AIMS Mathematics, 9(10): 27308–27329.
DOI: 10.3934/math.20241327
Received: 29 July 2024
Revised: 07 September 2024
Accepted: 14 September 2024
Published: 20 September 2024

https://www.aimspress.com/journal/Math

Research article

On finding a satisfactory partition in an undirected graph: algorithm design and results

Samer Nofal*

Department of Computer Science, German Jordanian University, Amman, Jordan

* **Correspondence:** Email: samer.nofal@gju.edu.jo.

Abstract: A satisfactory partition is a partition of undirected-graph vertices such that the partition has only two nonempty parts, and every vertex has at least as many adjacent vertices in its part as it has in the other part. Generally, the problem of determining whether a given undirected graph has a satisfactory partition is known to be NP-complete. In this paper, we show that for a given undirected graph with n vertices, a satisfactory partition (if any exists) can be computed recursively with a recursion tree of depth of $O(\ln n)$ in expectation. Subsequently, we show that a satisfactory partition for those undirected graphs with recursion tree depth meeting the expectation can be computed in time $O(n^3 2^{O(\ln n)})$.

Keywords: undirected graph; satisfactory partition; exact algorithm; time complexity; expected recursion depth
Mathematics Subject Classification: 05C85, 68W40

1. Introduction

We denote by

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

an undirected graph with a set, $\mathcal{V}$, of n vertices, and a set, $\mathcal{E}$, of m edges such that $\mathcal{E}$ is a set of 2-vertex subsets of $\mathcal{V}$. We say that $u \in \mathcal{V}$ is adjacent to $v \in \mathcal{V}$ in $\mathcal{G}$ if and only if $\{u, v\} \in \mathcal{E}$. We denote by $u \sim v$ an edge $\{u, v\} \in \mathcal{E}$. Let $\mathcal{A}$ and $\mathcal{B}$ be disjoint nonempty subsets of $\mathcal{V}$. We call $\{\mathcal{A}, \mathcal{B}\}$ a *partition* of $\mathcal{G}$ if and only if

$$\mathcal{A} \cup \mathcal{B} = \mathcal{V}.$$

We call $\{\mathcal{A}, \mathcal{B}\}$ a *satisfactory partition* of $\mathcal{G}$ if and only if $\{\mathcal{A}, \mathcal{B}\}$ is a partition of $\mathcal{G}$ such that for every vertex $u \in \mathcal{A}$,

$$|\{v \in \mathcal{A} : u \sim v\}| \geq |\{v \in \mathcal{B} : u \sim v\}|,$$

and for every vertex $x \in \mathcal{B}$,

$$|\{y \in \mathcal{B} : x \sim y\}| \geq |\{y \in \mathcal{A} : x \sim y\}|.$$

See, for example, the graph in Figure 1, where $\{1, 2, 3, 4, 5, 6\}$ and $\{7, 8, 9, 10, 11, 12\}$ are a satisfactory partition of the depicted graph.

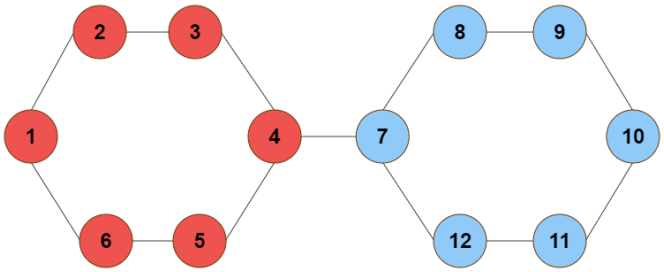

Figure 1. An undirected graph with its satisfactory partition.

The satisfactory partition problem (SPP for short) is the problem of deciding whether an undirected graph $\mathcal{G}$ has a satisfactory partition. The SPP was introduced in a paper [1], where an integer-programming formulation of the SPP was given, and a heuristic procedure was employed for solving the SPP. For an interpretation of the SPP, consider the problem of organizing a sightseeing tour on two boats where it is required to separate the participants into two groups and to satisfy everyone. A participant is satisfied if he knows at least as many people on his boat as on the other. This problem can be seen as an instance of the SPP on a graph where the participants are the vertices, and two vertices are adjacent if the corresponding persons know each other (see the article [1]). Research [2] showed that the SPP can be solved in polynomial time on graphs with bounded clique width. An article [3] studied the complexity of different variants of the SPP. A work [4] proved that the SPP is NP-complete. However, an article [5] showed that for graphs with maximum degree at most 4, the SPP is polynomially solvable. Additionally, a paper [6] presented several parameterized algorithms for solving the SPP.

In this paper, we give new results on the time complexity of computing satisfactory partitions. We prove that for a given undirected graph with n vertices, a satisfactory partition (if any exists) can be computed recursively with a recursion tree of depth $O(\ln n)$ in expectation. Subsequently, we show that a satisfactory partition for those undirected graphs with recursion tree depth matching the expectation can be computed in time $O(n^3 2^{O(\ln n)})$.

The article is structured as follows: Section 2 discusses related work in a broader scope of graph partitioning literature. Section 3 illustrates a recursive algorithm to compute a satisfactory partition of a given undirected graph. Section 3 shows that the algorithm's recursion tree has a depth of $O(\ln n)$ in expectation. In Section 4, we give further arguments on the correctness of our results. Section 5 discusses our results, limitations, and future directions, while section 6 concludes the article.

2. Related work

In the introduction, we discussed closely related work to the study presented in this article. This section gives a glimpse of broader literature overviewing diverse works on various graph processing

and applications in recent years. Thus, a study [7] employed RNA graph partitioning to discover RNA modularity. A work [8] studied subgraphs of pair vertices. An article [9] presented a novel edge detection algorithm based on a hierarchical graph-partition approach. A paper [10] exploited a genetic algorithm for graph partition in a heterogeneous cluster. A study [11] presented a large-scale graph partition algorithm with redundant multi-order neighbor vertex storage. An article [12] investigated forbidden subgraphs in reduced power graphs of finite groups. A work [13, 14] discussed an ant-local search algorithm for the partition graph coloring problem. An article [15] analyzed an efficient and balanced graph partition algorithm for the subgraph-centric programming model on large-scale power-law graphs. A work [16] introduced an improved spectral graph partition intelligent clustering algorithm for low-power wireless networks. A study [17] presented an artificial intelligence knowledge graph for dynamic networks. A paper [18] used a graph partition sampling algorithm in medical intelligent systems and orthopedic clinical nursing. A study [19] proposed a robust spectral clustering algorithm based on grid partition and decision graph. An article [20] argued about improving a graph-based label propagation algorithm with group partition for fraud detection. A study [21] presented results on monochromatic vertex disconnection of graphs. A paper [22] analyzed a task partition algorithm based on grid and graph partition for distributed crowd simulation. An article [23] introduced a novel sports video background segmentation algorithm based on graph partition. A work [24] discussed a property graph partition algorithm based on improved barnacle mating optimization. A study [25] put forward a text mining method of dispatching operation ticket systems based on graph partition spectral clustering algorithms. A work [26] discussed distinguishing colorings of graphs and their subgraphs. A paper [27] proposed an implementation of a parallel graph partition algorithm to speed up computations. A work [28] presented a memetic algorithm with two distinct solution representations for the partition graph coloring problem. A paper [29] studied the informational entropy of B-ary trees after a vertex cut. A work [30] discussed the existence of a graph whose vertex set can be partitioned into a fixed number of strong domination-critical vertex sets. An article [31] examined network bipartitioning in the anti-communicability Euclidean space. The interested reader may find further citations in the reviewed literature to other studies within the broad domain of graph processing algorithms.

