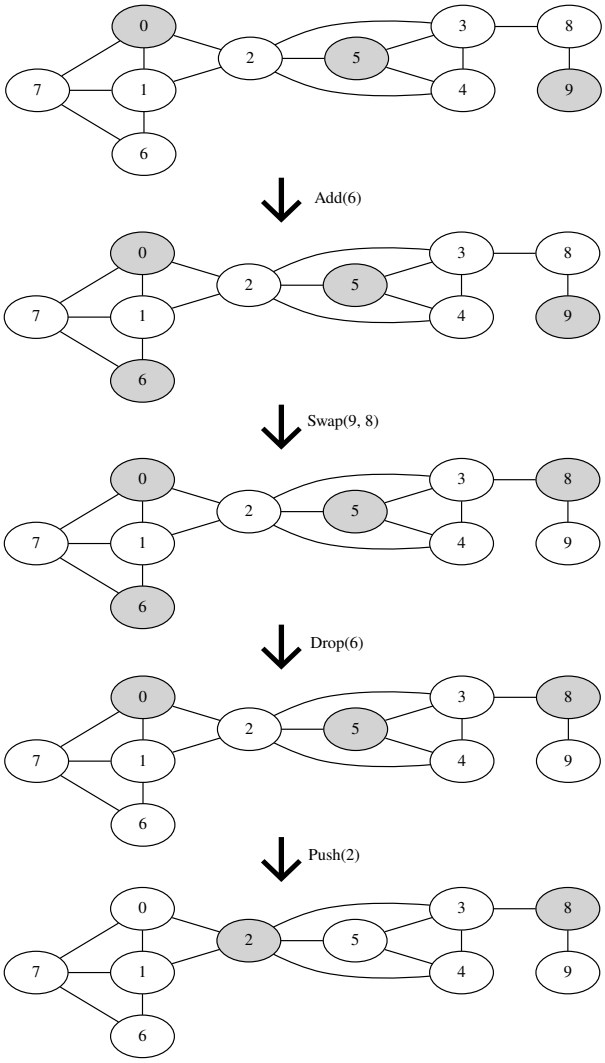


Figure 3.6: Effects of operators used in a local search algorithm

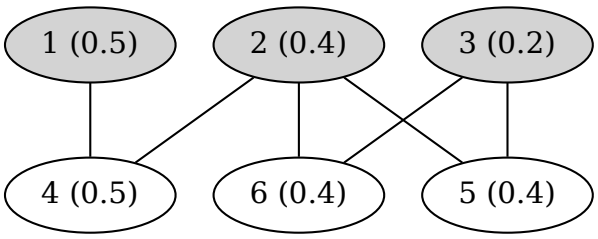


Figure 3.7: State in which ADD and DROP cannot be used. The weight of each vertex is shown between parentheses.

The requirement that S remain an independent set after each iteration imposes the following conditions on the use of the ADD and SWAP operators:

- ADD(v) can only be performed if v has no neighbor in the current solution:

$$N_G(v) \cap S = \emptyset$$

- SWAP(u, v) can only be performed if v has either no neighbor in the current solution, or is adjacent to u but no other vertices belonging to the current solution:

$$N_G(v) \cap S \subseteq \{u\}$$

The third operator, DROP, cannot improve the solution but is required for the neighborhood graph to be weakly optimally connected. For example, Figure 3.7 shows a situation from which it is not possible to reach any of the two optimal solutions ($\{1, 5, 6\}$ and $\{4, 5, 6\}$) without using the DROP operator. Indeed, since all vertices are either in the solution or adjacent to multiple vertices belonging to it, it is not possible to use SWAP or ADD.

It is possible to use a simpler neighborhood which relies on a single operator known as PUSH [47]. PUSH(v) adds v to the current solution and removes from it any vertex adjacent to v, so as to maintain a valid independent set (see Figure 3.6). This increases the objective function by

$$\delta_{\text{PUSH}(v)} = w_v - \sum_{u \in N_G(v) \cap S} w_u$$

The neighborhood defined by this operator has some benefits over the one defined using ADD and SWAP:

- The neighborhood structure is weakly optimally connected without needing the DROP operator. It is always possible to reach some solution S^* by sequentially applying PUSH(v) for any vertex $v \in S^* \setminus S$.

  Each operation adds a vertex v found in S^* but missing from S. The vertices that may be removed by the operation cannot belong to S^* as this would contradict the fact that S^* is an independent set. This means that this process eventually terminates, with the new solution equal to S^*.

- The set of local maxima using the neighborhood defined by PUSH is a subset of the set of local maxima of the neighborhood defined using the ADD, SWAP and DROP operators.

  Consider an independent set S which is a local maximum of the neighborhood defined using operator PUSH. This means that it is not possible to improve this solution by a single application of the PUSH operator. It is not possible to improve it using DROP because

$$\delta_{\text{DROP}(v)} = -w_v \leq 0$$

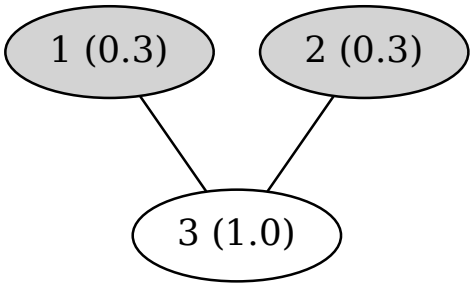


Figure 3.8: Solution which can only be improved using PUSH. The weight of each vertex is shown between parentheses.

It is also not possible to use the ADD operator on such a solution S, since for any vertex for which $\text{ADD}(v)$ is allowed, $\text{PUSH}(v)$ is equivalent and would improve the objective function by

$$\delta_{\text{PUSH}(v)} = \delta_{\text{ADD}(v)} = w_v \geq 0$$

The only remaining possibility would be to improve the solution using the SWAP operator. If this were possible, it would be possible to improve the solution at least as much using PUSH. Consider a pair of vertices (u, v) such that $\text{SWAP}(u, v)$ is legal. Either this means that v is not adjacent to any vertex in the solution, which means that

$$\delta_{\text{PUSH}(v)} = w_v \geq w_v - w_u = \delta_{\text{SWAP}(u,v)}$$

or alternatively, that v is adjacent to u and no other vertex belonging to the solution, in which case

$$\delta_{\text{PUSH}(v)} = w_v - \sum_{x \in N_G(v) \cap S} w_x = w_v - w_u = \delta_{\text{SWAP}(u,v)}$$

In other words, any such solution S is also a local maximum in the neighborhood defined using ADD, SWAP and DROP.

Note also that the two sets are not equal for at least some graphs: Figure 3.8 shows a state which cannot be improved using ADD, SWAP and DROP, but performing PUSH(3) would allow the optimal solution to be reached.

- There can be as many as $|S|(|V| - |S|)$ valid SWAP operations in any given state, whereas there can only be $|V| - |S|$ valid PUSH operations. This makes the neighborhood defined by PUSH less costly to explore.

For these reasons, the following subsections will use a formulation that solely relies on the PUSH operator.

3.3.3 Strategies to Escape Local Optima

Without metaheuristics, a local search algorithm will start cycling between previously visited states once it hits a local maximum. It is therefore important to allow the algorithm to perform actions which degrade the current solution, in order to start exploring different areas of the neighborhood graph.

Tabu search is a strategy which achieves this by restricting the neighborhood of a solution to some set of *legal* neighbors. This restriction is determined using a *tabu list*, maintained by the algorithm to avoid cycling back to recently visited solutions.

For the maximum weight independent set problem, a vertex v can be added to the tabu list T when v is removed because of a PUSH operation. As long as v remains in the tabu list, it is forbidden to use PUSH(v). Eventually, v should be removed from the tabu list so that it can be added to the solution again once the search has moved to another section of the neighborhood graph. An important parameter is the number of iterations for which a vertex remains in the tabu list. For example, one could choose to pick a random duration within the following interval:

$$[7, 7 + |\{u \in V | \deg(u) = \deg(v)\}|)$$

This definition was empirically determined to be effective in [47]. Using it, vertices will tend to remain in the tabu list longer in larger graphs. This allows more iterations to be performed in order to move the search to another part of the search space.

Sometimes, the algorithm may determine that it is beneficial to perform an action even if that action is tabu. Typically, this can be done when it makes the solution better than any of the previously visited solutions. Such a condition is known as an *aspiration criterion*. Algorithm 7 shows a tabu search with an aspiration criterion.

Algorithm 7 Tabu search based on the PUSH operator

Input: A graph $G = (V, E)$
Output: An independent set $S*$ with weight $W*$

$S^* \leftarrow \emptyset$ {Best solution so far}
$W^* \leftarrow 0$ {Total weight of S^*}
$S \leftarrow \emptyset$ {Current solution}
$W \leftarrow 0$ {Total weight of S}
{Initialize tabu list}
for $v \in V$ **do**
  $T_v \leftarrow 0$
end for
$i \leftarrow 0$ {Iteration number}
while stopping condition not met **do**
  $P \leftarrow \{v \in V \setminus S \mid T_v \leq i \vee W + \delta_{\text{PUSH}(v)} > W^*\}$
  $v \leftarrow \arg\max_{v \in P} \delta_{\text{PUSH}(v)}$ {Pick the best non-tabu vertex to push}
  for $u \in (S \cap N_G(v))$ **do**
    $d \leftarrow$ a random integer between 7 and $7 + |\{v \in V | \deg(v) = \deg(u)\}|$
    $T_u \leftarrow i + d$ {Mark removed vertices as tabu}
  end for
  $W \leftarrow W + \delta_{\text{PUSH}(v)}$
  $S \leftarrow (S \cup v) \setminus N_G(v)$ {Perform PUSH(v)}
  if $W > W^*$ **then**
    $S^* \leftarrow S$ {Store the new best solution}
    $W^* \leftarrow W$
  end if
  $i \leftarrow i + 1$
end while

Sometimes, the tabu metaheuristic is not enough to escape from local optima when these are too far away from a better solution. To handle such situations, restarts can be used to move to completely different parts of the search space.

The principle of using restarts is that if the best solution was found more than L iterations ago, it is beneficial to diversify the search by replacing the current solution by a completely different solution. The value of L can be chosen as some small constant multiplied by the amount of vertices in the graph (e.g. $2|V|$), making restarts less frequent for larger graphs.

