# A constructive parallel heuristic approach for the MWPSP

**Rafael C. Silva, Vinicius Coelho,**
**Wellington Martins, Humberto Longo, Leslie Foulds**

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Avenida Esperança s/n, Campus Samambaia – 74690-900 Goiânia – GO – Brazil

rafael_castro@discente.ufg.br, vinicoelhose@gmail.com,
{wsmartins,longo,lesfoulds}@ufg.br

***Abstract.*** *Algorithms for the problem of identifying a maximum edge-weight planar subgraph of a given edge-weighted graph $G$ are relevant in a wide variety of application areas. In this paper, we propose a new local search constructive heuristic for this $\mathcal{NP}$-hard problem, along with a GPU-based algorithm designed to accelerate its computational process. The experimental findings demonstrated that considerable speedup is attainable while maintaining good-quality results.*

## 1. Introduction

The analysis of various systems of discrete objects is often facilitated by depicting the system as a network where nodes generally represent the objects (such as stocks, facilities, electrical junctions, proteins or social groups) and the arcs represent the relationships between them. It is often useful to visualise the data by drawing the network in the plane (without arcs crossing). For all but trivial instances this requires that only a strict subset of the arcs can be included. The arcs are usually nonnegatively weighted (e.g., being correlations between the objects). Hence, it is commonly assumed that the best balance between the retention of relationship information and the clarity of the network drawing involves identifying the planar subnetwork of maximum total arc weight of a given arc-weighted graph. Planarity ensures easy visualisation of the object relationships by enabling the network to be drawn without arcs crossing. Henceforth, we use graph theoretic notation and terminology to analyse the above-mentioned problem. A formal description of the problem in terms of graph theory is given next.

This work focuses on complete, undirected, simple graphs $G = (V, E)$, with finite vertex set $V$ and edge set $E$, where $|V| = n$ and $|E| = m$. $G$ is termed planar if it can be drawn in the plane, such that no two of its edges intersect geometrically, except at a vertex with which they are both incident; and *maximally planar* if it is planar and no further edge can be added to it without rendering it no longer planar (*nonplanar*). Furthermore, it is assumed that $G$ is nonnegatively edge-weighted in the sense that there exists a function $w : E \to \mathbb{R}^+$.

Given a complete, edge-weighted graph $G$, the *Maximum-Weight Planar Subgraph Problem* (MWPSP) involves searching for a planar subgraph $G' = (V, E')$, with the highest sum of edges weights $\sum_{e \in E'} w(e)$. Since the edge weights are all nonnegative, there will always be a solution $(V, E')$ that is maximally planar. The special case where the edge weights are all equal, i.e. $w(e) = c,\ c \in \mathbb{R}^+,\ \forall\ e \in E$, degenerates to finding a planar subgraph with $3n - 6$ edges and can be solved in $\mathcal{O}(n)$ time [Jünger and Mutzel 1993].

The MWPSP has important applications in a variety of fields, including: $(i)$ financial analysis, where the vertices and edges represent stocks and the correlations between them, respectively [Tumminello et al. 2005, Massara et al. 2017]; $(ii)$ as a subproblem of the plant (facility) layout problem, where the vertices and edges represent the activities of the facility and the adjacencies between some of them, respectively [Seppänen and Moore 1970, Pesch 1999, Ahmadi-Javid et al. 2015]; $(iii)$ integrated circuit design, where vertices and edges represent electrical elements and the physical connections between them [Lengauer 2012]; $(iv)$ systems biology, where vertices and edges represent proteins and the interactions between them in a metabolic network [Song et al. 2007]; and $(v)$ social systems, where the vertices and edges represent social agents (e.g. individuals, groups or companies) and social interaction [Easley and Kleinberg 2010]. The nature of applications $(ii)$ and $(iii)$ dictates that any feasible solution to the problem must be a planar subgraph. However subgraphs regarded as solutions to $(i)$, $(iv)$ and $(v)$ need not necessarily be planar – in these cases the requirement of planarity is introduced as a soft constraint, added only to promote visualisation.

This article addresses the following research question: How to improve approximate MWPSP solutions, allowing larger instances to be efficiently resolved? We propose a new heuristic method, termed here *Restricted Seeds*. In this method, we construct the solution not only by applying face dimpling but also edge dimpling as well, as explained in Section 3. The remainder of the method proceeds in the same way as a previously reported heuristic method known as *face dimpling ($\mathcal{FD}$)*, but selects only a relatively small subset of the $K_4$s of the input graph, instead of working with the entire possible set of such subgraphs. Combining these changes makes the method not only faster, but also improves solution quality compared to the $\mathcal{FD}$ heuristic. A GPU parallel implementation of the proposed method is described and computational results obtained with numerical instances (including some of relatively large-scale), both from the literature and synthetically constructed are reported. Another algorithm, called here *All Seeds*, is an extended version of the *Restricted Seeds* method but tests all possible *seeds* instead.

The remaining of the text is organised as follows. The next section briefly presents the related work on algorithms for the MWPSP and Section 3 outlines some existing construction procedures that build up a planar subgraph step by step. These procedures are utilised to develop new, faster, parallel heuristic algorithm in Section 5. Section 4 defines some parallel algorithms concepts used here and briefly talks about GPU and CUDA. Computational experience is reported in Section 6 and a summary and some conclusions round out the paper in the last section.