3. The algorithm

We formalize the structures of our exact process of finding a satisfactory partition, $\{\mathcal{A}, \mathcal{B}\}$, of a given undirected graph

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}).$$

Hence, let γ: $\mathcal{V} \to \mathbb{N}_0$, α:$\mathcal{V} \to \mathbb{N}_0$, and β: $\mathcal{V} \to \mathbb{N}_0$ be total mappings such that

$$\gamma(v) = |\{u \mid u \sim v\}|, \quad \alpha(v) = 0 \quad \text{and} \quad \beta(v) = 0$$

for all v. During our process of finding a satisfactory partition, $\{\mathcal{A}, \mathcal{B}\}$, of the given graph $\mathcal{G}$, for every vertex $v \in \mathcal{V}$, we use $\alpha(v)$ to track the number of v's adjacent vertices that are in part $\mathcal{A}$, and we use $\beta(v)$ to track the number of v's adjacent vertices that are in part $\mathcal{B}$. Since initially we have empty parts, i.e., $\mathcal{A} = \emptyset$ and $\mathcal{B} = \emptyset$, we initialize $\alpha(v)$ and $\beta(v)$ with 0 for all v. On the other hand, for all vertices v, we employ $\gamma(v)$ to track the number of v's adjacent vertices that are not assigned to a part yet. Subsequently, for every v, $\gamma(v)$ is started with $|\{u \mid u \sim v\}|$.

Additionally, we utilize another total mapping μ: $\mathcal{V} \to \{0, 1, 2\}$ such that for all vertices v, $\mu(v)$ is initialized with 0 to indicate that v is not assigned to either $\mathcal{A}$ nor $\mathcal{B}$. Thus, assigning 1 to $\mu(v)$ means that the vertex v now is a member of $\mathcal{A}$, while assigning 2 to $\mu(v)$ implies that v now is a member of $\mathcal{B}$. Whenever a vertex v joins some part, we update the status of the adjacent vertices of v as follows: If v joins $\mathcal{A}$, then we increment

$$\alpha(u) \leftarrow \alpha(u) + 1$$

for all $u \sim v$. Likewise, if a vertex v joins part $\mathcal{B}$, then we update

$$\beta(u) \leftarrow \beta(u) + 1$$

for all $u \sim v$. Whenever we assign a vertex to some part, we verify that this assignment adheres to the satisfactory partition specification. That is, we check that for every vertex $v \in \mathcal{A}$ (i.e., $\mu(v) = 1$),

$$\alpha(v) + \gamma(v) \geq \beta(v);$$

likewise, we ensure that for every vertex v in part $\mathcal{B}$ (i.e., $\mu(v) = 2$),

$$\beta(v) + \gamma(v) \geq \alpha(v).$$

Consequently, whenever we put a vertex in some part, if for some vertex $v \in \mathcal{V}$, $\mu(v) = 1$ with

$$\beta(v) > \alpha(v) + \gamma(v),$$

or, $\mu(v) = 2$ with

$$\alpha(v) > \beta(v) + \gamma(v),$$

then a contradiction with the satisfactory-partition specification is found; hence we return to a previous stage of the search process where the satisfactory-partition specification is unviolated by revoking one or more vertices from part $\mathcal{A}$ (or from part $\mathcal{B}$) as we elaborate throughout this section.

Further, in finding a satisfactory partition, we perform the following routine every time we put a vertex in some part. For every vertex v in the graph,

(i) If $v \in \mathcal{A}$ (i.e., $\mu(v) = 1$) and half of the adjacent vertices of v are in part $\mathcal{B}$, then we assign to part $\mathcal{A}$ the adjacent vertices of v that are not assigned to any part yet;

(ii) If $v \in \mathcal{B}$ (i.e., $\mu(v) = 2$) and half of the adjacent vertices of v are in part $\mathcal{A}$, then we assign to part $\mathcal{B}$ the adjacent vertices of v that are not assigned to any part yet. This routine is repeated every time we put a vertex in some part. We stress that assigning a vertex to some part may require other vertices, which are not assigned to any part yet, to join a specific part to fulfill the satisfactory partition requirement.

Our exact process for constructing a satisfactory partition of a given graph

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

is rigorously described in Algorithm 1. The algorithm finds a satisfactory partition (if any exists) immediately after $\Sigma(\emptyset, \mathcal{V}, \mu, \alpha, \beta, \gamma)$ is invoked with μ, α, β, and γ being initialized such that every vertex $y \in \mathcal{V}$ has

$$\mu(y) = 0, \quad \gamma(y) = |\{z : z \sim y\}|, \quad \alpha(y) = 0 \quad \text{and} \quad \beta(y) = 0.$$

We shortly explain the first parameter (i.e., $\emptyset$) passed to the algorithm. The second parameter of our procedure Σ is initially the whole set, $\mathcal{V}$, of the graph vertices. Referring to lines 30–31, a call for a new instance, k, of the algorithm (i.e., the procedure Σ) entails creating a new copy of the passed structures such that the last copy of the structures (belonging to the caller instance, $k-1$, of Σ) are not affected by the updates carried on during the execution of instance k. In other words, all the calls for the procedure Σ are done by passing a *copy* of the parameters.

Algorithm 1: $\Sigma(\Delta, \mathcal{V}, \mu, \alpha, \beta, \gamma)$.