In particular, two algorithms can be considered to compute the new solution when a restart occurs [47]:

- RandomRestart (Algorithm 8): Start from a random valid independent set. This is done by starting from an empty set, and randomly selecting a vertex v not adjacent to any previously selected vertex before adding it to the solution. The algorithm ends when there are no such vertices anymore.
- ReconstructSolution (Algorithm 9): Use the current solution to construct a new solution by iteratively replacing its vertices. Initially all vertices in $V \setminus S$ are marked as available. Whenever a vertex is removed from the solution, that vertex is marked as unavailable until the new solution is constructed. At each step, the algorithm selects the available vertex which maximizes $\delta_{\text{Push}(v)}$ and performs Push(v). The algorithm ends when all vertices are unavailable.

Algorithm 8 RandomRestart

Input: A graph $G = (V, E)$
Output: An independent set S
1: $S \leftarrow \emptyset$
2: $\overline{S} \leftarrow V$
3: **while** $\overline{S} \neq \emptyset$ **do**
4: $v \leftarrow$ a random vertex from $\overline{S}$
5: $S \leftarrow S \cup \{v\}$
6: $\overline{S} \leftarrow \overline{S} \setminus (v \cup N_G(v))$
7: **end while**

Algorithm 9 ReconstructSolution

Input: A graph $G = (V, E)$ and an independent set S_0
Output: A new independent set S
1: $S \leftarrow S_0$
2: $\overline{S} \leftarrow V \setminus S$
3: **while** $\overline{S} \neq \emptyset$ **do**
4: $v \leftarrow \arg\max_{v \in \overline{S}} \delta_{\text{Push}(v)}$
5: $S \leftarrow (S \cup v) \setminus N_G(v)$
6: $\overline{S} \leftarrow \overline{S} \setminus \{v\}$
7: **end while**

When a restart is decided, the algorithm uses ReconstructSolution with probability ρ, and uses RandomRestart otherwise. This is shown in Algorithm 10. The value of ρ should be selected in order to make a trade off between intensification and diversification: the ReconstructSolution algorithm uses the objective function to create a new solution, and does not tend to diversify the search as much as RandomRestart, which can move to a random independent set.

3.3.4 Data Structures for a Local Search Algorithm

The most expensive operation in the tabu search algorithm presented in the previous subsection is the selection of which vertex v should be pushed into the solution: the naive approach, which sequentially considers every possible operation, needs to check $|V| - |S|$ operations, one for each vertex which is currently not part of the solution.

The BMS (Best from Multiple Selections) can be used in order to alleviate the cost of this step [41]. The idea of this heuristic is that, instead of checking every element of the current neighborhood in order to find the best neighbor of the solution, it is possible to find a good

Algorithm 10 RESTART

Input: A graph $G = (V, E)$ and an independent set S_0
Output: A new independent set S
1: $x \leftarrow$ a random uniformly distributed value in $[0, 1]$
2: **if** $x \leq \rho$ **then**
3: $\quad S \leftarrow \text{RECONSTRUCTSOLUTION}(G, S_0)$
4: **else**
5: $\quad S \leftarrow \text{RANDOMRESTART}(G)$
6: **end if**

enough move by selecting the best of k randomly selected elements of the neighborhood. One can also explore some smaller section of the neighborhood exhaustively and use the BMS heuristic when exploring some other part of the neighborhood [47]. While this approach does not guarantee that the best vertex will be selected, it allows a vertex to be selected at a cheaper cost.

We propose another method allowing the cost of the vertex selections step to be reduced, while still selecting the same vertex. This result is obtained by maintaining a data structure allowing the best vertex to be determined efficiently.

Specifically, consider a priority queue Q which supports the following operations:

- ENQUEUE(Q, x): adds x to the priority queue.
- DEQEUE(Q): removes and returns the element with the highest priority.
- INCREASE-KEY(Q, x): updates the priority queue after the priority of x has increased.
- DECREASE-KEY(Q, x): updates the priority queue after the priority of x has decreased.

Using a binary heap, any of these operations can be implemented with a worst-case time complexity of $\Theta(\log n)$, where n is the amount of elements in the priority queue [10].

Such a priority queue can be used to determine which vertex should be selected by storing in it all vertices that are not currently part of the solution. If the aspiration criterion is ignored, the vertex which should be selected is the one which achieves the highest reward $\delta_{\text{PUSH}(v)}$ among all vertices such that PUSH(v) is not tabu. It follows that a vertex u should be considered to have a higher priority than another vertex v iff one of the below conditions is verified:

- PUSH(v) is tabu but PUSH(u) is not;
- neither PUSH(u) nor PUSH(v) are tabu and $\delta_{\text{PUSH}(u)} > \delta_{\text{PUSH}(v)}$.

Algorithm 11 details how this queue is maintained in a tabu search algorithm with restarts. At each iteration, the algorithm selects a vertex v, adds it to the solution, and removes its neighbors from it. Since the queue should contain all elements not in the solution, this means removing v from the queue (statement 27), and enqueuing all of its neighbors (statement 34).

It is not sufficient to just maintain the contents of the queue. Whenever the relative order of two elements changes, INCREASE-KEY and DECREASE-KEY need to be used.

1. When pushing a vertex into the solution is no longer tabu, its priority increases (statement 16).
2. When a vertex v is added to the solution, the priority of all its neighbors should decrease, since pushing them into the solution would now require v to be removed from it (statement 29).
3. When a vertex v is removed from the solution, the priority of all its neighbors should increase, since pushing them into the solution no longer requires v to be removed from it (statement 36).

In order to account for the aspiration criterion, the algorithm must be modified slightly. Two priority queue Q_1 and Q_2 should be maintained. Q_1 contains non-tabu vertices not belonging to the solution, whereas Q_2 contains vertices which are not tabu. Both queues should be ordered according to the reward. In this case, the vertex which should be selected is always at the front of either Q_1 or Q_2. The vertex at the front of Q_2 can only be added to the solution if it allows to reach a solution better than any previously visited solution.

In order to properly maintain both these queues, vertices which are removed from the solution should be moved into Q_2. Once they are no longer tabu, they should be move from Q_2 to Q_1. When the algorithm changes the priority of a vertex v, Increase-Key or Decrease-Key must be used on the queue which contains v. This can be determined by checking whether or not v is currently tabu.

Algorithm 11 Tabu search based on the PUSH operator, implemented with a priority queue

Input: A graph $G = (V, E)$
Output: An independent set $S*$ with weight $W*$

$S^* \leftarrow \emptyset$
$W^* \leftarrow 0$
$S \leftarrow \emptyset$
$W \leftarrow 0$
{Initialize tabu list}
for $v \in V$ **do**
  $T_v \leftarrow 0$
end for
{Initialize priority queue}
$Q \leftarrow$ EMPTY-QUEUE()
for $v \in V$ **do**
  ENQUEUE(Q, v)
end for
$i \leftarrow 0$ {The current iteration number}
$i^* \leftarrow 0$ {Iteration number when the best known solution was found}
while stopping condition not met **do**
  for v such that $T_v = i$ and $v \notin S$ **do**
    INCREASE-KEY(Q, v) {Increase the priority of vertices that are no longer tabu}
  end for
  if $i^* - i \geq L$ **then**
    $S \leftarrow$ RESTART(G, S) {Restart if we have not been able to improve the solution}
    $W \leftarrow \sum_{v \in S} w_v$
    $Q \leftarrow$ EMPTY-QUEUE() {Reset priority queue}
    for $v \in V \setminus S$ **do**
      ENQUEUE(Q, v)
    end for
    $i^* \leftarrow i$
  end if
  $v \leftarrow$ DEQUEUE(Q) {Pick the best non-tabu vertex to push}
  for $u \in N_G(v) \setminus S$ **do**
    DECREASE-KEY(Q, u) {$\delta_{\text{PUSH}(u)}$ decreased for neighbors of the added vertex}
  end for
  for $u \in (S \cap N_G(v))$ **do**
    $d \leftarrow$ a random integer between 7 and $7 + |\{v \in V | \deg(v) = \deg(u)\}|$
    $T_u \leftarrow i + d$ {Mark removed vertices as tabu}
    ENQUEUE(Q, u) {Removed vertices are now eligible to be pushed}
    for $x \in N_G(u) \setminus \{v\}$ **do**
      INCREASE-KEY(Q, x) {$\delta_{\text{PUSH}(x)}$ increased for neighbors of removed vertices}
    end for
  end for
  $W \leftarrow W + \delta_{\text{PUSH}(v)}$
  $S \leftarrow (S \cup \{v\}) \setminus N_G(v)$ {Perform PUSH(v)}
  if $W > W^*$ **then**
    $S^* \leftarrow S$ {Store the new best solution}
    $W^* \leftarrow W$
    $i^* \leftarrow i$
  end if
  $i \leftarrow i + 1$
end while

3.4 An Iterative Approach to Improve upon the Initial Solution

The local search algorithm presented in section 3.3 is well suited even for large graphs, and can find a reasonable solution in a short amount of time. However, it quickly becomes very difficult for it to find ways to improve the solution.

In this section, a way to combine local search algorithms with the exact resolution methods presented in section 3.2 is considered. This approach uses the Large Neighborhood Search framework, presented in subsection 3.4.1. A formulation of the maximum weight independent set problem within this framework is presented in subsection 3.4.2. The two majors components of such a formulation are an algorithm to select fragments and an algorithm to re-optimize them. The size of these fragments is an important parameter of the algorithm, and a strategy to adjust it dynamically is presented in subsection 3.4.3.

The benefit of this approach, in addition to being more effective at improving solutions obtained from local search, is that it can easily be implemented in parallel (subsection 3.4.4).

3.4.1 The Large Neighborhood Search Framework

The Large Neighborhood Search (LNS) was introduced first to solve the Vehicle Routing Problem [35]. LNS is a type of Local Search algorithm which explores a larger neighborhood by using an exact resolution algorithms such as branch and bound.

Consider a solution which is defined by giving an assignment to a set of variables $x_1, \ldots, x_n$. At each iteration, the algorithm selects a subset of those parameters, henceforth referred to as a *fragment*. The second step of each iteration is to *re-optimize* them by finding the best solution such that all parameters not in the selected fragment keep the value they had in the current solution, whereas the parameters within the fragment are *relaxed*, i.e. allowed to take any possible value.

The optimization step is much more expensive than the move selection of more typical local search algorithm. The idea is to consider a larger neighborhood (whose size is typically exponential in the size of the selected fragments), so as to improve quality of local optima reached by the algorithm [31].

The algorithm thus requires two important components:

1. A fragment selection algorithm which selects a subset of the solution parameters. It can be desirable for the parameters within a fragment to be somehow related, so that it makes sense to consider all of them simultaneously when optimizing the solution.

2. A neighborhood exploration algorithm which finds the best solution that can be obtained when relaxing the selected fragment.

The following subsections detail how these algorithms can be implemented for the maximum weight independent set problem.