## 2. Related Work

Numerous exact and heuristic algorithms for the MWPSP have been reported over the last 50 years. The heuristic algorithms are commonly based on vertex-by-vertex construction, local search improvement and learning metaheuristics. The improvement algorithms involve local topological moves, including edge substitution and the relocation of vertices from one face of a planar subgraph to another. As the MWPSP is known to be strongly $\mathcal{NP}$-hard [Giffin 1984], it is no surprise that even the best current exact MWPSP algorithms are able to resolve instances with only a relatively small number of vertices.

[Massara et al. 2017] recently proposed a heuristic MWPSP method that uses var-

ious construction moves, called the TMFG algorithm. The method starts from a $K_4$ (a complete graph on four vertices), termed here as *seed*, and adds vertices one at a time by using the face dimpling move (see Figure 1). They note that the TMFG algorithm can be extended to include the edge dimpling move (see Figure 1) and improvement moves as well. At each step, the actual move implemented is chosen in order to maximise a score function, most naturally, the sum of the weights of the edges to be added. When confined to face dimpling, the TMFG method is more flexible than the well known deltahedron method of [Foulds and Robinson 1978], as it does not involve a pre-ordering of the vertices. Like the deltahedron method, at each iteration, TMFG computes the increase in score that would be achieved by inserting any of the remaining vertices in any currently existing face. The dimpling move is applied to the vertex-face pair that induces the maximum increase in score. At first glance, this requires the calculation of the increase in score for every feasible vertex-face pair, requiring $\mathcal{O}(n^2)$ calculations at each iteration, leading to an overall complexity of $\mathcal{O}(n^3)$. Importantly, this computational burden has been significantly reduced by maintaining a cache that records only the best possible vertex-face pair and updating only the records affected by the corresponding face dimpling move. The maintenance of the cache requires $\mathcal{O}(n)$ running time and the complexity of the TFMG algorithm reduces to only $\mathcal{O}(n^2)$.

During this Millennium breakthroughs in information technology have had significant effects on discrete optimisation. In particular, the rapid growth and widespread availability of parallel processors [Kirk and Wen-Mei 2016] has meant that its use for various large-scale, difficult discrete optimisation problem instances in operational research is a growing trend [Migdalas et al. 2013]. While parallel processing is not expected to significantly reduce the worst-case run time of large-scale numerical instances of combinatorial optimisation problems that are strongly $\mathcal{NP}$-hard (which would require an exponential number of processors), optimal solutions to moderately-sized instances can be obtained via parallel processing in a reasonable amount of time. In addition, the execution time of various heuristic algorithms for many hard combinatorial problems is polynomially bounded and, in these situations, parallel processing can sometimes significantly increase the size of solvable instances, while keeping the time relatively low.

Thus, to reduce the processing time of well-established sequential methods on reasonably large MWPSP instances, modern parallel architectures that use multiple cores might be considered. Today, virtually all CPUs (Central Processing Units) support parallelism through the use of multiple cores. In a similar way, many-core architectures such as GPUs (graphical cards used for general purpose computing) have dramatically increased the number of available cores – sometimes in the thousands. Many-core processors, also known as accelerators, work with relatively simple and slower cores and are becoming increasingly affordable due to mass marketing in the gaming industry. They are designed for massive, multi-threaded systems and require a significant number of threads. This imposes some constraints in developing appropriate algorithms, requiring the design of novel solutions and new implementation approaches.

[Massara et al. 2017] suggested the use of parallel and GPU computing in the MWPSP resolution approaches. However, apparently, these authors have not followed through. [Coelho et al. 2016] proposed a successful method, termed here as $\mathcal{FD}$ heuristic, which starts from each of the possible *seed* of a graph, using fine-grained parallelism in

GPU's, builds solutions from the successive application of face dimpling, described in Section 3, and returns the best result found. But, as Coelho et al. points out, testing all possible *seed* is a bottleneck in their method, as his computational results showed that only a few seeds led to good results and most of them lead to unsatisfactory results. In fact, only one seed matters in the whole process: the one that produces the best solution!

## 3. The vertex-by-vertex construction approach to the MWPSP

As mentioned earlier, many of the existing sequential MWPSP heuristics are based on construction procedures. We discuss some of these strategies and their results, with a view to developing parallelised MWPSP heuristics. A given maximal planar graph can be transformed into a new one by various step-by-step local topological moves. The choice of the actual move at each step is often made greedily on the basis of the highest increase in total subgraph weight. We now describe some commonly-used moves for sequentially constructing a solution to the MWPSP. These moves are shown on the complete vertex-labelled graph with four vertices $K_4$, termed here a *tetrahedron*.