1 **while** $\Delta \neq \emptyset$ **do**
2 Extract a vertex v from Δ;
3 **if** $\mu(v) = 1$ **then**
4 **if** $\beta(v) > \alpha(v) + \gamma(v)$ **then** return;
5 **if** $\alpha(v) + \gamma(v) - 1 \leq \beta(v) \leq \alpha(v) + \gamma(v)$ **then**
6 **foreach** $u \sim v$ *with* $\mu(u) = 0$ **do**
7 $\mu(u) \leftarrow 1$; $\mathcal{V} \leftarrow \mathcal{V} \setminus \{u\}$; $\Delta \leftarrow \Delta \cup \{u\}$;
8 **foreach** $u \sim v$ **do**
9 $\alpha(u) \leftarrow \alpha(u) + 1$; $\gamma(u) \leftarrow \gamma(u) - 1$;
10 **if** $\mu(u) = 2 \wedge \alpha(u) > \beta(u) + \gamma(u)$ **then** return;
11 **if** $\mu(u) = 2 \wedge \beta(u) + \gamma(u) - 1 \leq \alpha(u) \leq \beta(u) + \gamma(u)$ **then**
12 **foreach** $s \sim u$ *with* $\mu(s) = 0$ **do**
13 $\mu(s) \leftarrow 2$; $\mathcal{V} \leftarrow \mathcal{V} \setminus \{s\}$; $\Delta \leftarrow \Delta \cup \{s\}$;
14 **if** $\mu(v) = 2$ **then**
15 **if** $\alpha(v) > \beta(v) + \gamma(v)$ **then** return;
16 **if** $\beta(v) + \gamma(v) - 1 \leq \alpha(v) \leq \beta(v) + \gamma(v)$ **then**
17 **foreach** $u \sim v$ *with* $\mu(u) = 0$ **do**
18 $\mu(u) \leftarrow 2$; $\mathcal{V} \leftarrow \mathcal{V} \setminus \{u\}$; $\Delta \leftarrow \Delta \cup \{u\}$;
19 **foreach** $u \sim v$ **do**
20 $\beta(u) \leftarrow \beta(u) + 1$; $\gamma(u) \leftarrow \gamma(u) - 1$;
21 **if** $\mu(u) = 1 \wedge \beta(u) > \alpha(u) + \gamma(u)$ **then** return;
22 **if** $\mu(u) = 1 \wedge \alpha(u) + \gamma(u) - 1 \leq \beta(u) \leq \alpha(u) + \gamma(u)$ **then**
23 **foreach** $s \sim u$ *with* $\mu(s) = 0$ **do**
24 $\mu(s) \leftarrow 1$; $\mathcal{V} \leftarrow \mathcal{V} \setminus \{s\}$; $\Delta \leftarrow \Delta \cup \{s\}$;
25 **if** $\mathcal{V} = \emptyset$ **then**
26 **if** $\{v \mid \mu(v) = 1\} \neq \emptyset \wedge \{v \mid \mu(v) = 2\} \neq \emptyset$ **then**
27 μ is a satisfactory partition; end the execution of the algorithm;
28 **else**
29 Extract (a randomly selected) v from $\mathcal{V}$;
30 select randomly i from $\{1, 2\}$; $\mu(v) \leftarrow i$; $\Sigma(\{v\}, \mathcal{V}, \mu, \alpha, \beta, \gamma)$;
31 Let $j \in \{1, 2\}$ but $j \neq i$; $\mu(v) \leftarrow j$; $\Sigma(\{v\}, \mathcal{V}, \mu, \alpha, \beta, \gamma)$.

Let us go through the algorithm's actions in further detail, line by line. Referring to the caption of Algorithm 1, note that our procedure Σ is recursive (see lines 30 and 31). As noted earlier, we initially

run Σ with $\Delta = \emptyset$, the whole vertex set of the given graph $\mathcal{V}$, and the total mappings $\mu, \alpha, \beta, \gamma$ such that

$$\mu(v) = \alpha(v) = \beta(v) = 0$$

while

$$\gamma(v) = |\{u : u \sim v\}|$$

for all $v \in \mathcal{V}$. Thus, because Δ is initially empty, we delay discussing the purpose of Δ and the *while* loop at line 1 in the algorithm. Likewise, since $\mathcal{V}$ is initially nonempty, let us skip line 25 and go to line 29, where we take a vertex v out of $\mathcal{V}$. Referring to line 30 (respectively line 31), supposing $i = 1$ (respectively $j = 2$), we assign v to part $\mathcal{A}$ (respectively part $\mathcal{B}$) by applying $\mu(v) \leftarrow 1$ (respectively $\mu(v) \leftarrow 2$). Subsequently, we run the algorithm again by invoking $\Sigma(\{v\}, \mathcal{V}, \mu, \alpha, \beta, \gamma)$ trying to construct a satisfactory partition with $v \in \mathcal{A}$ (see line 30). If this attempt is unsuccessful, we try constructing a satisfactory partition with v in $\mathcal{B}$ (see line 31).

Note that during the very first call for the procedure Σ, there is no need to try constructing a satisfactory partition by assigning v (the extracted vertex from $\mathcal{V}$ in line 29) to part $\mathcal{B}$, as this is symmetric to the scenario where $v \in \mathcal{A}$. Suppose no satisfactory partition is possible with $v \in \mathcal{A}$. In that case, there is no satisfactory partition possible with $v \in \mathcal{B}$, which means that in the initial call for Σ, we can omit to execute line 31. We leave this tiny detail out of Algorithm 1. However, suppose one wants to include such detail. In that case, the algorithm should be started with $\Sigma(\{x\}, \mathcal{V} \setminus \{x\}, \mu, \alpha, \beta, \gamma)$, where the first parameter (i.e., $\{x\}$) indicates that the process of constructing a satisfactory partition is started by assigning some vertex x to some part. As said earlier, it does not matter whether x is set to part $\mathcal{A}$ or $\mathcal{B}$. Hence, one might start the algorithm by assigning x to part $\mathcal{A}$. Therefore, x must be removed from the vertex set as indicated by $\mathcal{V} \setminus \{x\}$, the second parameter passed to the algorithm. For the rest of the parameters (i.e., μ, α, β, and γ), we mentioned earlier that these structures are usually initialized such that for every vertex $y \in \mathcal{V}$,

$$\mu(y) = 0, \quad \gamma(y) = |\{z : z \sim y\}|, \quad \alpha(y) = 0 \quad \text{and} \quad \beta(y) = 0.$$

But since we start the algorithm with some vertex x being included in part $\mathcal{A}$, we must initially set $\mu(x) \leftarrow 1$.

Back to the description of Algorithm 1, now assume that the algorithm has been recursively invoked with $\Sigma(\{v\}, \mathcal{V}, \mu, \alpha, \beta, \gamma)$ where v is assigned to either part $\mathcal{A}$ or part $\mathcal{B}$ (see lines 30 and 31). At this point, we note that $\Delta = \{v\}$. So, we run the *while* loop (line 1) and extract v from Δ according to line 2. The purpose of Δ is to temporarily hold those vertices that are newly assigned to some part until we accordingly update the total mappings (employed by the algorithm) as we elaborate next. Referring to line 3, we check if v was assigned to part $\mathcal{A}$ (i.e., $\mu(v) = 1$), and if so, the following actions need to be performed. In line 4, we examine if the number, $\beta(v)$, of v's adjacent vertices that are in part $\mathcal{B}$ is greater than the sum of the number, $\alpha(v)$, of v's adjacent vertices contained in part $\mathcal{A}$ plus the number, $\gamma(v)$, of v's adjacent vertices that are not assigned to any part yet; if it is the case that

$$\beta(v) > \alpha(v) + \gamma(v),$$

then the algorithm returns to the previous instance of Σ. In line 5, we verify if the number, $\beta(v)$, of v's adjacent vertices that are in part $\mathcal{B}$ has reached the maximum allowed value of $\lfloor \frac{|\{w:w\sim v\}|}{2} \rfloor$ by examining

the inequality laid down in line 5; if this inequality is true then the v's adjacent vertices that are not assigned to any part yet must join part $\mathcal{A}$ (the part of v); see lines 6 and 7 where for every u adjacent to v with $\mu(u) = 0$, we assign u to part $\mathcal{A}$, remove u from $\mathcal{V}$, and then put u in Δ. To see that the inequality in line 5 is true if

$$\beta(v) = \lfloor \frac{|\{w : w \sim v\}|}{2} \rfloor,$$

let

$$k = |\{w : w \sim v\}|.$$

Recall that throughout the algorithm it is the case that

$$\beta(v) + \alpha(v) + \gamma(v) = |\{w : w \sim v\}|.$$

If k is even, then the inequality is

$$\frac{k}{2} - 1 \leq \frac{k}{2} \leq \frac{k}{2}.$$

If k is odd, then the inequality is

$$\frac{k+1}{2} - 1 \leq \frac{k-1}{2} \leq \frac{k+1}{2}.$$

Now we show that if the inequality (in line 5) is true, then it is the case that

$$\beta(v) = \lfloor \frac{|\{w : w \sim v\}|}{2} \rfloor.$$

Given that

$$\beta(v) + \alpha(v) + \gamma(v) = |\{w : w \sim v\}|,$$

the inequality can be rewritten as

$$\alpha(v) + \gamma(v) - 1 \leq |\{w : w \sim v\}| - \alpha(v) - \gamma(v) \leq \alpha(v) + \gamma(v),$$

which is

$$2\alpha(v) + 2\gamma(v) - 1 \leq |\{w : w \sim v\}| \leq 2\alpha(v) + 2\gamma(v).$$