3.4.2 LNS Formulation of the Maximum Weight Independent Set Problem

Consider the maximum weight independent set problem. Any independent set S can be represented by using one Boolean variable x_i for each vertex v_i in the graph. x_i is set to true iff $v_i \in S$.

The fragment selection algorithm of an LNS formulation of the problem can therefore select a subset X of the vertex set V. As mentioned in the previous subsection, it is desirable for elements of this subset to be somehow related. In this case, the length of the shortest path between two vertices can be used as a measure of how related two vertices are: vertices that are close to each other are related. When considering the application to indirect mesh generation, it makes sense that the way in which some tetrahedra will be combined heavily influences the ways

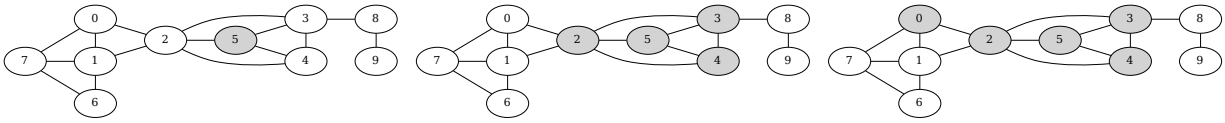


Figure 3.9: Generation of a fragment of size 5.

that adjacent tetrahedra can be combined, but has no direct impact on the decision made for tetrahedra located in a completely different part of the mesh.

The following procedure can be used to select a fragment of up to n vertices: first pick a random vertex $v \in V$, then perform a breadth first traversal of the graph starting from v. Each time a vertex is visited by the algorithm, it is added to the fragment X, until either n vertices have been selected, or the entire graph has been traversed. In other words, the algorithm will first select v, then vertices at a distance of 1 of v, followed by vertices at a distance of 2, etc. until n vertices have been selected. Figure 3.9 illustrates an execution of this procedure.

Given such a set X and the current solution S, the second step is to find a new independent set S', by only changing which subset Y of X should belong to the solution, while maximizing the weight of S'. This optimization problem can be written as:

$$\begin{gathered}\arg\max_{Y \subseteq X} \sum_{v_i \in S \setminus X} w_i + \sum_{v_i \in Y} w_i \\ \text{such that } S' = (S \setminus X) \cup Y \text{ is an independent set}\end{gathered}$$

The fact that S' must be an independent set imposes the following conditions on Y:

1. There can't be any edge between an element that will remain in the solution and an element that will be added to it:

$$\forall v \in (S \setminus X).\, \forall u \in Y.\, (u, v) \notin E$$

2. There can't be any edge between two newly selected elements:

$$\forall u, v \in Y.\, (u, v) \notin E$$

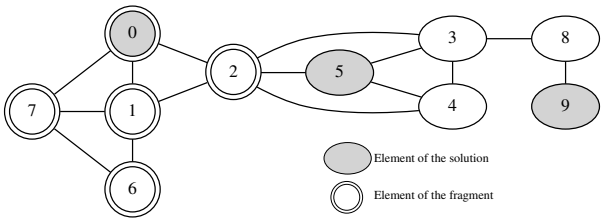
The first condition can be verified by only considering vertices which are not adjacent to a vertex belonging to the solution and not the selected fragment. In other words, Y must be chosen as a subset of

$$X' = \{v \in X \mid N_G(v) \cap (S \setminus X) = \emptyset\}$$

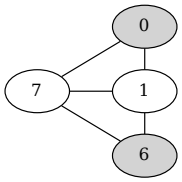
In order to also satisfy the second condition, Y must also be an independent set of the induced subgraph $G_{X'}$, i.e. the subgraph whose vertex set is X' and whose edge set is the set of edges that connect elements of X'.

Since the objective is to maximize the total weight of S', the total weight of Y should also be maximized. In other words, the fragment X can be re-optimized by finding a maximum weight independent set of the smaller graph $G_{X'}$. This can be done using the branch and bound algorithms presented in section 3.2. Figure 3.10 shows a possible execution of an iteration of the algorithm.

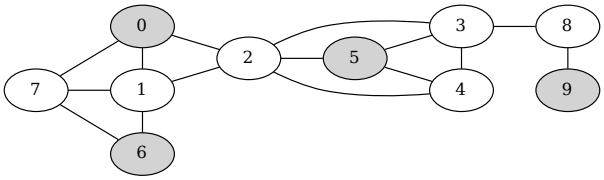
In practice, the initial solution can be computed by using the local search algorithms presented in section 3.3, and stopping them after a few restarts. The benefit of computing an initial solution in this manner is that it provides a good initial lower bound for any potential fragment, reducing the time needed to re-optimize them [39].



(a) Initial solution



(b) Re-optimization step



(c) Updated solution

Figure 3.10: An iteration of the LNS algorithm

3.4.3 Adaptive Adjustment of the Fragment Size

It is important to pick a good value for the size n of the fragments which the algorithm relaxes and re-optimizes. Since selecting such a value before running the algorithm can prove difficult, a strategy which automatically adjusts n is considered. The parameter for this strategy instead becomes the desired computation time T.

The first observation made to design this strategy is that selecting larger fragments allows better solutions to be found. Secondly, exploring. a previously explored fragment X generally does not improve the solution as much as exploring a new fragment. Indeed, exploring each fragment more than once is not as effective as simply increasing the size of fragments and only exploring them once. The overlap between fragment even makes it worthwhile to only explore a portion of all possible fragment, and instead use larger fragments.

Note that when the size of fragments is fixed, the amount of possible fragments is equal to the amount of vertices in the graph: there is one fragment per vertex which can be selected as a starting point. Therefore, our strategy (Algorithm 12) attempts to explore $N = \gamma|V|$ fragments in a given amount of time T, where $\gamma \in [0, 1]$ is a parameter controlling how many fragments are visited. The size of future fragments is readjusted each time a multiple of M fragments have been explored by the algorithm. The algorithm estimates the time it would take to explore the remaining fragments based on the time it took to explore the previous M fragments. If this estimate is smaller than (resp. greater than) the time, then the algorithm increases (resp. decreases) the fragment size. This attempts to use fragments as large as possible while still finishing in time.

3.4.4 Parallel Implementation

One advantage of this approach is that it is possible to adapt the algorithm to run on parallel architectures by exploring multiple fragments in parallel.

Consider two fragments X_1 and X_2. The algorithm will re-optimize X_1 and X_2 in parallel and compute Y_1 and Y_2. If S was the original solution, the new solution reached when both of

Algorithm 12 Automatic adjustment of the fragment size

p

Input: T the amount of time left for the algorithm to run; ΔT the amount of time elapsed since the algorithm was last run; M the amount of fragments explored in that time; $N_{\text{unexplored}}$ the amount of fragments that must still be explored by the algorithm.

1: $T_{\text{avg}} \leftarrow \frac{\Delta T}{M}$
2: $T_{\text{left}} \leftarrow N_{\text{unexplored}} \times T_{\text{avg}}$
3: **if** $T_{\text{left}} < T$ **then**
4: Increase the size of fragments by a small constant
5: **else if** $T_{\text{left}} > T$ **then**
6: Decrease the size of fragments by a small constant
7: **end if**

these fragments have been explored will be

$$S' = ((S \setminus X_1) \setminus X_2) \cup Y_1 \cup Y_2$$

This approach cannot possibly work if X_1 and X_2 overlap (Figure 3.11a). Even if they do not overlap, but there is a pair of vertices $v_i \in X_1$ and $v_j \in X_2$ such that v_i is adjacent to v_j (Figure 3.11b), it is not safe to explore the two fragments in parallel: if $v_i \in Y_1$ and $v_j \in Y_2$, S' is not an independent set.

When the two fragments do not overlap and do not contain adjacent vertices (Figure 3.11c), it is always safe to explore them in parallel, as the solution S' will always be an independent set:

- there is no edge between two elements $(S \setminus X_1) \setminus X_2$, because S is an independent set;
- there is no edge between an element $(S \setminus X_1) \setminus X_2$ and an element of Y_1 or Y_2 due to the conditions imposed on these sets in subsection 3.4.2;
- there is no edge between an element of Y_1 and an element of Y_2, as these are subset of X_1 and X_2 respectively;
- there is no edge between two elements of Y_1 (or two elements of Y_2), because the algorithm that finds them looks only for independent sets.

Let us consider a scenario where a single thread generates new fragments and puts them in a queue (Algorithm 13). These fragments are then explored in parallel by multiple workers (Algorithm 14).

To allow parallel exploration of all fragments, the algorithm needs to track the set of vertices belonging to or adjacent to an *active* fragment, i.e. a fragment currently in the queue or being explored by one of the worker threads. Vertices belonging to this set cannot belong to newly generated fragments, hence the fragment selection algorithm must ignore them when selecting a starting vertex or performing the breadth first traversal.

This set is tracked by maintaining for each vertex v_i a counter c_i of how many active fragments it belongs to or is adjacent to. While a vertex can only belong to one active fragment, it can also belong to none while being adjacent to multiple fragments. For example, in Figure 3.11c, vertex 2 is adjacent to 2 fragments. Algorithm 15 takes care of updating this counter, both when a fragment gets enqueued and when it has been explored. The update of each counter needs to be performed atomically as multiple worker threads may update the same counter simultaneously.

When a worker thread has re-optimized a fragment, the solution must be updated to reflect it. This is accomplished using Algorithm 16. The solution S is represented as an array of Boolean values: S_i is true iff $v_i \in S$. Since fragments explored in parallel do not overlap, two worker threads running the update in parallel will never simultaneously access the same element of the solution. It is therefore safe for multiple worker threads to update the solution in parallel.

There is also no risk for the work generation thread to create a fragment containing a vertex v_i adjacent to another vertex v_j which was added to the solution by a worker thread as it was updating the solution. The reason this cannot happen happen is because v_i is adjacent to an active fragment and the worker threads only update the counters after updating the solution.