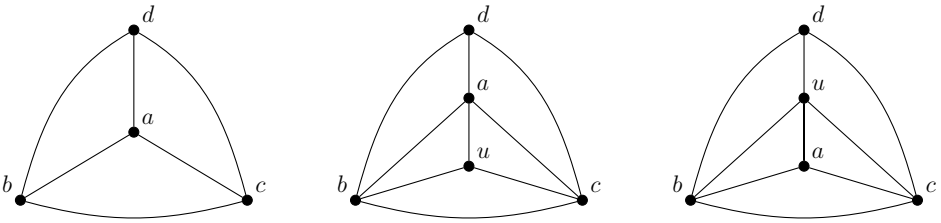
To illustrate the moves, suppose $G = (V, E)$ is a maximally planar graph with a set of faces $F$. In the first move, at each iteration, a vertex is inserted into a triangular face of the current graph and three edges are added by connecting the newly inserted vertex to the vertices of the triangular face. This procedure is termed here as *face dimpling* (denoted as $T_2$ by [Massara et al. 2017]), as illustrated in Figures 1a and 1b. Suppose $n \geqslant 3$, $\{a, b, c\} \in F$ and there is a potential vertex $u \notin V$. The face dimpling move inserts $u$ into the face defined by vertices $a, b$ and $c$ to transform $G$ into the maximally planar graph $G' = (V', E')$ where $V' = V \cup \{u\}$, $E' = E \cup \{\{a, u\}, \{b, u\}, \{c, u\}\}$, $F' = (F \setminus \{\{a, b, c\}\}) \cup \{\{a, b, u\}, \{b, c, u\}, \{a, c, u\}\}$. Clearly, the move preserves planarity at each iteration.

A second construction move is called here *edge dimpling* (termed the "$A$ move" by [Massara et al. 2017] in honour of its inventor, [Alexander 1930]). The edge dimpling operation removes an existing edge and inserts a new vertex and four edges, as shown in Figures 1a and 1c. Suppose $n \geqslant 4$, $\{a, d\} \in E$ and there is a potential vertex $u \notin V$. In this move, the edge $\{a, d\}$ is replaced by the vertex $u$, which is connected to vertices $a, b, c$ and $d$ to transform $G$ into the maximal planar graph $G' = (V', E')$ where $V' = V \cup \{u\}$, $E' = (E \backslash \{a, d\}) \cup \{\{a, u\}, \{b, u\}, \{c, u\}\}, \{d, u\}\}$, and $F' = F \cup \{\{a, b, u\}, \{a, c, u\}, \{b, d, u\}, \{c, d, u\}\} \backslash \{\{a, b, d\}, \{a, c, d\}\}$. Once again, the move preserves planarity at each iteration. Unlike face dimpling, where each insertion produces a vertex of degree 3, edge dimpling produces a vertex of degree 4.

A third general construction move, termed here *face-edge dimpling*, chooses between a face or an edge dimpling move at each construction step of a maximal planar subgraph. In MWPSP, this choice is made greedily, based on the largest overall increase in the total edge weight of the current subgraph.

## 4. Parallelism and GPU

The following sections presents an overview of the GPU (graphical processing unit) architecture, which serves as the primary hardware for parallelism in this work. A fundamental concept in parallel computing, the *speedup* measures the performance improvement achieved by executing a parallel algorithm compared to its sequential counterpart.




**(a) The original graph $G$ (before any topological move).** **(b) Face dimpling move (vertex $u$ inserted into face $\{a, b, c\}$).** **(c) Edge dimpling move (vertex $u$ inserted within edge $\{a, d\}$).**

**Figure 1. The face and the edge dimpling move.**

It quantifies the reduction in execution time when utilising multiple processing units concurrently. The *speedup* of a parallel algorithm can be calculated by dividing the execution time of the sequential algorithm by the execution time of the parallel algorithm. Mathematically, the *speedup* ($S$) is defined as: $S = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}$; where $T_{\text{sequential}}$ represents the execution time of the sequential algorithm, and $T_{\text{parallel}}$ represents the execution time of the parallel algorithm. A *speedup* value greater than 1 indicates that the parallel algorithm is faster than the sequential algorithm. For example, if the *speedup* is 2, it means that the parallel algorithm executes in half the time of the sequential algorithm. The larger the *speedup*, the more efficient the parallelization.

### 4.1. GPU architecture and CUDA

Graphics Processing Units (*GPUs*) initially became popular with their use for graphic rendering and image processing, especially in gaming applications. However, their parallel processing structure makes them a great alternative for efficient processing of large data blocks. Computers contain a Central Processing Unit (*CPU*), where most of the processing is done. A typical CPU is designed to perform serial tasks quickly and with low latency. Additionally, CPUs are designed to easily switch between tasks. GPUs, on the other hand, are designed to optimize the number of tasks executed per unit of time (*throughput*) by processing tasks in parallel. Thus, a processing core (*core*) of a GPU usually has a higher latency than a CPU core, but the GPU focuses on processing more data tasks through a much larger number of cores performing in parallel.

In a GPU there are multiple *streaming multi-processors* (SM) and each one contain multiple *single processors* (SP). During a clock cycle, all the SPs within an SM execute the same instruction, but with different data. Thus, each SM is considered a SIMD (*single instruction multiple data*) processor, and if different SPs within an SM need to perform different instructions, these instructions are executed serially. A GPU also contains a global memory, with access available to all SPs, and a high-speed shared memory for the SPs of each SM.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It allows developers to harness the power of GPUs for general-purpose computation. In CUDA, computations are divided into threads, which are executed on the GPU. A *thread* is the basic unit of execution in CUDA. It represents an independent sequence of instructions that can be executed in parallel. Threads are organized into thread blocks, and multiple thread blocks can be executed concurrently. Each thread is identified by a unique thread index within a block. A

*block* is a group of threads that can cooperate with each other by sharing data and synchronizing execution. Threads within a block can communicate using shared memory. A block is executed on a single streaming multiprocessor (SM) of the GPU. The number of threads in a block is limited by the hardware resources of the SM, such as the number of available registers and shared memory.