Divide all sides by two and then take the floor value. The inequality becomes

$$\left\lfloor \alpha(v) + \gamma(v) - \frac{1}{2} \right\rfloor \leq \left\lfloor \frac{|\{w : w \sim v\}|}{2} \right\rfloor \leq \lfloor \alpha(v) + \gamma(v) \rfloor .$$

Hence,

$$\alpha(v) + \gamma(v) - 1 \leq \left\lfloor \frac{|\{w : w \sim v\}|}{2} \right\rfloor \leq \alpha(v) + \gamma(v).$$

That is,

$$\beta(v) = \left\lfloor \frac{|\{w : w \sim v\}|}{2} \right\rfloor.$$

Back to the algorithm. In line 8, we process the adjacent vertices of v as follows: For every vertex $u \sim v$, we update

$$\alpha(u) \leftarrow \alpha(u) + 1 \text{ and } \gamma(u) \leftarrow \gamma(u) - 1 \text{ (line 9).}$$

Subsequently, we check whether the specifications of satisfactory partition are violated as a consequence of v being included in part $\mathcal{A}$. More specifically, referring to line 10, if a vertex u (such that $u \sim v$) is in part $\mathcal{B}$, then we verify that $\alpha(u)$, the number of adjacent vertices of u that are in part $\mathcal{A}$, is still less than or equal to $\beta(u) + \gamma(u)$, which is the number of adjacent vertices of u that are either in the part of u (i.e., $\mathcal{B}$) or not assigned to any part. Otherwise, again referring to line 10, the algorithm returns to the previous state (i.e., the algorithm returns to the caller instance of Σ). Moving on to line 11, the algorithm tries to find out if any adjacent vertices of u must join a specific part, according to the definition of satisfactory partition. Therefore, in line 11, we check that if u is in part $\mathcal{B}$ and that the number of u's adjacent vertices in part $\mathcal{A}$ has reached the maximum allowed value, i.e.,

$$\alpha(u) = \lfloor \frac{|\{w : w \sim u\}|}{2} \rfloor,$$

then for every vertex $s \sim u$ such that s is not assigned to any part yet (lines 11 and 12), s must join part $\mathcal{B}$, and consequently s is removed from $\mathcal{V}$ and then included in Δ (line 13).

The rest of the *while* loop, lines 14–24 in Algorithm 1, deal with the case where the vertex v (extracted from Δ in line 2) is assigned to part $\mathcal{B}$. This is analogous to the scenario of v being assigned to part $\mathcal{A}$ (lines 3–13). However, we trace lines 14–24 for the reader's convenience. Referring to line 15, if the number, $\alpha(v)$, of v's adjacent vertices that are in part $\mathcal{A}$ is greater than the sum of the number, $\beta(v)$, of v's adjacent vertices that are in part $\mathcal{B}$ plus the number, $\gamma(v)$, of v's adjacent vertices that are not assigned to any part, then the algorithm returns to the previous stage (i.e., to the caller instance of Σ). As to line 16, if the number, $\alpha(v)$, of v's adjacent vertices that are in part $\mathcal{A}$ has reached the maximum allowed value, i.e.,

$$\alpha(u) = \lfloor \frac{|\{w : w \sim u\}|}{2} \rfloor,$$

then, in lines 17 and 18, we put into part $\mathcal{B}$ every $u \sim v$ with $\mu(u) = 0$ (i.e., every $u \sim v$ not assigned a part yet). In line 18, we move u from $\mathcal{V}$ to Δ so that u is processed in a subsequent round of the *while* loop. In lines 19–20, we update

$$\beta(u) \leftarrow \beta(u) + 1$$

and

$$\gamma(u) \leftarrow \gamma(u) - 1$$

for every $u \sim v$. In line 21, if a vertex $u \sim v$ is in part $\mathcal{A}$ and the number, $\beta(u)$, of u's adjacent vertices that are in part $\mathcal{B}$ exceeds the sum of the number, $\alpha(u)$, of u's adjacent vertices that are in part $\mathcal{A}$ plus the number, $\gamma(u)$, of u's adjacent vertices that are not assigned to any part, then the algorithm returns to the previous state (i.e., the algorithm returns to the caller instance of the procedure Σ). In line 22, we check whether there is a vertex $u \sim v$ such that $u \in \mathcal{A}$ (i.e., $\mu(u) = 2$), but the number, $\beta(u)$, of u's adjacent vertices belonging to part $\mathcal{B}$, has reached the maximum allowed value, i.e.,

$$\beta(u) = \lfloor \frac{|\{w : w \sim u\}|}{2} \rfloor;$$

if this is true, then we put each $s \sim u$ into part $\mathcal{A}$, provided that s is not already included in any part; see lines 23 and 24. In line 24, we move s from $\mathcal{V}$ to Δ so that s is processed further in a later round of the *while* loop.

Thereby, the *while* loop continues as long as Δ has some vertices that need to be processed. In summary, we note that the *while* loop's actions ensure that the current *part* assignment of graph vertices is consistent with the satisfactory partition specifications. Going beyond the *while* loop, in line 25, the algorithm checks if all the graph vertices are included in some part; if not, i.e., if $\mathcal{V} \neq \emptyset$, then we repeat the same process as discussed above by re-applying lines 29–31, and so forth. Referring to the lines 25–27, if $\mathcal{V} = \emptyset$ (line 25) with $\mathcal{A}$ and $\mathcal{B}$ being nonempty (line 26), then we stop the search process for a satisfactory partition since $\mathcal{A}$ and $\mathcal{B}$ is a satisfactory partition of the given graph. Referring to line 26, recall that the set

$$\{v \mid \mu(v) = 1\}$$

designates part $\mathcal{A}$, whereas the set

$$\{v \mid \mu(v) = 2\}$$

denotes part $\mathcal{B}$. By this, we completed a description of Algorithm 1 and its structures. The next section discusses the algorithm's running time (and running space).

Example 1. *Let us demonstrate the operation of Algorithm 1 on the graph depicted in Figure 2.*

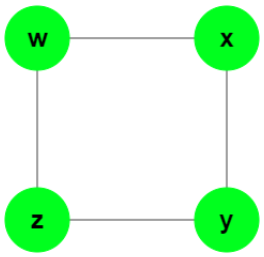

Figure 2. An undirected graph.