Algorithm 13 Pseudo-code of the thread selecting fragments

Input: A concurrent queue Q containing fragments, an array of counters $\mathbf{c}$

```
1: while stopping criterion is not met do
2:     X ← Select-Fragment()
3:     Increase-Counters(X, c, 1)
4:     Enqueue(Q, X)
5: end while
6: Close(Q)
```

Algorithm 14 Pseudo-code of a worker thread exploring fragments

Input: A concurrent queue Q containing fragments, an array of counters $\mathbf{c}$, and the current solution S

```
1: while Q is not closed do
2:     X ← Dequeue(Q)
3:     Y ← Re-Optimize(X)
4:     Update(S, X, Y)
5:     Increase-Counters(X, c, −1)
6: end while
```

Algorithm 15 Procedure used to increase counters

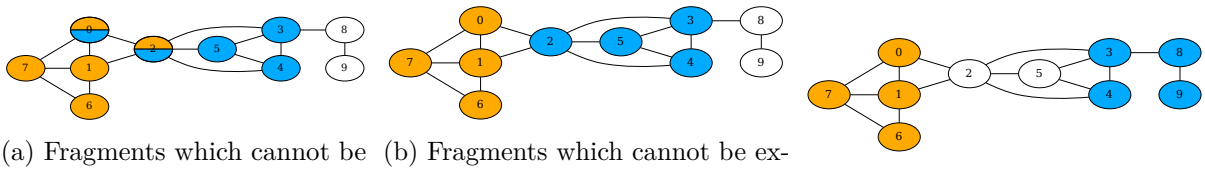
Input: A fragment X, an array of counters $\mathbf{c}$, and an integer Δ

```
1: N ← X ∪ (⋃_{v_i ∈ X} N_G(v_i))
2: for each vertex v_i ∈ N do
3:     Atomically increase c_i by Δ
4: end for
```

Algorithm 16 Procedure updating the solution after optimizing a fragment

Input: The current solution S as an array of Boolean values, a fragment X and a select subset Y

1: **for** $v_i \in X$ **do**
2: **if** $v_i \in Y$ **then**
3: $S_i \leftarrow$ **true**
4: **else**
5: $S_i \leftarrow$ **false**
6: **end if**
7: **end for**



(a) Fragments which cannot be explored in parallel because of overlap.

(b) Fragments which cannot be explored in parallel because they are connected by an edge.

(c) Two fragments which can be explored in parallel.

Figure 3.11: Fragments which can or cannot be explored in parallel.

Chapter 4

Computational Experiments

In this chapter, we analyze the efficacy of the algorithm described in section 3.4 and its components. Specifically, we look at branch and bound algorithms used to explore small subgraphs (section 4.1) and local search algorithm to compute an initial solution (section 4.2), before looking at the complete algorithm (section 4.3).

A summary of the meshes which were used for these tests is shown in Table 4.1. In this table, the amount of vertices and tetrahedra refer to the original tetrahedral mesh, whereas the amount of potential hexahedra refer to how many hexahedra can be obtained by recombining the mesh's tetrahedra, i.e. it is the number of vertices in the incompatibility graph. The size of this graph depends on the minimum quality threshold used when searching for potential hexahedra.

All timings were obtained by executing the algorithms on an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with 16 GB of RAM. Our code was compiled with GCC 6.3.1 (with the optimization flag `-O3` and all architecture-specific optimizations enabled). lpsolve 5.5.1 was used to solve linear programs. The C++ implementation of our method was contributed as part of the Gmsh project (`http://www.gmsh.info/`).

4.1 Exact Resolution for Small Graphs

As part of the algorithm presented in section 3.4, algorithms which find the optimal solution for small graphs, such as those described in section 3.2 are used. Table 4.2 lists the exact algorithms that are evaluated in this section. All these algorithms find a solution of the same weight. The following metrics will be considered to evaluate how effective they are at finding such a solution:

- the time to complete the search;
- the amount of branching nodes in the search tree.

A good algorithm needs to make a good compromise between the amount of computation time spent at each node of the search tree and the amount of nodes it needs to expand.

Input graphs of various sizes were generated using the fragment generation algorithm described in subsection 3.4.2, as this is the context in which these algorithms will be used in practice.

Figure 4.1c shows how long it takes for the algorithm to find the optimal solution for subgraphs of various sizes. Each sample is the average execution time when running the algorithm on 20 different subgraphs of the same size. The amount of branching nodes within the search tree is shown in Figure 4.1d.

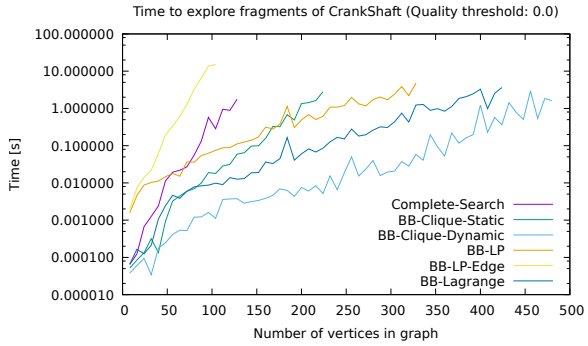
The general trend shown is that more expensive upper bounding procedures are more worthwhile for larger, sparser instances. The **Complete-Search** quickly becomes far too expensive to be used in practice, with fragments of size less than hundred needing multiple seconds to be explored.

Name	# vertices	# tetrahedra	Quality threshold	# potential hexahedra
CrankShaft	23,246	104,302	0.0	304,409
			0.1	232,455
			0.2	186,368
			0.6	27,217
CrankShaft2	136,351	717,149	0.2	1,980,972
CrankShaftSmall	10,818	42,223	0.6	7272
Fusee	11,975	50,750	0.7	7716
Fusee_1	71,947	349,893	0.1	1,278,262
Caliper	130,572	675,289	0.2	1,597,116
random1	3,000	16,915	0.2	25,962
random2	10,000	61,402	0.4	30,575
random3	20,000	124,958	0.5	23,517
random4	100,000	646,968	0.1	1,821,002
			0.7	5,453

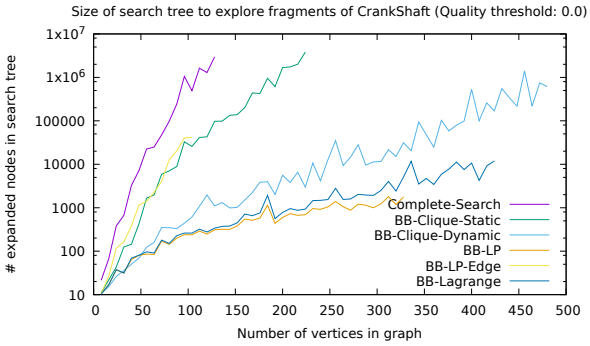
Table 4.1: Size metrics for the meshes used to measure the efficacy of the algorithm. The first two columns are properties of properties of the input mesh, while the last two refer to the incompatibility graph generated by the algorithm.

Algorithm Name	Description
Complete-Search	A complete search algorithm, without computing any upper bound for pruning purposes.
BB-Clique-Static	Branch and bound algorithm using an upper bound computed with a clique partition for the entire graph.
BB-Clique-Dynamic	Same as **BB-Clique-Static**, but the clique partition is recomputed at every node of the search tree.
BB-LP-Edge	Branch and bound algorithm based on a linear relaxation computed for the edge formulation of the problem.
BB-LP	Branch an bound algorithm using an upper bound computed with a linear relaxation computed for the clique formulation of the problem.
BB-Lagrange	Branch an bound algorithm using an upper bound computed with a Lagrangian relaxation computed for the clique formulation of the problem.

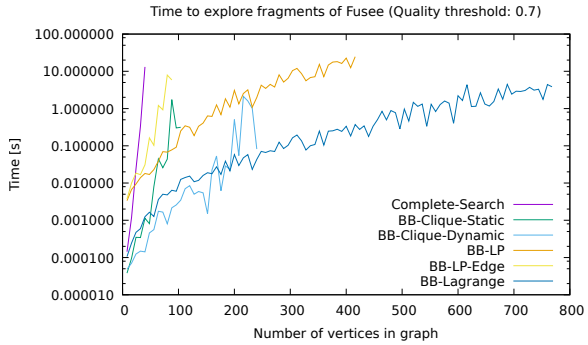
Table 4.2: Description of the evaluated exact resolution algorithms.



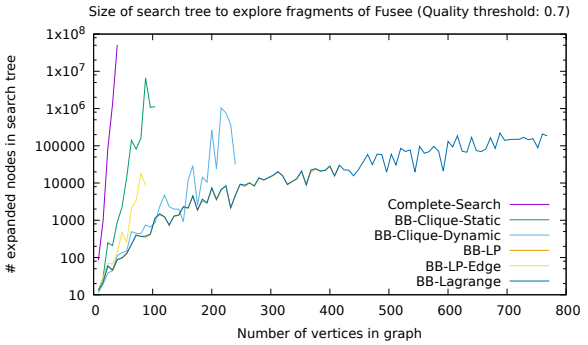
(a) Time required to explore subgraphs of CrankShaft



(b) Size of the search trees required to explore subgraphs of CrankShaft

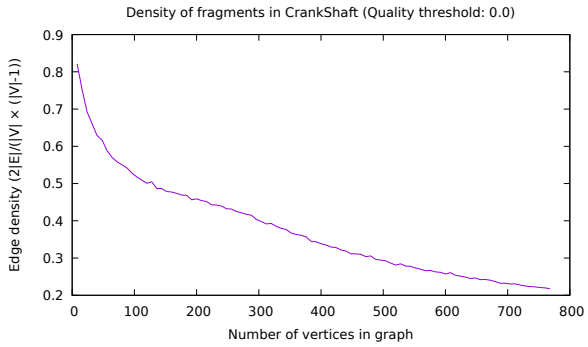


(c) Time required to explore subgraphs of Fusee

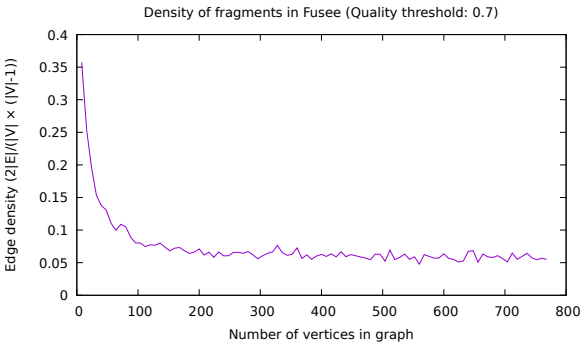


(d) Size of the search trees required to explore subgraphs of Fusee

Figure 4.1: Exact resolution applied to subgraphs of a few instances



(a) Edge density of fragments when the quality threshold is low



(b) Edge density of fragments when the quality threshold is high

Figure 4.2: Edge density for fragments of different sizes

Algorithm Name	Description
Greedy	Greedily constructs a solution by iteratively adding a vertex of highest weight which is compatible with all previously selected vertices.
ReTSQ	A local search based on the PUSH operator, storing candidate vertices in two priority queues (as described in section 3.3).
ReTS1	A local search based on the PUSH operator, selecting the best move by iterating over all candidate vertices, as described in [47]. We used the authors' implementation [16].
ReTS2	A local search based on the PUSH operator, using the Best from Multiple Selection heuristic to explore part of the neighborhood, as described in [47]. We used the authors' implementation [16].