In CUDA programming, a kernel refers to a function that runs on the GPU and is executed in parallel by multiple threads. When calling a CUDA kernel, we specify the launch configuration, which includes the number of thread blocks and threads per block. A CUDA kernel is called as follows: `kernelFunction<<<numberOfBlocks, numberOfThreads>>>(args);` where `kernelFunction` is the name of the CUDA kernel function and `<<numberOfBlocks, numberOfThreads, sizeOfSharedMemory>>` denotes the launch configuration and `args` are the arguments passed to the kernel function.

The `numberOfBlocks` parameter represents the number of blocks, and the `numberOfThreads` parameter specifies the number of threads per block. The actual number of threads executed in parallel will be the product of `numberOfBlocks` and `numberOfThreads`. The `sizeOfSharedMemory` parameter specifies the size of shared memory (in bytes) to be allocated per block. This allows developers to explicitly manage and optimise the use of shared memory, which can lead to significant performance improvements in certain GPU algorithms.

## 5. Proposed parallelized restricted-seeds heuristic

The $\mathcal{FD}$ heuristic, proposed by [Coelho et al. 2016], uses the face dimpling move, described in Section 3, to greedily build a solution to an MWPSP instance, starting from every possible *seed* (all tetrahedra of $G$). Since the number of seeds is $\mathcal{O}(n^4)$ and each is used to construct a solution in $\mathcal{O}(n^3)$ topological moves, the overall time complexity of the $\mathcal{FD}$ heuristic is $\mathcal{O}(n^7/p)$, where $p$ is the total number of threads available. However, it can be assumed that only a subset of the seeds actually leads to high quality solutions. Thus, the new *Restricted Seeds* heuristic works similarly to the $\mathcal{FD}$ heuristic, but it does not process the entire set of seeds, only a strict subset of them, and also uses the edge dimpling subroutines, instead of only face dimpling.

The choice of this subset is the key to a good performance of the *Restricted Seeds*, regarding processing time and, more importantly, the quality of returned solutions. Therefore, an efficient way to obtain subsets with the smallest possible number of seeds that can potentially lead to high quality solutions is proposed. An intuitive way of doing this is to consider the weight of each seed considered (the sum of its edge weights), since it is common sense that seeds with relatively high weights are usually likely to lead to good quality solutions.

The proposed method comprises a pre-processing phase concerning the complete set of all $\binom{n}{4}$ possible seeds that can be extracted from $G$. This phase sorts all these seeds in non-increasing order according to their weight, with a time complexity $\mathcal{O}(n^4 \cdot \log(n))$. Since this sorting phase must be performed only once before the runs of the restricted seeds heuristic, it does not affect the complexity of the main part of the method, which has time complexity $\mathcal{O}((y+z)n^3)$, given the parameters $y$ and $z$ described below. However, if $y + z = \mathcal{O}(1)$, then the final complexity of the proposed heuristic is $\mathcal{O}(n^3)$, without a

significant loss in the final quality of the best solution, if we test all seeds, as discussed in Section 6.1.

Let $C$ denote the subset of filtered (finally chosen) seeds. Only the first heaviest $x$ seeds in the ordered set of seeds are considered for the filtering process. Initially, the first $y$ (where $y \leqslant z$) heaviest seeds are added to $C$. Next, $z$ more seeds ($y + z \leqslant x$) are randomly chosen among the $(x - y)$ remaining seeds and are added to $C$. Then, an algorithm, called here as *Face-Edge Dimpling* (*FED*), is applied to each seed in $C$. The reason for randomly choosing some additional seeds is to improve the chances of escaping from an eventual local optima that could be reached if only the heaviest $y$ seeds were used. The approach used to apply parallelism in this new method was not different from the one used in the $\mathcal{FD}$, the major difference being that in order to sort the seeds in the set $C$, it used a parallel sorting algorithm as well.

The pseudocode of the *Restricted Seeds* is presented in two parts. The first part (Algorithm 1) details the processing done at the CPU level, and how the kernel was called. First, at step 1, we select the set of *seed* that will be used, in a process described before, and store it at $\mathcal{C}$. After it, we move both the graph $G$ and the seeds $\mathcal{C}$ to the GPU (steps 2 and 3), as it will be used in the construction of the solution. The steps 4 - 6, defines the values that will be used in the kernel configuration. The step 7 call the kernel function, that will be detailed in 2. This kernel will be responsible to construct the solution. Finally, the maximal subgraph obtained will be returned at step 8.