Initially, we invoke $\Sigma(\Delta, \mathcal{V}, \mu, \alpha, \beta, \gamma)$ with

$$\begin{aligned}
\Delta &= \emptyset,\\
\mathcal{V} &= \{w, x, y, z\},\\
\mu &= \{(w, 0), (x, 0), (y, 0), (z, 0)\},\\
\alpha &= \{(w, 0), (x, 0), (y, 0), (z, 0)\},\\
\beta &= \{(w, 0), (x, 0), (y, 0), (z, 0)\},\\
\gamma &= \{(w, 2), (x, 2), (y, 2), (z, 2)\}.
\end{aligned}$$

Assume that w is extracted from the vertex set (in line 29). Then, in line 30, suppose w is assigned to the first part $\mathcal{A}$ by labeling it with $\mu(w) \leftarrow 1$. Then, we invoke $\Sigma(\Delta, \mathcal{V}, \mu, \alpha, \beta, \gamma)$ with

$$\begin{aligned}
\Delta &= \{w\},\\
\mathcal{V} &= \{x, y, z\},\\
\mu &= \{(w, 1), (x, 0), (y, 0), (z, 0)\},\\
\alpha &= \{(w, 0), (x, 0), (y, 0), (z, 0)\},
\end{aligned}$$

$$\beta = \{(w, 0), (x, 0), (y, 0), (z, 0)\},$$
$$\gamma = \{(w, 2), (x, 2), (y, 2), (z, 2)\}.$$

Now, line 2 is operated to extract w from Δ, which makes Δ empty; then, line 9 is executed twice for w's adjacent vertices x and z such that

$$\alpha(x) = 1, \quad \gamma(x) = 1, \quad \alpha(z) = 1 \quad \text{and} \quad \gamma(z) = 1.$$

Thus, after execution of this round of the while loop, the state is

$$\begin{aligned}
\Delta &= \emptyset, \\
\mathcal{V} &= \{x, y, z\}, \\
\mu &= \{(w, 1), (x, 0), (y, 0), (z, 0)\}, \\
\alpha &= \{(w, 0), (x, 1), (y, 0), (z, 1)\}, \\
\beta &= \{(w, 0), (x, 0), (y, 0), (z, 0)\}, \\
\gamma &= \{(w, 2), (x, 1), (y, 2), (z, 1)\}.
\end{aligned}$$

Now, line 29 is executed. Assume that x is extracted from $\mathcal{V}$. Then, in line 30, suppose x is assigned to part $\mathcal{B}$ by labeling it with $\mu(x) \leftarrow 2$. Afterwards, in line 30, we invoke $\Sigma(\Delta, \mathcal{V}, \mu, \alpha, \beta, \gamma)$ with

$$\begin{aligned}
\Delta &= \{x\}, \\
\mathcal{V} &= \{y, z\}, \\
\mu &= \{(w, 1), (x, 2), (y, 0), (z, 0)\}, \\
\alpha &= \{(w, 0), (x, 1), (y, 0), (z, 1)\}, \\
\beta &= \{(w, 0), (x, 0), (y, 0), (z, 0)\}, \\
\gamma &= \{(w, 2), (x, 1), (y, 2), (z, 1)\}.
\end{aligned}$$

Now, we track the actions taken in a round of the while loop. Line 2 is operated to extract x from Δ, which makes Δ empty. Lines 17 and 18 are executed such that

$$\mu(y) = 2, \quad \mathcal{V} = \{z\}, \quad \Delta = \{y\}.$$

Lines 19 and 20 are operated such that

$$\beta(w) = 1, \quad \gamma(w) = 1, \quad \beta(y) = 1, \quad \gamma(y) = 1.$$

Then, lines 23 and 24 are executed such that

$$\mu(z) = 1, \quad \mathcal{V} = \emptyset, \quad \Delta = \{y, z\}.$$

In summary, after execution of this round of the while loop, the state is

$$\Delta = \{y, z\},$$

$$\begin{aligned}
&\mathcal{V} = \emptyset,\\
&\mu = \{(w,1),(x,2),(y,2),(z,1)\},\\
&\alpha = \{(w,0),(x,1),(y,0),(z,1)\},\\
&\beta = \{(w,1),(x,0),(y,1),(z,0)\},\\
&\gamma = \{(w,1),(x,1),(y,1),(z,1)\}.
\end{aligned}$$

The following actions are taken in the next round of the while loop. Line 2 is run to extract y from Δ, so

$$\Delta = \{z\}.$$

Lines 19 and 20 are executed such that

$$\beta(x) = 1, \quad \gamma(x) = 0, \quad \beta(z) = 1, \quad \gamma(z) = 0.$$

The state after execution of the second round is

$$\begin{aligned}
&\Delta = \{z\},\\
&\mathcal{V} = \emptyset,\\
&\mu = \{(w,1),(x,2),(y,2),(z,1)\},\\
&\alpha = \{(w,0),(x,1),(y,0),(z,1)\},\\
&\beta = \{(w,1),(x,1),(y,1),(z,1)\},\\
&\gamma = \{(w,1),(x,0),(y,1),(z,0)\}.
\end{aligned}$$

In the following round of the while loop, line 2 is run to extract z from Δ, so

$$\Delta = \emptyset.$$

Lines 8 and 9 are executed such that

$$\alpha(w) = 1, \quad \gamma(w) = 0, \quad \alpha(y) = 1, \quad \gamma(y) = 0.$$

Now, the state is

$$\begin{aligned}
&\Delta = \emptyset,\\
&\mathcal{V} = \emptyset,\\
&\mu = \{(w,1),(x,2),(y,2),(z,1)\},\\
&\alpha = \{(w,1),(x,1),(y,1),(z,1)\},\\
&\beta = \{(w,1),(x,1),(y,1),(z,1)\},\\
&\gamma = \{(w,0),(x,0),(y,0),(z,0)\}.
\end{aligned}$$

Since Δ is empty now, there are no more rounds of the while loop. Hence, line 27 is executed to terminate the algorithm as a satisfactory partition $\{\{x,y\},\{w,z\}\}$ is found.

4. Algorithm complexity

We first discuss the implementation of $\mathcal{V}$ in Algorithm 1. We implement $\mathcal{V}$ as a variable-size array, $\mathcal{R}$, of $|\mathcal{V}|$ pairs where each pair has a vertex v and a boolean value b reflecting whether v is deleted from $\mathcal{V}$ or not such that

$$b = true$$

means that the vertex v is still in $\mathcal{V}$. Now, deleting v from $\mathcal{V}$ is implemented by firstly locating v in $\mathcal{R}$ using another fixed-size array $\mathcal{I}[v]$ that holds the current position of v in $\mathcal{R}$. Note that throughout the algorithm's execution, the size of $\mathcal{I}$ constantly equals the number of vertices of the originally inputted graph. Hence, by applying

$$\mathcal{R}[\mathcal{I}[v]].b \leftarrow false,$$

we delete v from $\mathcal{V}$. Therefore, deleting one vertex from $\mathcal{V}$ costs constant time, which implies that line 29 in the algorithm runs in constant time.

However, observe that before we randomly select a vertex to be extracted from $\mathcal{V}$ (line 29), we must shrink the $\mathcal{V}$'s underlying array $\mathcal{R}$ due to vertex removal from $\mathcal{V}$ (that happened in lines 7, 13, 18, 24, and 29 since the last time we ran line 29). To this end, we create an array, $\mathcal{R}'$, of pairs (v, b) with a size of $|\mathcal{R}| - d$, where d is the number of vertex deletions that occurred since the last time we shrank $\mathcal{R}$. Then, we check all $\mathcal{R}[k]$; if

$$\mathcal{R}[k].b = true,$$

then we copy $\mathcal{R}[k]$ into $\mathcal{R}'[m]$ and perform

$$\mathcal{I}[\mathcal{R}[k].v] \leftarrow m.$$

After processing all $\mathcal{R}[k]$, we cancel $\mathcal{R}$ and replace it with $\mathcal{R}'$ to represent the current $\mathcal{V}$.