Table 4.3: Algorithms used to compute an initial solution.

Approaches based on linear programming significantly reduce the size of the search tree, especially clique based formulations. The Lagrangian approach can be used to reduce the time spent computing them while only slightly increasing the size of the search tree for easier instances, and still being faster for larger ones.

Upper bounds based on clique partitions are not as tight, but can be more effective overall due to the fact that they are so cheap to compute: **BB-Clique-Dynamic** is usually the fastest approach, except for graphs with a very low density. For those, spending more time computing tighter upper bounds becomes worthwhile. In practice, such subgraphs are only generated if the quality threshold is high. If the quality threshold is lower, sparse fragments must also be very large (Figure 4.2). This makes it too expensive to find the optimal solution for such subgraphs no matter which algorithm is used, hence **BB-Clique-Dynamic** will be preferred overall.

4.2 Computation of an Initial Solution

At the beginning of the algorithm, a fast method is used to compute an initial solution, which can then be improved. For this purpose, a few algorithms have been considered (Table 4.3).

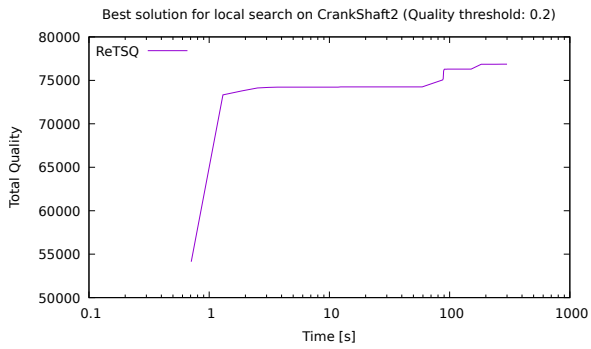
All the local search algorithms were run with the same parameters: the number of iterations before a restart occur are set to $L = 2|V|$ and the probability to reconstruct the solution to $\rho = 0.7$. When the BMS heuristic is used, the number of vertices sampled by the algorithm is set to 50.

Table 4.4 shows how effective the solutions found by those different algorithms are. Local search algorithms usually find better solutions than the greedy approach, especially when the quality threshold is lower. The effectiveness of the different local search strategies depends on how quickly they can perform an iteration: for the same formulation, algorithms which perform more iterations per second are able to find better solutions in the same amount of time. The BMS heuristic usually helps, but maintaining a priority queue is even less costly, at least for larger graphs.

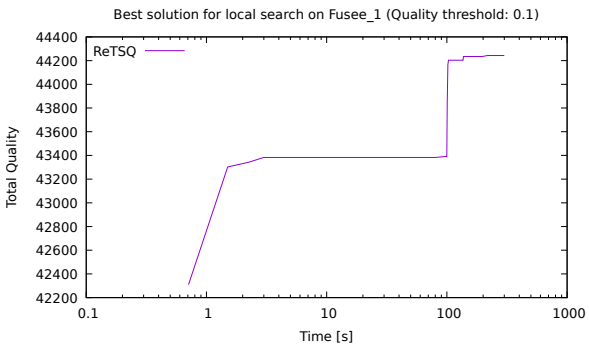
Those results were obtained by running the different algorithms with a time limit of 5 minutes, which is far more than the time required for the greedy algorithm. Figure 4.3 shows that a solution almost as good as the final solution is typically reached within seconds, even on graphs with millions of vertices. The time between successive improvements on the solution keeps growing as the algorithm is allowed to run for longer periods of time. This motivates the approach presented in section 3.4 to more quickly improve the solution.

Agorithm	# iterations	Best Quality
CrankShaft (Quality threshold: 0.6)		
Greedy	N/A (time: 55.45 ms)	6644.48
ReTS1	13100000	6920.835
ReTS2	15500000	6941.673
ReTSQ	200115766	7017.55
Fusee (Quality threshold: 0.7)		
Greedy	N/A (time: 18.34 ms)	3852.41
ReTS1	55400000	3876.106
ReTS2	52800000	3876.146
ReTSQ	408026890	3885.11
random1 (Quality threshold: 0.2)		
Greedy	N/A (time: 28.31 ms)	299.887
ReTS1	29700000	443.762
ReTS2	49700000	447.198
ReTSQ	40684894	447.085
random2 (Quality threshold: 0.4)		
Greedy	N/A (time: 34.49 ms)	1201.47
ReTS1	20600000	1536.335
ReTS2	26400000	1531.957
ReTSQ	85968483	1542.1
random3 (Quality threshold: 0.5)		
Greedy	N/A (time: 35.11 ms)	2282.17
ReTS1	21500000	2590.162
ReTS2	16100000	2583.613
ReTSQ	153919729	2626.79
random4 (Quality threshold: 0.7)		
Greedy	N/A (time: 17.09 ms)	3076.68
ReTS1	92100000	3074.235
ReTS2	54200000	3076.265
ReTSQ	571089302	3076.93
CrankShaftSmall (Quality threshold: 0.6)		
Greedy	N/A (time: 11.94 ms)	1648.34
ReTS1	60800000	1737.512
ReTS2	63600000	1737.486
ReTSQ	273074113	1762.01
CrankShaft2 (Quality threshold: 0.2)		
Greedy	N/A (time: 3.37 s)	75340.34
ReTS1	OOM	
ReTS2	OOM	
ReTSQ	15783000	76864.7
Fusee_1 (Quality threshold: 0.1)		
Greedy	N/A (time: 1.51 s)	43366.4
ReTS1	OOM	
ReTS2	OOM	
ReTSQ	12204000	44242.7

Table 4.4: Results of a algorithms used to compute an initial solution. Local search algorithms were run with a time limit of 300 seconds. OOM indicates that the algorithm ran out of memory.



(a) Results obtained for CrankShaft2



(b) Results obtained for Fusee_1

Figure 4.3: Evolution over time of the best solution found by local search

Time limit	Fragment size						
	64	128	192	256	320	384	448
1.0 s	$0.12\|V\|$	$0.07\|V\|$	$0.05\|V\|$	$0.04\|V\|$	$\mathbf{0.02}\|\mathbf{V}\|$	$0.00\|V\|$	$0.00\|V\|$
5.0 s	$0.62\|V\|$	$0.36\|V\|$	$0.25\|V\|$	$0.19\|V\|$	$\mathbf{0.10}\|\mathbf{V}\|$	$0.02\|V\|$	$0.01\|V\|$
10.0 s	$1.24\|V\|$	$0.72\|V\|$	$0.51\|V\|$	$0.38\|V\|$	$\mathbf{0.20}\|\mathbf{V}\|$	$0.03\|V\|$	$0.01\|V\|$
50.0 s	$6.22\|V\|$	$3.64\|V\|$	$2.56\|V\|$	$1.87\|V\|$	$0.86\|V\|$	$\mathbf{0.18}\|\mathbf{V}\|$	$0.03\|V\|$
200.0 s	$24.79\|V\|$	$14.93\|V\|$	$10.49\|V\|$	$7.58\|V\|$	$2.00\|V\|$	$\mathbf{0.46}\|\mathbf{V}\|$	$0.09\|V\|$

Table 4.5: Amount of iterations performed by LNS on CrankShaft, for different fragment sizes. Bold entries correspond to optimal fragment size in Figure 4.5a

4.3 Evaluation of the Large Neighborhood Search Algorithm

An algorithm based on the LNS framework is used to further improve the solution, as described in section 3.4. **ReTSQ** is used to construct an initial solution, and **BB-Clique-Dynamic** is used to re-optimize fragments.

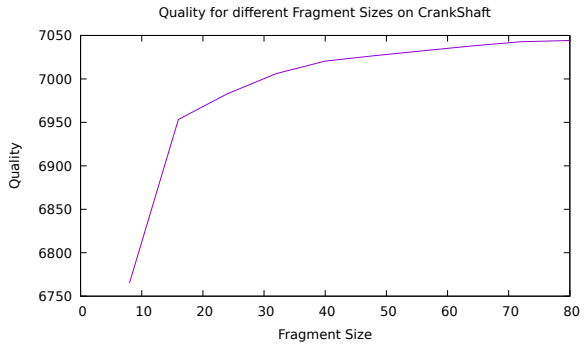
Figure 4.4 shows the local optimum that can be obtained using the LNS algorithm as a function of the fragment size. This was obtained by starting with an empty solution, then selecting and optimizing all possible fragments, since there are only $|V|$ possible starting points. This process was repeated until the solution stopped improving. This illustrates the fact that the qualities of local optima increases with the size of the fragments considered by the algorithm.

While this means that better solutions can be reached by using larger fragments, it is not practical to wait for a local optimum to be reached before stopping the algorithm. Figure 4.5 shows that increasing the size of fragments eventually decreases the quality of the solution reached by the algorithm within a certain time limit. Increasing the time limit increases the optimal fragment size.

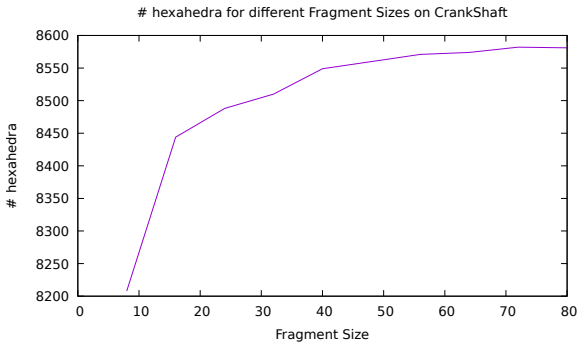
Combining this with the amount of iterations performed by the algorithm (Tables 4.5 and 4.6) shows that the fragment size from which results start to deteriorate is such that the algorithm does not have enough time to visit all possible fragments of this size.

The use of local search allows an initial lower bound to be computed for any fragment. This significantly reduces the time required to re-optimize them, especially when they are larger (Figure 4.6b). In addition, even for small fragments, local search is more flexible and can find solutions that can't be obtained by only re-optimizing subgraphs. This allows the combination of both algorithms to ultimately reach a better solution than either algorithm separately (Figure 4.6a).