Algorithm 2 details the kernel itself, wich run for each thread. In the first step 1, we move the input graph $G$ to the shared memory of the GPU. In the step 2 we construct a unique index $k$ for the current thread, where $0 \leq k \leq totalNumberOfThreads = numberOfBlocks \cdot threadsPerBlock$. The value of $totalNumberOfThreads$ can be obtained using CUDA built-in variables. The construction of the solution for each *seed* (*seed* growth) is done between the steps 3 and 14. In these steps is successively applied between the face dimpling or edge dimpling movement, accordingly with the greatest local improvement. Variables $\mathcal{C}_k$ and $\mathcal{S}$, defines, respectively, the initial seeds and the vertices that were not inserted in the solution yet. Sets $\mathcal{E}$ and $\mathcal{F}$ defines the edges and faces of the current solution, respectively. Steps 8 to 15 show the process of applying the face dimpling and edge dimpling routines. Since the quantity of edges and faces in a maximal planar graph with $n$ vertices are $3n - 6$ and $2n - 4$ respectively, and at each construction step we look at every face vertex and edge vertex combination, then the complexity of a *seed* growth is $\sum_{i=4}^{n} i \cdot (2i - 4 + 3i - 6) \in \mathcal{O}(n^3)$. The edges of every final solution, for each seed, are stored in $\mathcal{R}$ (step 16). This step is done in $\mathcal{O}(n)$. Every other step in this method is done in $\mathcal{O}(1)$. The index $k$ is increased by the $totalNumberOfThreads$ in step 17, which guarantees that each thread processes a disjoint subset of seeds, which together form the set $\mathcal{C}$. Finally, in the step 18 we get the edges that form the solution with greatest weight, and return it together with the initial set of vertices forming a maximal planar subgraph, hopefully with the greatest weight as possible, in the step 19. Since both the face and the edge dimpling seed growth are done in $\mathcal{O}(n^3)$ and $y + z$ seeds are tested, the overall complexity is $\mathcal{O}((y + z)n^3)$.

Some pseudocode details have been consciously omitted. For example, the way the results of different threads are combined is not detailed, as it was considered that this does not affect the understanding of the described method and because it is a trivial task

used in many other parallel algorithms. Thus, the focus is on the most important steps of the proposed method itself.

**Algorithm 1:** RestrictedSeeds

**Input:** Graph $G = (V, E)$ and parameters $x, y, z \in \mathbb{N}$ that reduce the search space of viable solutions
**Output:** Maximal planar subgraph $G'$, filtered from $G$.

1 $\mathcal{C} \leftarrow FilteredSeeds(G, x, y, z)$;
2 Store $G$ in GPU;
3 Store $\mathcal{C}$ in GPU;
4 $threadsPerBlock \leftarrow 2^9$;
5 $numberOfBlocks \leftarrow \lceil \frac{|C|}{threadsPerBlock} \rceil$
6 $sharedMemorySize \leftarrow |G|$;
7 $G' \leftarrow$ FaceEdgeDimpling
$<<< numberOfBlocks, threadsPerBlock, sharedMemorySize >>>(G, \mathcal{C})$;
8 **return** $G'$.

**Algorithm 2:** FaceEdgeDimpling( $G, C$ ) // kernel function

**Input:** Graph $G = (V, E)$, $C$ as a set of seeds
**Output:** Maximal planar subgraph $G'$, filtered from $G$.

1 moveToSharedMemory(G)
2 $k \leftarrow$ current thread index
3 **while** $k \leq |\mathcal{C}|$ **do**
4 $\quad \mathcal{C}_k \leftarrow \{v_1^k, v_2^k, v_3^k, v_4^k\} \in \mathcal{C}$
5 $\quad \mathcal{S} \leftarrow V(G) - \{\mathcal{C}_k\}$
6 $\quad \mathcal{E} \leftarrow \{\{v_1^k, v_2^k\}, \{v_1^k, v_3^k\}, \{v_1^k, v_4^k\}, \{v_2^k, v_3^k\}, \{v_2^k, v_4^k\}, \{v_3^k, v_4^k\}\}$
7 $\quad \mathcal{F} \leftarrow \{\{v_1^k, v_2^k, v_3^k\}, \{v_1^k, v_2^k, v_4^k\}, \{v_1^k, v_3^k, v_4^k\}, \{v_2^k, v_3^k, v_4^k\}\}$
8 $\quad$ **while** $(\mathcal{S} \neq \varnothing)$ **do**
9 $\quad\quad face\_move \leftarrow \{\text{max gain with } \{s, f\},\ \forall\, f \in \mathcal{F} \text{ e } s \in \mathcal{S}\}$
10 $\quad\quad edge\_move \leftarrow \{\text{max gain with } \{s, e\},\ \forall\, e \in \mathcal{E} \text{ e } s \in \mathcal{S}\}$
11 $\quad\quad$ **if** $(face\_move > edge\_move)$ **then**
12 $\quad\quad\quad$ Call *Face Dimpling* routine;
13 $\quad\quad$ **else**
14 $\quad\quad\quad$ Call *Edge Dimpling* routine;
15 $\quad\quad$ Update the sets $\mathcal{F}$, $\mathcal{E}$ and $\mathcal{S}$;
16 $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{E}$;
17 $\quad k \leftarrow k + totalNumberOfThreads$;
18 $\mathcal{M}' \leftarrow \max(\mathcal{R})$;
19 **return** $G' = (V, \mathcal{M}')$.