Now, we show that the recursion tree of Algorithm 1 has a logarithmic depth in expectation.

Theorem 1. *For a given undirected graph with n vertices, the recursion tree of Algorithm 1 has a depth $O(\ln n)$ in expectation.*

Proof. Whenever we make a *non-terminal* recursive call to the algorithm, we extract a randomly selected vertex v from the current vertex set $\mathcal{V}$; see line 29 in the algorithm. So, the expected depth of the recursion tree of Algorithm 1 is

$$\sum_{v=1}^{n} E_v(\mathcal{X}),$$

where $E_v(\mathcal{X})$ is the expected number of times a vertex v is selected and extracted within a simple, complete path of the recursion tree starting from the root call until a terminal call where $\mathcal{V}$ is empty (see line 25 in the algorithm).

Considering line 29, let $\mathcal{V} = \{1, ..., i\}$ such that $1 \leq i \leq n$. Recall that throughout the algorithm's execution, $\mathcal{V}$ is a subset of the vertex set of the originally inputted graph. At the start of the execution, $\mathcal{V}$ contains n vertices. But as we go on with the algorithm's execution, $\mathcal{V}$ is reduced to $i \leq n$ vertices as an effect of executing lines 7, 13, 18, 24, and 29 in the algorithm. The algorithm keeps reducing $\mathcal{V}$ across multiple recursive calls until $\mathcal{V}$ is empty; see line 25 in Algorithm 1.

To calculate the expected number of times a given vertex v is selected and extracted within a simple, complete path of the recursion tree, let

$$\mathcal{X} : \{(\{1, 2, ..., i\}, v) \mid 1 \leq i \leq n\} \rightarrow \{0, 1\}$$

be a random variable that represents the number of times we select and extract a given vertex v from a set of i vertices. For every i, it is the case that

$$\mathcal{X}((\{1, 2..., i\}, v)) = 1$$

if $1 \leq v \leq i$ and otherwise

$$\mathcal{X}((\{1, 2..., i\}, v)) = 0.$$

For a given $i \leq n$, it is the case that

$$\mathcal{P}(\mathcal{X}((\{1, 2, ..., i\}, v)) = 1) = \left(\frac{1}{i}\right)\left(\frac{1}{n}\right),$$

where $\frac{1}{i}$ is the probability of selecting v from $i \leq n$ vertices and, $\frac{1}{n}$ is the probability of having $i \leq n$ vertices. Hence, the expected depth of the recursion tree of Algorithm 1 is

$$\begin{aligned}\sum_{v=1}^{n} E_v(\mathcal{X}) &= \sum_{v=1}^{n}\sum_{i=1}^{n} \mathcal{X}((\{1, 2, ..., i\}, v))\mathcal{P}(\mathcal{X}((\{1, 2, ..., i\}, v)) = 1) \\ &= \sum_{v=1}^{n}\sum_{i=1}^{n}(1)\left(\frac{1}{i}\right)\left(\frac{1}{n}\right) = \sum_{v=1}^{n}\left(\frac{1}{n}\right)\sum_{i=1}^{n}\left(\frac{1}{i}\right) = \sum_{i=1}^{n}\left(\frac{1}{i}\right).\end{aligned}$$

Thus,

$$\sum_{v=1}^{n} E_v(\mathcal{X}) \in O(\ln n).$$

This ends the proof. □

In the following theorem, we illustrate the overall time complexity of Algorithm 1 for cases where the recursion tree of the algorithm has a logarithmic depth.

Theorem 2. *Let $\mathcal{G}$ be an undirected graph with n vertices such that the Algorithm 1's recursion tree depth matches the expectation, i.e., $O(\ln n)$. Then, Algorithm 1 runs in time $O(n^3 2^{O(\ln n)})$.*

Proof. Observe that the algorithm's running time is bounded by the number of Σ calls multiplied by the running time of the *while* loop (in line 1 in the algorithm). However, the *while* loop requires at most $O(n^3)$ time due to the three nested loops that have no more than n rounds each. Referring to lines 29–31 in the algorithm, assume two Σ calls are made for every graph vertex. Then, the algorithm's running time in this extreme scenario is bounded by $O(n^3 2^n)$. Nevertheless, as the depth of the recursion tree of Algorithm 1 is $O(\ln n)$, the recursion tree size is $O(2^{O(\ln n)})$. Consequently, the overall running time of Algorithm 1 is estimated by the running time of the *while* loop (i.e., $O(n^3)$) multiplied by the recursion tree size (i.e., $O(2^{O(\ln n)})$). This means Algorithm 1 runs in time $O(n^3 2^{O(\ln n)})$. □

Regarding the space complexity of Algorithm 1, we note that the input graph requires $O(n^2)$ space. In the following theorem, we discuss the *additional* space needed by Algorithm 1 other than the space required to hold the input graph. Next, we show that the *additional* space of Algorithm 1 is linearithmic in expectation.

Theorem 3. *For a given undirected graph with n vertices, Algorithm 1 runs in additional space* $O(n \ln n)$ *in expectation.*

Proof. We note that the maximum size of any structure employed by the algorithm is $O(n)$ and the expected depth of the recursion of the algorithm is $O(n)$, which implies that the maximum number of copies of any structure employed by the algorithm is $O(n)$. Thus, the additional space of Algorithm 1 is $O(n^2)$. But since the expected depth of the recursion of the algorithm is logarithmic, as established earlier, the expected additional space is $O(n \ln n)$. □

5. Algorithm correctness

The following technical lemmata together solidify the validity of Algorithm 1. Recall that the algorithm finds (if any exists) a satisfactory partition $\{\mathcal{A}, \mathcal{B}\}$ for a given undirected graph $\mathcal{G}$. In the following lemma, we state that throughout the execution of our algorithm, the algorithm (line 3) checks *every* vertex assigned to part $\mathcal{A}$; likewise, the algorithm (line 14) checks *every* vertex put in part $\mathcal{B}$.

Lemma 1. *Throughout the execution of Algorithm 1 on a graph* $\mathcal{G}$*, for all vertices y of* $\mathcal{G}$*, whenever y is put into part* $\mathcal{A}$ *or part* $\mathcal{B}$*, then the algorithm examines the adjacent vertices, z, of y to update the mappings* $\alpha(z)$*,* $\beta(z)$*, and* $\gamma(z)$ *accordingly.*

Proof. Recall that assigning $\mu(x)$ to 1 means that x is included in part $\mathcal{A}$, while $\mu(x)$ getting 2 indicates that x is assigned to part $\mathcal{B}$. Now, it suffices to note that whenever the algorithm assigns a vertex to a part, it immediately adds the vertex to Δ; see lines 7, 13, 18, 24, 30, and 31. Subsequently, the algorithm processes every vertex v in Δ; see line 2; and later, the algorithm examines all v's adjacent vertices, u, to update the mappings $\alpha(u)$, $\beta(u)$, and $\gamma(u)$ accordingly; see lines 3, 8, 9, 14, 19, and 20. □

The following three lemmata show the correct usage of the mappings α, β, and γ, respectively.