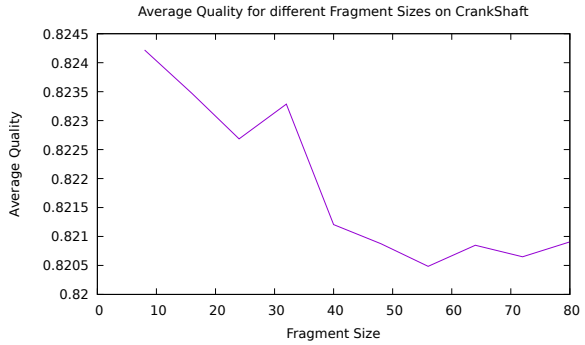
Adapting the size of fragments automatically allows the algorithm to pick good settings without requiring guesses to be made. Figure 4.7 shows results obtained by the algorithm trying



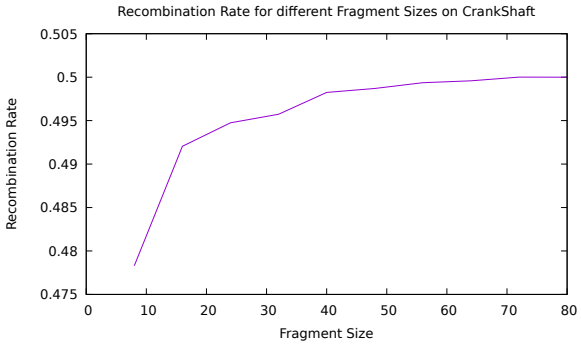
(a) Total quality of the solution



(b) Amount of hexahedra in the recombined mesh

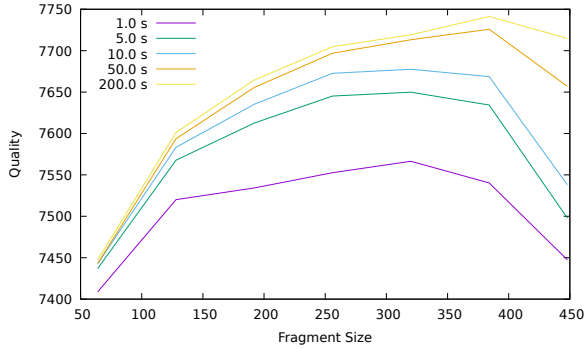


(c) Average quality of selected hexahedra

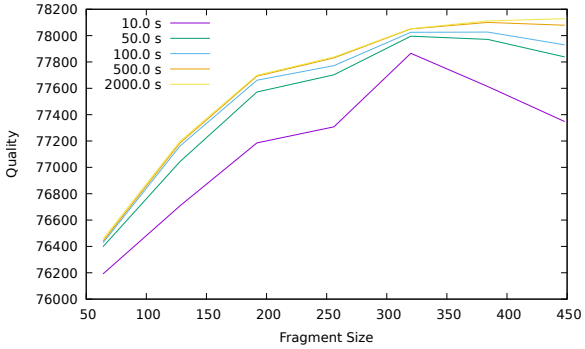


(d) Recombination rate of tetrahedra

Figure 4.4: Local optima reached by LNS algorithm for CrankShaft



(a) Results obtained for CrankShaft, with a quality threshold of 0.0.

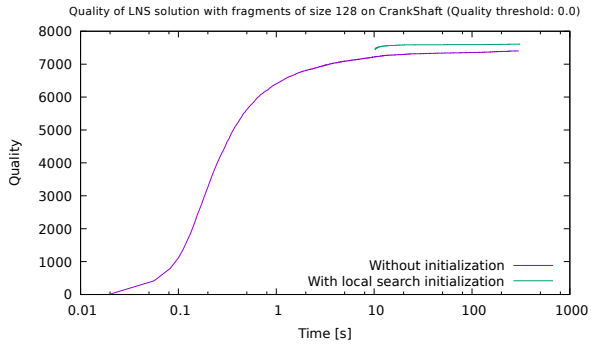


(b) Results obtained for CrankShaft2, with a quality threshold of 0.2.

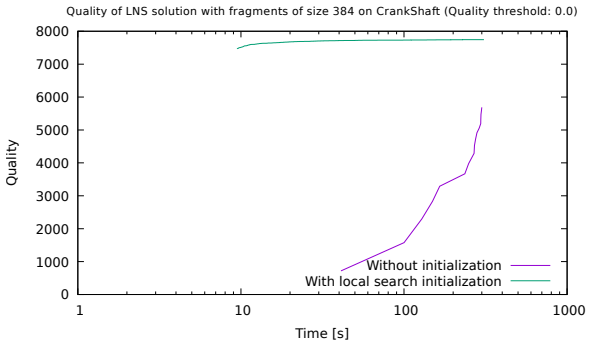
Figure 4.5: Quality of the solution as a function of the fragment size. Timings do not include the time spent computing an initial solution (10 s).

Time limit	Fragment size						
	64	128	192	256	320	384	448
10.0 s	$0.17\|V\|$	$0.10\|V\|$	$0.07\|V\|$	$0.05\|V\|$	$\mathbf{0.04}\|\mathbf{V}\|$	$0.03\|V\|$	$0.01\|V\|$
50.0 s	$0.86\|V\|$	$0.51\|V\|$	$0.33\|V\|$	$0.25\|V\|$	$\mathbf{0.20}\|\mathbf{V}\|$	$0.14\|V\|$	$0.04\|V\|$
100.0 s	$1.72\|V\|$	$1.02\|V\|$	$0.66\|V\|$	$0.50\|V\|$	$0.40\|V\|$	$\mathbf{0.28}\|\mathbf{V}\|$	$0.08\|V\|$
500.0 s	$8.68\|V\|$	$5.15\|V\|$	$3.31\|V\|$	$2.51\|V\|$	$2.03\|V\|$	$\mathbf{1.18}\|\mathbf{V}\|$	$0.33\|V\|$
2000.0 s	$35.40\|V\|$	$20.82\|V\|$	$13.54\|V\|$	$10.24\|V\|$	$8.25\|V\|$	$4.43\|V\|$	$\mathbf{1.00}\|\mathbf{V}\|$

Table 4.6: Amount of iterations performed by LNS on CrankShaft2, for different fragment sizes. Bold entries correspond to optimal fragment size in Figure 4.5b

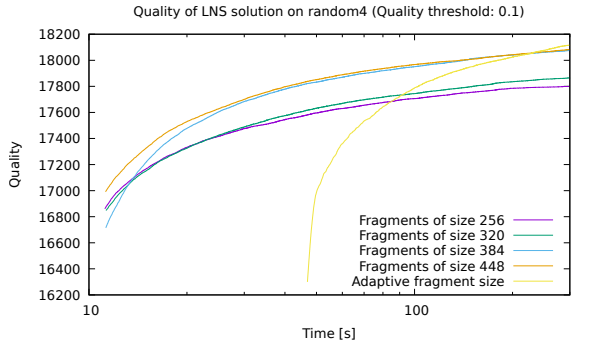


(a) Effect of computing an initial solution when optimizing smaller fragments

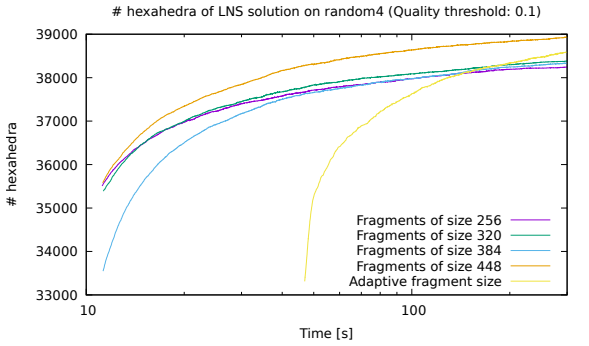


(b) Effect of computing an initial solution when optimizing larger fragments

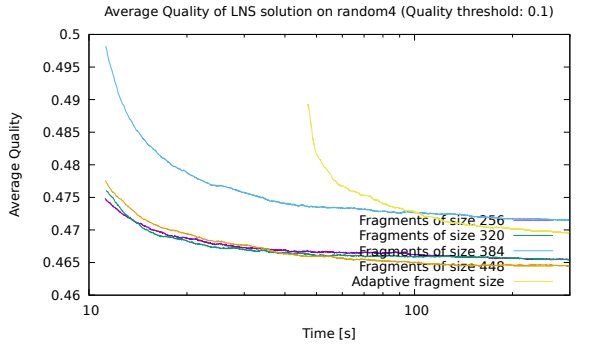
Figure 4.6: Quality of the solution over time, with and without using local search to compute an initial solution.



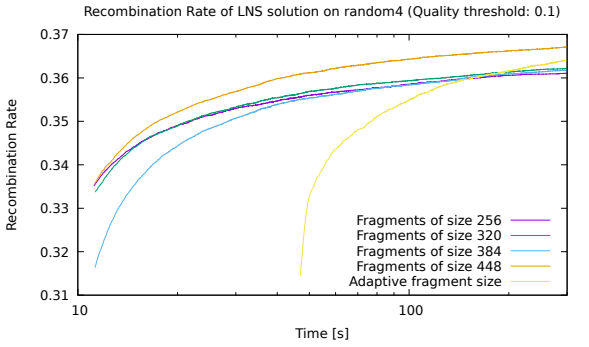
(a) Total quality of the solution



(b) Amount of hexahedra in the recombined mesh

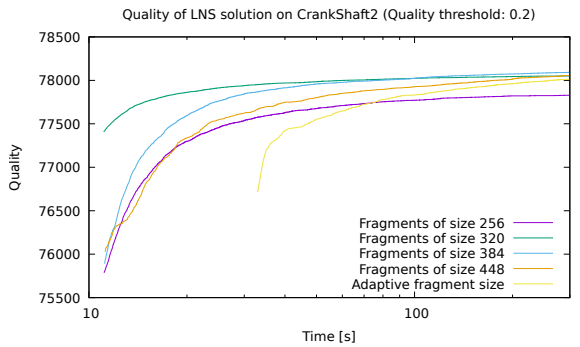


(c) Average quality of selected hexahedra

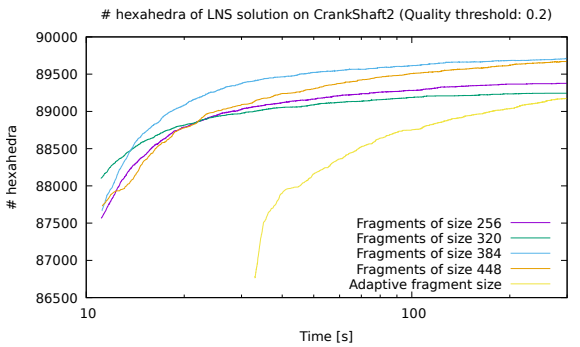


(d) Recombination rate of tetrahedra

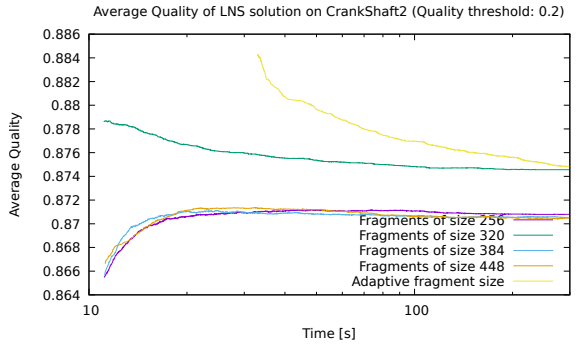
Figure 4.7: Solution obtained by the LNS algorithm for random4, using an adaptive fragment size.