## 6. Some computational results

The tests with the sequential version of *Restricted Seeds* (Section 6.1) were conducted on a personal computer with 16GB of RAM, 11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz, 8x2 cores, processor, and Ubuntu 20.04 operating system. It was was used the best possible sequential implementation, in terms of asymptotic complexity, of this algorithm. We achieved $\mathcal{O}((x + y)n^2 log(n))$, as we took advantage of sparse structures that don't work very well on GPUs, with the steps 8–15 spending $\mathcal{O}(n^2 log(n))$. The parallel versions of *Restricted Seeds*, *All Seeds* (applying *FED* using all the seeds) and $\mathcal{FD}$ were tested using

an Intel(R) Xeon(R) CPU @ 2.30GHz, 2x2 cores, model 63 and a Tesla T4 GPU with 2560 CUDA Cores, both devices with 16GB of main memory. The results presented are the average outcomes of 10 independent runs, taking into account both processing and transfer times (GPU only).

The experiments used synthetically generated instances ($K_n$, $n = 10, 25, \ldots, 100$), originally proposed by Coelho et al., [Coelho et al. 2016], termed here as *Coelho's Instances* with weights $w(e), e \in E(K_n)$ randomly generated in the range $[0, 199]$[1]. These instances were generated in a way that was specially hard to solve in an optimal manner, or close to it. We also used the 100-vertex *Tumminello Instance* [Tumminello et al. 2005], which is based on a real empirical study of practical financial data derived from the US stock market, with vertices and edges representing stocks and given correlation coefficients between them, respectively.

### 6.1. Parallel Restricted Seeds results

Table 1 compares the results obtained with those of the heuristics $\mathcal{FD}$ [Coelho et al. 2016], *Restricted Seeds*, *All Seeds* and *TMFG* [Massara et al. 2017] using *Coelho's Instances*. The parameters $y$ and $z$, described in 5, were both fixed at 95, so $O(y + z)$ is $O(1)$.

**Table 1. Results from the *Restricted Seeds* and the *All Seeds* heuristics.**

| Instance | | All Seeds | | Face Dimpling | | TMFG | Restricted Seeds | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | $\sigma$ | GPU ($s$) | Value | GPU ($s$) | gap (%) | gap (%) | Seq. ($s$) | GPU ($s$) | Value ($\mu$) | gap range (%) | speedup |
| 10 | 50.14 | 0.004 | 2107 | 0.368 | 0.00 | 4.74 | 0.031 | 0.004 | 2107.0 | [0,00, 0.00] | 7.45 |
| 15 | 56.17 | 0.022 | 5772 | 0.362 | 0.50 | 0.97 | 0.064 | 0.012 | 5772.0 | [0.00, 0.00] | 4.98 |
| 20 | 55.30 | 0.053 | 8321 | 0.365 | 3.12 | 6.89 | 0.165 | 0.028 | 8266.6 | [0.42, 0.89] | 5.80 |
| 25 | 58.70 | 0.131 | 11431 | 0.370 | 1.22 | 4.13 | 0.287 | 0.047 | 11364.9 | [0.46, 0.70] | 6.05 |
| 30 | 55.93 | 0.352 | 13644 | 0.375 | 1.53 | 4.85 | 0.454 | 0.077 | 13470.5 | [1.14, 1.40] | 5.87 |
| 35 | 58.93 | 0.667 | 17028 | 0.412 | 1.15 | 2.89 | 0.679 | 0.117 | 16793.8 | [1.33, 1.42] | 5.76 |
| 40 | 56.44 | 1.460 | 19454 | 0.489 | 1.14 | 3.54 | 1.047 | 0.150 | 19349.2 | [0.45, 0.63] | 6.94 |
| 45 | 56.57 | 2.993 | 21669 | 0.663 | 0.84 | 2.15 | 1.572 | 0.209 | 21466.2 | [0.89, 0.98] | 7.50 |
| 50 | 57.30 | 6.706 | 25126 | 0.995 | 1.11 | 1.89 | 2.448 | 0.306 | 24919.3 | [0.77, 0.88] | 7.99 |
| 55 | 58.20 | 11.648 | 27771 | 1.698 | 1.12 | 3.02 | 3.401 | 0.324 | 27621.2 | [0.50, 0.58] | 10.48 |
| 60 | 57.04 | 22.613 | 30432 | 2.801 | 1.14 | 3.65 | 4.699 | 0.356 | 30069.6 | [1.01, 1.37] | 13.19 |
| 65 | 56.39 | 37.106 | 33119 | 4.309 | 0.69 | 2.12 | 6.361 | 0.416 | 32884.5 | [0.59, 0.83] | 15.28 |
| 70 | 58.26 | 62.602 | 36410 | 7.844 | 0.84 | 2.74 | 8.790 | 0.428 | 36071.5 | [0.86, 1.00] | 20.52 |
| 75 | 57.50 | 100.736 | 38953 | 12.733 | 0.91 | 2.41 | 11.585 | 0.510 | 38656.9 | [0.72, 0.80] | 22.68 |
| 80 | 57.39 | 161.685 | 41771 | 20.062 | 0.66 | 2.23 | 15.015 | 0.587 | 41433.1 | [0.79, 0.82] | 25.53 |
| 85 | 57.47 | 241.985 | 44268 | 30.921 | 0.73 | 2.53 | 21.052 | 0.643 | 44045.6 | [0.45, 0.56] | 32.73 |
| 90 | 57.37 | 375.967 | 47301 | 47.297 | 0.68 | 3.11 | 26.935 | 0.837 | 46933.0 | [0.78, 0.78] | 32.15 |
| 95 | 57.48 | 603.318 | 49961 | 69.420 | 0.58 | 1.80 | 37.986 | 0.865 | 49587.0 | [0.67, 0.82] | 43.89 |
| 100 | 58.57 | 866.924 | 53377 | 100.670 | 0.64 | 2.65 | 46.129 | 1.025 | 52922.5 | [0.79, 0.92] | 45.00 |
| **Average gaps/speedup (%)** | | | | | **0.98** | **2.92** | | | | **[0.66, 0.81]** | **16.83** |