Lemma 2. *Throughout the execution of Algorithm 1 on a graph* $\mathcal{G}$*, for any vertex y in* $\mathcal{G}$*,* $\alpha(y)$ *indicates exactly the number of y's adjacent vertices that are in part* $\mathcal{A}$*.*

Proof. Recollect that our algorithm is started with

$$\alpha(y) = 0$$

for every vertex y in $\mathcal{G}$. Likewise, the algorithm is started with

$$\mu(y) = 0$$

for all y in $\mathcal{G}$, which means that the algorithm is started with part $\mathcal{A}$ being empty. Throughout its execution, the algorithm performs

$$\alpha(y) \leftarrow \alpha(y) + 1$$

for any vertex y whenever an adjacent vertex of y is assigned to part $\mathcal{A}$, see lines 3, 8, and 9. Recall that, by definition, assigning $\mu(x) \leftarrow 1$ for any vertex x is equivalent to assigning x to part $\mathcal{A}$. □

Lemma 3. *Throughout the execution of Algorithm 1 on a graph $\mathcal{G}$, for any vertex y in $\mathcal{G}$, $\beta(y)$ indicates exactly the number of y's adjacent vertices that are in part $\mathcal{B}$.*

Proof. Recall that our algorithm is started with

$$\beta(y) = 0$$

for every vertex y in $\mathcal{G}$. Additionally, the algorithm is started with

$$\mu(y) = 0$$

for all y in $\mathcal{G}$, which implies that part $\mathcal{B}$ is initially empty. The algorithm updates

$$\beta(y) \leftarrow \beta(y) + 1$$

for any vertex y whenever the algorithm assigns an adjacent vertex of y to part $\mathcal{B}$, see lines 14, 19, and 20. By definition, assigning $\mu(x) \leftarrow 2$ for any vertex x implies putting x into part $\mathcal{B}$. □

Lemma 4. *Throughout the execution of Algorithm 1 on a graph $\mathcal{G}$, for any vertex y in $\mathcal{G}$, $\gamma(y)$ indicates exactly the number of y's adjacent vertices that are not assigned to any part.*

Proof. Consider that our algorithm is started with

$$\gamma(y) = |\{x : x \sim y\}|$$

for every vertex y in $\mathcal{G}$. Similarly, the algorithm is started with part $\mathcal{A}$ and part $\mathcal{B}$ being empty. The algorithm updates

$$\gamma(y) \leftarrow \gamma(y) - 1 \quad \text{(lines 9 and 20)}$$

for any vertex y whenever the algorithm assigns an adjacent vertex of y to either part $\mathcal{A}$ (see lines 3, 8, and 9) or part $\mathcal{B}$ (see lines 14, 19, and 20). □

Now, we emphasize the integrity of line 4, where Algorithm 1 might return to a previous point of the search process under some conditions, as detailed in the following lemma:

Lemma 5. *For any graph $\mathcal{G}$, at any stage of the execution of Algorithm 1, if there is $v \in \mathcal{A}$ (i.e., $\mu(v) = 1$) with*

$$\beta(v) > \alpha(v) + \gamma(v),$$

then there is no

$$\mathcal{A}' \supseteq \mathcal{A}, \quad \mathcal{B}' \supseteq \mathcal{B}$$

such that $\{\mathcal{A}', \mathcal{B}'\}$ is a satisfactory partition of $\mathcal{G}$.

Proof. This follows directly from the definition of satisfactory partition. But we mean here to highlight that the premise of this lemma corresponds to the conditions of lines 3 and 4. Likewise, we stress that the consequence of this lemma agrees with the action of Algorithm 1 as declared in line 4, where the algorithm returns to a previous instance of the procedure Σ, which holds the previous copy of an under-construction satisfactory partition. □

Next, we show the soundness of line 10, where Algorithm 1 goes back to a previous point of the search process under some conditions, as illustrated in the following lemma:

Lemma 6. *For any graph $\mathcal{G}$, at any stage of the execution of Algorithm 1, if there is a vertex $v \in \mathcal{A}$ (i.e., $\mu(v) = 1$) with a vertex $u \sim v$ such that $u \in \mathcal{B}$ (i.e., $\mu(u) = 2$) and*

$$\alpha(u) > \beta(u) + \gamma(u),$$

then there is no

$$\mathcal{A}' \supseteq \mathcal{A}, \quad \mathcal{B}' \supseteq \mathcal{B}$$

such that $\{\mathcal{A}', \mathcal{B}'\}$ is a satisfactory partition of $\mathcal{G}$.

Proof. This lemma follows directly from the definition of satisfactory partition. Observe that the premise of this lemma corresponds to the conditions of lines 3, 8, and 10 in Algorithm 1. Similarly, the consequence of this lemma is consistent with the action of Algorithm 1 as stated in line 10, where the algorithm returns to the previous instance of the procedure Σ that holds the previous copy of an under-construction satisfactory partition. □

In the following lemma, we focus on the correctness of line 15, where Algorithm 1 might return to a previous point of the search process under some conditions, as detailed next.

Lemma 7. *For any graph $\mathcal{G}$, at any stage of the execution of Algorithm 1, if there is $v \in \mathcal{B}$ (i.e., $\mu(v) = 2$) with*

$$\alpha(v) > \beta(v) + \gamma(v),$$

then there is no

$$\mathcal{A}' \supseteq \mathcal{A}, \quad \mathcal{B}' \supseteq \mathcal{B}$$

such that $\{\mathcal{A}', \mathcal{B}'\}$ is a satisfactory partition of $\mathcal{G}$.

Proof. This follows directly from the definition of satisfactory partition. Note that the premise of this lemma corresponds to the conditions of lines 14 and 15 in Algorithm 1. Likewise, the consequence of this lemma agrees with the action of Algorithm 1 as declared in line 15, where the algorithm returns to the previous instance of the procedure Σ, which holds the previous copy of an under-construction satisfactory partition. □

Now, we underline the correctness of line 21, where Algorithm 1 returns to a previous stage of the search process when some conditions hold, as stated in the following lemma:

Lemma 8. *For any graph $\mathcal{G}$, throughout the execution of Algorithm 1, if there is a vertex $v \in \mathcal{B}$ (i.e., $\mu(v) = 2$) such that there exists $u \sim v$ with $u \in \mathcal{A}$ (i.e., $\mu(u) = 1$) and*

$$\beta(u) > \alpha(u) + \gamma(u),$$

then there is no

$$\mathcal{A}' \supseteq \mathcal{A}, \quad \mathcal{B}' \supseteq \mathcal{B}$$

such that $\{\mathcal{A}', \mathcal{B}'\}$ is a satisfactory partition of $\mathcal{G}$.

Proof. This lemma is in line with the definition of satisfactory partition. Observe that the premise of this lemma corresponds to the condition of lines 14, 19, and 21 in Algorithm 1. Further, the consequence of this lemma is consistent with the action of Algorithm 1 as stated in line 21, where the algorithm returns to the previous instance of the procedure Σ that holds the previous copy of an under-construction satisfactory partition. □

Next, we prove the correctness of lines 11–13 and 16–18, where Algorithm 1 decides to assign part $\mathcal{B}$ for some vertex provided that some conditions hold, as demonstrated by the following lemma:

Lemma 9. *For any graph $\mathcal{G}$, at any stage of the execution of Algorithm 1, if there is $x \in \mathcal{B}$ (i.e., $\mu(x) = 2$) such that*

$$\beta(x) + \gamma(x) - 1 \leq \alpha(x) \leq \beta(x) + \gamma(x)$$

and, there is a satisfactory partition, $\{\mathcal{A}', \mathcal{B}'\}$, of $\mathcal{G}$ such that $\mathcal{A}' \supseteq \mathcal{A}$ with $\mathcal{B}' \supseteq \mathcal{B}$, then for every $y \sim x$ with $y \notin \mathcal{A} \cup \mathcal{B}$, $y \in \mathcal{B}'$.