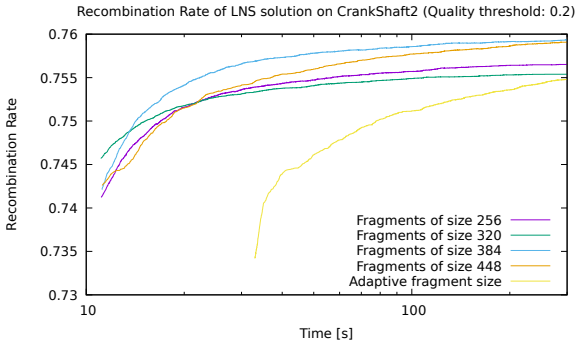
(a) Total quality of the solution



(b) Amount of hexahedra in the recombined mesh



(c) Average quality of selected hexahedra



(d) Recombination rate of tetrahedra

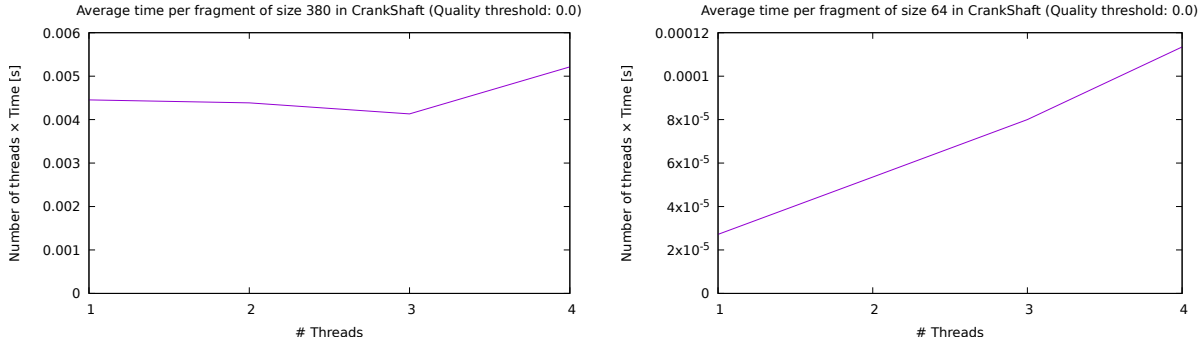
Figure 4.8: Solution obtained by the LNS algorithm for CrankShaft2, using an adaptive fragment size.

to visit $0.25|V|$ fragments by adapting their size by steps of 8 every 500 iterations. While this strategy simplifies the process of adjusting the parameters of the algorithm, it does not always reach the best possible solution given a certain time budget (Figure 4.8, obtained with the same settings). Note also

The algorithm also has the benefit of running in parallel. Figure 4.9a shows that the algorithm can achieve close to linear scaling up to the point where all physical cores are used. Because there is always one thread dedicated to generating fragments in addition to the worker threads, all 4 cores are already being used when 3 worker threads are active. This explains the worse scaling when adding a 4th worker thread. Such linear scaling is only possible if the single producer can keep up with the worker threads. If they are too fast (e.g. because the fragment size is low), adding more of them does not improve performances (Figure 4.9b).

Note that while a large number of potential hexahedra have a very low quality (Figure 4.10a), hexahedra belonging to the solution obtained by any of the algorithms we described are more likely to have a higher quality (Figure 4.10). Setting a very low quality threshold increases the size of the graph and the time needed to obtain a good solution, which can lower the quality of the solution obtained within a given time frame (Figure 4.11).

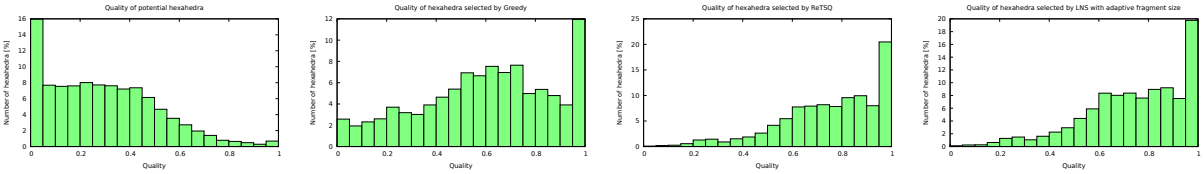
Lastly, Figure 4.12 shows some of the meshes generated by the algorithm.



(a) Linear scaling obtained when the producer can generate fragments fast enough.

(b) No scaling when the fragments are too small and quickly explored.

Figure 4.9: Parallel performances of LNS implementation



(a) Qualities of all hexahedra found while generating the graph

(b) Qualities of hexahedra in the output of the greedy algorithm

(c) Qualities of hexahedra in the output of the local search algorithm

(d) Qualities of hexahedra in the output of the LNS algorithm

Figure 4.10: Qualities of hexahedra selected from CrankShaft. Each algorithm was allowed to run for 300 seconds at most.

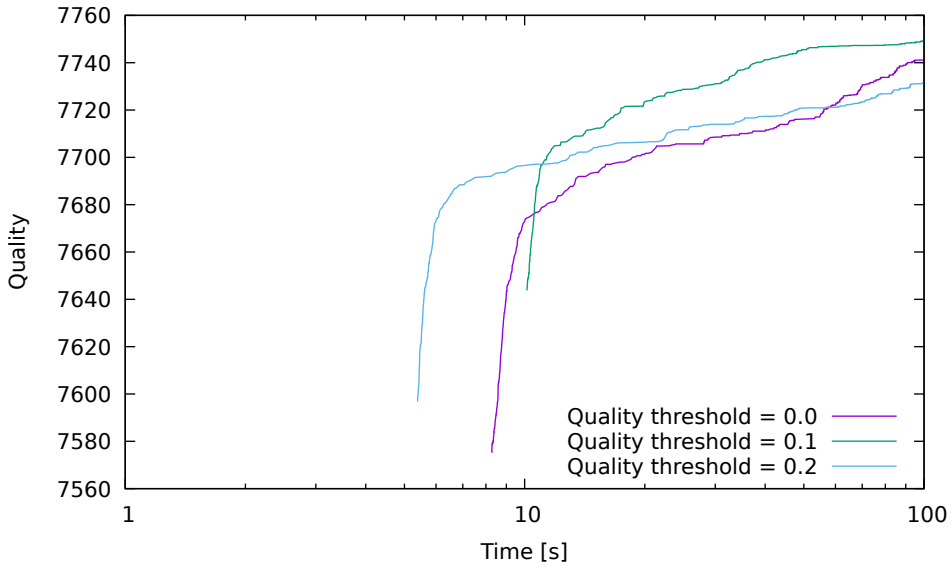
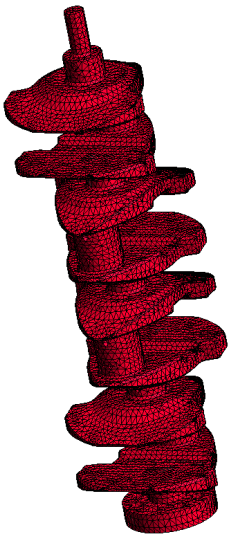
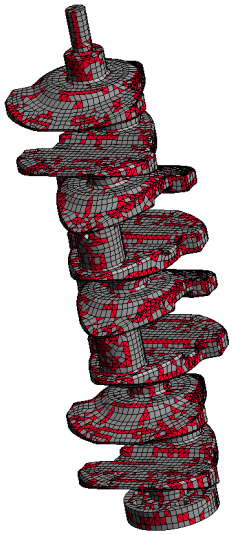


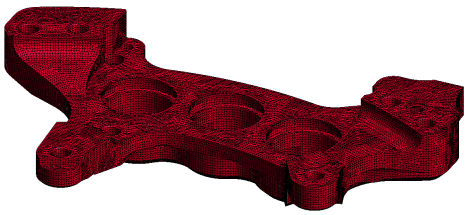
Figure 4.11: Results of LNS with adaptive fragment sizes and different quality thresholds. Decreasing this threshold can deteriorate the solution if the time limit is too low.



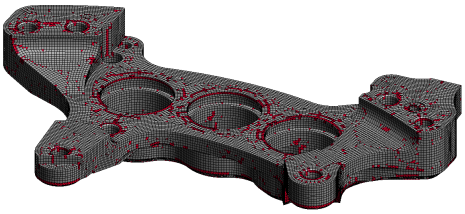
(a) Tetrahedral mesh for CrankShaft



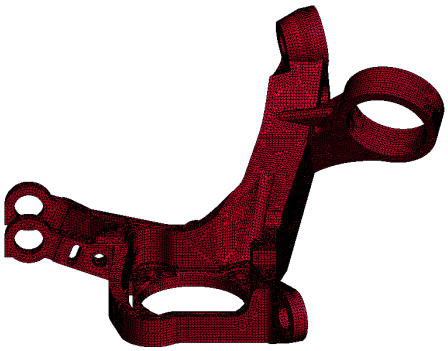
(b) Output mesh generated for CrankShaft (quality threshold: 0.0). This mesh contains 10,457 hexahedra and 36,130 tetrahedra.



(c) Tetrahedral mesh for Caliper



(d) Output mesh generated for Caliper (quality threshold: 0.2). This mesh contains 91,187 hexahedra and 119,302 tetrahedra. The worst hexahedra has a quality of 0.200762.



(e) Tetrahedral mesh for Fusee_1

(f) Output mesh generated for Fusee_1 (quality threshold: 0.1). This mesh contains 49,563 hexahedra and 48,253 tetrahedra. The worst hexahedra has a quality of 0.101.

Figure 4.12: Meshes generated by the LNS algorithm in 5 minutes with adaptive fragment size

Chapter 5

Conclusion

In the previous chapters, we have presented an iterative algorithm which solves the MWIS problem on a large graph by optimizing small subgraphs. We have shown that this method is effective on the incompatibility graphs generated as part of an indirect meshing algorithm, and can be executed in parallel with linear scaling. In addition, we have described how an existing local search formulation of the problem can be implemented efficiently using a priority queue. We have shown that this strategy is generally faster and more effective than heuristics such as BMS for graphs containing tens of thousands of vertices. We now consider some of the ways the algorithm can be improved, based on the observations made in chapter 4.