Table 1 has five column blocks. The first block concerns the characteristics of the instances and the following four blocks shows the results achieved by the *All Seeds*, *Face Dimpling*, *TMFG* and *Restricted Seeds* heuristics. Columns labelled $\boldsymbol{n}$ and $\boldsymbol{\Sigma}$ contains the number of vertices and the standard deviation of the edges weights of each instance. Columns labelled **GPU** contain the processing times (in seconds) in the parallelised versions of the heuristics, **Seq. (s)** column shows the running times (in seconds) of the *Restricted Seeds* method in the sequential version and columns labelled **Value** contain the

[1] https://github.com/viniscoelho/dimpling/tree/master/inputs.

weight of the planar graph obtained (in the RS heuristic it is the average value obtained in ten runs of the method). The **gap (%)** columns shows the percentage difference between the value obtained by the respective heuristic and the *All Seeds* method (a gap of 0.00% means that the solution is as good as the *All Seeds*). Finally, the last column contains the speedup of the parallel version of the *Restricted Seeds* heuristic compared to its sequential version. The average value presented for the *Restricted Seeds* heuristics is due a random step in the seed filtering method, as explained in Section 5, and the **gap range (%)** is an interval with a confidence level of 95% for the gaps obtained in ten runs of the method.

The variation in the standard deviations of the edge weights between the used instances is very low (less than 3.3 from n=15 to n=100). Although it is not a general rule, with more homogeneous weight distributions (smaller standard deviations), as in $n \in \{20, 25, 30\}$, the *Face Dimpling* and TMFG heuristics tend to obtain results with larger gaps. On the other hand, this feature seems to have less impact on the behaviour of the *Restricted Seeds* heuristic.

To better evaluate the relationship between parameter configuration and expected values, tests were performed with different instances and parameter configurations. The box plot graph in Figure 2 presents the results obtained using the *Tumminello Instance*. In this experiment, parameters $x$, $y$, and $z$ were configured using variables $X'$, $Y'$ and $Z'$, respectively, where $X'$ is a percentage of the total number of *seeds*. For example, if $T$ is the total number of different *seeds* for a graph $G$, setting $X' = 10$ implies $x = 10\%T$. Similarly, $Y'$ and $Z'$ configure the parameters $y$ and $z$ as percentages of the value $x$. For instance, if $Y' = 5$ and $Z' = 10$, then $y = 5\%x$ and $z = 10\%x$.

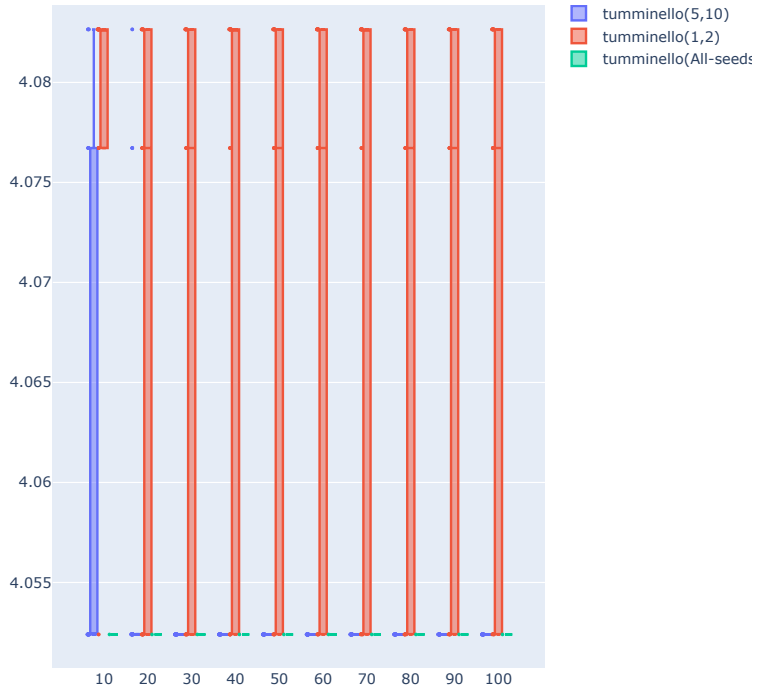



**Figure 2. Solution over different configurations for the Tumminello instance.**

The graph displays different configurations: the blue boxes represent configurations where $Y' = 5$ and $Z' = 10$, red boxes represent $Y' = 1$ and $Z' = 2$, and green boxes

correspond to a configuration that tests all the seeds. This last case is essentially the *All Seeds* method. The parameter $X'$ varies along the horizontal axis, starting at 10 and increasing by 10 up to 100. The vertical axis represents the gap between the solutions and the known upper bound for the *Tumminello Instance*. In contrast to the previous table, the graph includes every solution value obtained by running the algorithm 30 times with different configurations, except for the green configurations, which remains constant. Each configuration setting generates multiple solution values, and these are plotted to illustrate the range of solutions obtained.

The bar graph in Figure 3 illustrates the runtime in seconds for each configuration. By examining these graphs, we observe that the blue configuration at $X' = 30$ consistently yields the same solution as the *all seeds* method but requires significantly less time to compute.

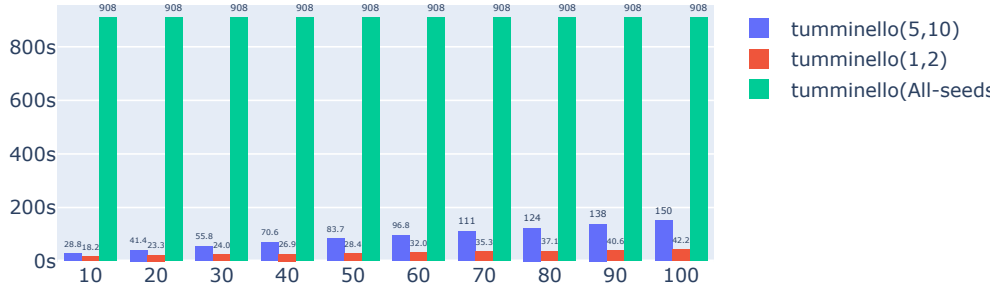



**Figure 3. Runtime (secs) over different configurations for the Tumminello instance.**

## 7. Conclusions

The solution approach proposed here considers only a subset of the possible *seeds* of a graph, instead all the *seeds*, and it also includes the edge dimpling move in the solution construction. The aim of this is to improve the solution quality and running time of the $\mathcal{FD}$ heuristic. Moreover, parallelism played a fundamental role in the approach, being on average 16 times faster for *Coelho's Instances*. The solution quality did not a great impact in comparison with the *All Seeds* method, but it was better than the *TMFG* method and the $\mathcal{FD}$ heuristic, using the *Coelho's Instances*. In the case of the *Tumminello Instance* [Tumminello et al. 2005], we showed how we could get the same result as the *All Seeds* method with only a small fraction of time.

Despite our progress in developing an approximate solution for MWPSP to handle more complex problem instances, our proposed approach does have limitations. First, it requires a data preprocessing step involving the sorting of all seeds. Second, the selection of the heaviest seeds combined with a random set requires parameterization that depends on experimental trials. Finally, the utilisation of a single GPU can pose limitations due to the demand for greater memory in handling larger problems. These aspects are areas we intend to address in our future work.

## References

Ahmadi-Javid, A., Ardestani-Jaafari, A., Foulds, L. R., Hojabri, H., and Farahani, R. Z. (2015). An algorithm and upper bounds for the weighted maximal planar graph problem. *Journal of the Operational Research Society*, 66(8):1399–1412.

Alexander, J. W. (1930). The combinatorial theory of complexes. *Annals of Mathematics*, 31(2):292–320.

Coelho, V. S., Martins, W. S., Foulds, L. R., Dias, E. S., Castonguay, D., and Longo, H. J. (2016). Uma proposta de solução aproximada para o problema do subgrafo planar de peso máximo. In *XVII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2016)*, pages 16–27.

Easley, D. and Kleinberg, J. (2010). *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, Cambridge, UK.

Foulds, L. R. and Robinson, D. F. (1978). Graph theoretic heuristics for the plant layout problem. *International Journal of Production Research*, 16(1):27–37.

Giffin, J. W. (1984). *Graph theoretic techniques for facilities layout*. PhD thesis, University of Canterbury, Christchurch, New Zealand.

Jünger, M. and Mutzel, P. (1993). Solving the Maximum Planar Subgraph Problem by Branch and Cut. In Rinaldi G., W. L., editor, *Proceedings of the 3rd International Conference on Integer Programming and Combinatorial Optimization (IPCO 3)*, pages 479–492.

Kirk, D. B. and Wen-Mei, W. H. (2016). *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, San Francisco.

Lengauer, T. (2012). *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media.

Massara, G. P., Di Matteo, T., and Aste, T. (2017). Network Filtering for Big Data: Triangulated Maximally Filtered Graph. *Journal of Complex Networks*, 5(2):1–18.

Migdalas, A., Pardalos, P., and Storøy, S. (2013). *Parallel Computing in Optimization*. Applied Optimization. Springer Science & Business Media, Heidelburg.

Pesch, E. (1999). Efficient facility layout planning in a maximally planar graph model. *International Journal of Production Research*, 37(2):263–283.

Seppänen, J. and Moore, J. M. (1970). Facilities Planning with Graph Theory. *Management Science*, 17(4):B–242–B–253.

Song, W.-M., Aste, T., and Di Matteo, T. (2007). Correlation-based biological networks. In *Proceedings of International Symposium on Microelectronics, MEMS and Nanotechnology*, volume 6802, pages 680212–1–680212–11, Canberra, Australia.

Tumminello, M., Aste, T., Di Matteo, T., and Mantegna, R. N. (2005). A tool for filtering information in complex systems. *Proceedings of the National Academy of Sciences of the United States of America*, 102(30):10421–10426.