The fact that our algorithm always optimizes subgraphs to optimality leads to poor diversification of the search, which is why combining it with a local search algorithm leads to better solution being found. Other ways to combine our algorithm with local search, such as the optimization of fragments through local search algorithms, could be considered to better diversify the search [31]. This could lower the cost of increasing the fragment size, as the time spent re-optimizing fragments increases exponentially with their sizes.

The branch and bound algorithm used to optimize the fragments could also be improved with more sophisticated bounding procedures, such as approaches based on MaxSAT [18]. This would allow larger fragments to be handled by the algorithm. Additionally, multiple threads could be used to explore a single fragment in parallel. This could lead to better performances than simply exploring multiple fragments in parallel, because super-linear scaling can occur in such parallel branch and bound algorithms [27].

While we proposed a simple method to adjust some of the algorithm's parameters automatically, this method does not always give the best results. This suggests that better ways to adjust these parameters should be devised. Such a strategy could be inspired by the method used in RLS [5].

In terms of performances, the algorithm can still be improved in a number of ways in order to better make use of systems with many cores. Notably, the local search algorithm used to compute the initial solution is also single threaded. A parallel approach such as a generalization of CLS [32] to weighted graphs could be considered. Alternatively, it may be more effective to execute the local search algorithm on each fragment explored by the algorithm, instead of running it once on the entire graph.

A similar issue can be found in the method used by the algorithm to generate fragment. As described, a single thread must generate all fragments. Consequently, the performances of the algorithm are limited by how quickly a single thread can generate fragments. This could be solved by allowing multiple producer threads to run in parallel. Such a change would require a strategy to ensure that all producers generate fragments that can be explored in parallel, or at least a way of dealing with conflicts.

Bibliography

[1] Pankaj K Agarwal and Nabil H Mustafa. "Independent set of intersection graphs of convex objects in 2D". In: *Computational Geometry* 34.2 (2006), pp. 83–95.

[2] Josep Argelich. "Max-SAT formalisms with hard and soft constraints". PhD thesis. Universitat de Lleida, 2011.

[3] Luitpold Babel and Gottfried Tinhofer. "A branch and bound algorithm for the maximum clique problem". In: *Mathematical Methods of Operations Research* 34.3 (1990), pp. 207–217.

[4] Egon Balas et al. "Polyhedral methods for the maximum clique problem". In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 26 (1996), pp. 11–28.

[5] Roberto Battiti and Marco Protasi. "Reactive local search for the maximum clique problem 1". In: *Algorithmica* 29.4 (2001), pp. 610–637.

[6] Arnaud Botella, Bruno Lévy, and Guillaume Caumon. "Indirect unstructured hex-dominant mesh generation using tetrahedra recombination". In: *Computational Geosciences* 20.3 (2016), pp. 437–451.

[7] Daniel Brélaz. "New methods to color the vertices of a graph". In: *Communications of the ACM* 22.4 (1979), pp. 251–256.

[8] Manoel Campelo and Ricardo C Correa. "A Lagrangian relaxation for the maximum stable set problem". In: *arXiv preprint arXiv:0903.1407* (2009).

[9] Jens Clausen. "Branch and bound algorithms-principles and examples". In: *Department of Computer Science, University of Copenhagen* (1999), pp. 1–30.

[10] Thomas H Cormen. "Introduction to algorithms". In: MIT press, 2009. Chap. 19.

[11] Ricardo C Corrêa et al. "A bit-parallel russian dolls search for a maximum cardinality clique in a graph". In: *arXiv preprint arXiv:1407.1209* (2014).

[12] Javier Etcheberry. "The set-covering problem: A new implicit enumeration algorithm". In: *Operations research* 25.5 (1977), pp. 760–772.

[13] Zhiwen Fang et al. "Solving maximum weight clique using maximum satisfiability reasoning". In: *Proceedings of the Twenty-first European Conference on Artificial Intelligence*. IOS Press. 2014, pp. 303–308.

[14] Marshall L Fisher. "The Lagrangian relaxation method for solving integer programming problems". In: *Management science* 27.1 (1981), pp. 1–18.

[15] Charles Friden, Alain Hertz, and Dominique de Werra. "Stabulus: a technique for finding stable sets in large graphs with tabu search". In: *Computing* 42.1 (1989), pp. 35–44.

[16] Jin-Kao Hao. *ReTS Implementation*. `http://info.univ-angers.fr/pub/hao/ReTS.html`. Retrieved on March 12, 2017. 2017.

[17] John Michael Harris, Jeffry L Hirst, and Michael J Mossinghoff. "Combinatorics and graph theory". In: vol. 2. Springer, 2008. Chap. 1.

[18] Hua Jiang, Chu-Min Li, and Felip Manya. "An Exact Algorithm for the Maximum Weight Clique Problem in Large Graphs". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 2017, pp. 830–838.

[19] Richard M Karp. "Reducibility among combinatorial problems". In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[20] Kengo Katayama, Akihiro Hamamoto, and Hiroyuki Narihisa. "An effective local search for the maximum clique problem". In: *Information Processing Letters* 95.5 (2005), pp. 503–511.

[21] Janez Konc and Dušanka Janezic. “An improved branch and bound algorithm for the maximum clique problem”. In: *proteins* 4.5 (2007).

[22] Deniss Kumlander. “A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search”. In: *Proc. 5th Int'l Conf. on Modelling, Computation and Optimization in Information Systems and Management Sciences*. Citeseer. 2004, pp. 202–208.

[23] Jan Karel Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

[24] Chu Min Li and Zhe Quan. “An Efficient Branch-and-Bound Algorithm Based on MaxSAT for the Maximum Clique Problem.” In: *AAAI*. Vol. 10. 2010, pp. 128–133.

[25] Pedro Martins. “Extended and discretized formulations for the maximum clique problem”. In: *Computers & Operations Research* 37.7 (2010), pp. 1348–1358.

[26] Ciaran McCreesh and Patrick Prosser. “Multi-threading a state-of-the-art maximum clique algorithm”. In: *Algorithms* 6.4 (2013), pp. 618–635.

[27] Ciaran McCreesh and Patrick Prosser. “The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound”. In: *ACM Transactions on Parallel Computing* 2.1 (2015), p. 8.

[28] Sia Meshkat and Dafna Talmor. “Generating a mixed mesh of hexahedra, pentahedra and tetrahedra from an underlying tetrahedral mesh”. In: *International Journal for Numerical Methods in Engineering* 49.1-2 (2000), pp. 17–30.

[29] Panos M Pardalos and Jue Xue. “The maximum clique problem”. In: *Journal of global Optimization* 4.3 (1994), pp. 301–328.

[30] J. Pellerin, A. Johnen, and J.-F. Remacle. “Combining tetrahedra into hexahedra: a vertex based strategy”. In: *ArXiv e-prints* (May 2017). arXiv: `1705.02451`.

[31] David Pisinger and Stefan Ropke. “Large neighborhood search”. In: *Handbook of metaheuristics*. Springer, 2010, pp. 399–419.

[32] Wayne Pullan, Franco Mascia, and Mauro Brunato. “Cooperating local search for the maximum clique problem”. In: *Journal of Heuristics* 17.2 (2011), pp. 181–199.

[33] Pablo San Segundo, Diego Rodriguez-Losada, and Agustin Jiménez. “An exact bit-parallel algorithm for the maximum clique problem”. In: *Computers & Operations Research* 38.2 (2011), pp. 571–581.

[34] Pablo San Segundo and Cristobal Tapia. “Relaxed approximate coloring in exact maximum clique search”. In: *Computers & Operations Research* 44 (2014), pp. 185–192.

[35] Paul Shaw. “Using constraint programming and local search methods to solve vehicle routing problems”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 1998, pp. 417–431.

[36] Dmitry Sokolov et al. “Hexahedral-dominant meshing”. In: *ACM Transactions on Graphics (TOG)* 35.5 (2016), p. 157.

[37] Etsuji Tomita and Toshikatsu Kameda. “An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments”. In: *Journal of Global optimization* 37.1 (2007), pp. 95–111.

[38] Etsuji Tomita and Tomokazu Seki. “An efficient branch-and-bound algorithm for finding a maximum clique”. In: *Discrete mathematics and theoretical computer science*. Springer, 2003, pp. 278–289.

[39] Etsuji Tomita et al. “A Much Faster Branch-and-Bound Algorithm for Finding a Maximum Clique”. In: *International Workshop on Frontiers in Algorithmics*. Springer. 2016, pp. 215–226.

[40] Etsuji Tomita et al. “A simple and faster branch-and-bound algorithm for finding a maximum clique”. In: *International Workshop on Algorithms and Computation*. Springer. 2010, pp. 191–203.

[41] Yiyuan Wang, Shaowei Cai, and Minghao Yin. “Two Efficient Local Search Algorithms for Maximum Weight Clique Problem.” In: *AAAI*. 2016, pp. 805–811.

[42] Qinghua Wu and Jin-Kao Hao. “A review on algorithms for maximum clique problems”. In: *European Journal of Operational Research* 242.3 (2015), pp. 693–709.

[43] Qinghua Wu and Jin-Kao Hao. “An adaptive multistart tabu search approach to solve the maximum clique problem”. In: *Journal of Combinatorial Optimization* 26.1 (2013), pp. 86–108.

[44] Qinghua Wu, Jin-Kao Hao, and Fred Glover. “Multi-neighborhood tabu search for the maximum weight clique problem”. In: *Annals of Operations Research* 196.1 (2012), pp. 611–634.

[45] Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. “Scalable maximum clique computation using mapreduce”. In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE. 2013, pp. 74–85.

[46] Soji Yamakawa and Kenji Shimada. “Fully-automated hex-dominant mesh generation with directionality control via packing rectangular solid cells”. In: *International journal for numerical methods in engineering* 57.15 (2003), pp. 2099–2129.

[47] Yi Zhou, Jin-Kao Hao, and Adrien Goëffon. “PUSH: A generalized operator for the Maximum Vertex Weight Clique Problem”. In: *European Journal of Operational Research* 257.1 (2017), pp. 41–54.

Appendix A

Quantitative Information about the Implementation

Our implementation is written in C++ and contains 42 classes, spread across 33 files with a total of 6208 lines of code. This code is found in the following directories of the repository:

1. `TetRecombination/tests/`
2. `TetRecombination/src/mwis/`
3. `TetRecombination/src/search/`
4. In `TetRecombination/src/`, the files `mwis.h`, `mwis.cpp` and `search.h`.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**