Proof. The premise of this lemma corresponds to lines 11, 14, and 16. The consequence of this lemma is guaranteed by Algorithm 1 according to the actions in lines 12, 13, 17, and 18. However, let us show the correctness of these actions (i.e., this lemma) by contradiction. Suppose there is $y \sim x$ with $y \notin \mathcal{A} \cup \mathcal{B}$ such that $y \notin \mathcal{B}'$. This means $y \in \mathcal{A}' \setminus \mathcal{A}$. Hence, this requires

$$\beta(x) + \gamma(x) - 2 \leq \alpha(x) + 1 \leq \beta(x) + \gamma(x) - 1.$$

Therefore, it holds that

$$\beta(x) + \gamma(x) - 3 \leq \alpha(x) \leq \beta(x) + \gamma(x) - 2.$$

Consequently,

$$\alpha(x) = \beta(x) + \gamma(x) - 2$$

or

$$\alpha(x) = \beta(x) + \gamma(x) - 3.$$

This contradicts the inequality given in the premise of this lemma

$$\beta(x) + \gamma(x) - 1 \leq \alpha(x) \leq \beta(x) + \gamma(x),$$

that means

$$\alpha(x) = \beta(x) + \gamma(x) - 1$$

or

$$\alpha(x) = \beta(x) + \gamma(x).$$

This completes the proof of this lemma. □

Now, we aim to show the correctness of lines 5–7 and 22–24, where Algorithm 1 decides to assign part $\mathcal{A}$ for some vertex provided that some conditions hold, as we elaborate in the following lemma:

Lemma 10. *For any graph $\mathcal{G}$, at any stage of the execution of Algorithm 1, if there is $x \in \mathcal{A}$ (i.e., $\mu(x) = 1$) such that*

$$\alpha(x) + \gamma(x) - 1 \leq \beta(x) \leq \alpha(x) + \gamma(x)$$

and if there is a satisfactory partition, $\{\mathcal{A}', \mathcal{B}'\}$, of $\mathcal{G}$ with $\mathcal{A}' \supseteq \mathcal{A}$ and $\mathcal{B}' \supseteq \mathcal{B}$, then for every $y \sim x$ with $y \notin \mathcal{A} \cup \mathcal{B}$, $y \in \mathcal{A}'$.

Proof. The premise of this lemma corresponds to lines 3, 5, and 22. The consequence of this lemma is guaranteed by Algorithm 1 according to the actions in lines 6, 7, 23, and 24. However, let us show the correctness of these actions (i.e., this lemma) by contradiction. Suppose there is $y \sim x$ with $y \notin \mathcal{A} \cup \mathcal{B}$ such that $y \notin \mathcal{A}'$. Thus, $y \in \mathcal{B}' \setminus \mathcal{B}$. This requires that

$$\alpha(x) + \gamma(x) - 2 \leq \beta(x) + 1 \leq \alpha(x) + \gamma(x) - 1.$$

Therefore, it holds that

$$\alpha(x) + \gamma(x) - 3 \leq \beta(x) \leq \alpha(x) + \gamma(x) - 2.$$

Consequently,

$$\beta(x) = \alpha(x) + \gamma(x) - 2$$

or

$$\beta(x) = \alpha(x) + \gamma(x) - 3.$$

This contradicts the inequality given in the premise of this lemma

$$\alpha(x) + \gamma(x) - 1 \leq \beta(x) \leq \alpha(x) + \gamma(x),$$

that implies

$$\beta(x) = \alpha(x) + \gamma(x) - 1$$

or

$$\beta(x) = \alpha(x) + \gamma(x).$$

This completes the proof of this lemma. □

In the following, we show the correctness of lines 25–27 in Algorithm 1.

Lemma 11. *For any graph $\mathcal{G}$, if line 27 of Algorithm 1 is executed, then the reported set*

$$\{\{x \mid \mu(x) = 1\}, \quad \{x \mid \mu(x) = 2\}\},$$

which denotes the set $\{\mathcal{A}, \mathcal{B}\}$, is truly a satisfaction partition of the inputted graph $\mathcal{G}$.

Proof. We need to show that the set $\{\mathcal{A}, \mathcal{B}\}$ reported by the algorithm in line 27 satisfies the definition of satisfactory partition. Thus, we first show that

$$\mathcal{A} \neq \emptyset \quad \text{and} \quad \mathcal{B} \neq \emptyset,$$

which is guaranteed by line 26. Likewise, we need to show that

$$\mathcal{A} \cap \mathcal{B} = \emptyset.$$

This is ensured using our total mapping μ throughout the algorithm. Additionally, we need to show that

$$\mathcal{A} \cup \mathcal{B} = \mathcal{V},$$

which is certain by the actions of the algorithms that whenever a vertex joins part $\mathcal{A}$ or part $\mathcal{B}$, v is removed from the vertex set $\mathcal{V}$, see lines 7, 13, 18, and 24. Further, the algorithm reports $\{\mathcal{A}, \mathcal{B}\}$ being a partition of the inputted graph if and only if $\mathcal{V} = \emptyset$; see line 25. Equally, we need to show that

$$\forall x \in \mathcal{A},\ \alpha(x) \geq \beta(x),$$

which is warranted by lines 4 and 21; note that whenever line 27 is executed, it is the case that $\gamma(x) = 0$ for all vertices x. Lastly, we need to prove that

$$\forall y \in \mathcal{B},\ \beta(y) \geq \alpha(y),$$

which is maintained by lines 10 and 15. □

In our last theorem, we stress that Algorithm 1 is correct.

Theorem 4. *Algorithm 1 finds a satisfactory partition (if any exists) of a given undirected graph.*

Proof. Lemmas 1–11 altogether show this claim. □

6. Discussion

Literature demonstrating applications of "satisfactory partitions" is scarce. Hence, a natural direction to extend this research line in the future is to conduct case studies in domains that can benefit from the "satisfactory partitions" notion. Nonetheless, identifying a satisfactory partition of a graph may play a vital role in understanding the social structure of communities. Take, for example, a community of voters; one might be interested in discovering networks of electors or coalitions that are internally cohesive. Likewise, consider a human management case where the company aims to create two teams to implement different projects. The company desires teams with minimal conflict, minimal communication across teams, and maximal collaboration within each team. Represent employees as vertices and put links between the vertices to denote interactions between the employees. A satisfactory partition of the resulting graph will be an optimal solution to the team formation problem. However, further work might examine diverse potential applications in other specialized domains, such as studying fullerenes (see, e.g., [32, 33]).

Limitations of our study lie in that we calibrate the design of our algorithm (i.e., Algorithm 1) to optimize its expected recursion depth. Specifically, we incorporate the process of the "while" loop, which runs in cubic time. However, for the general worst-case, the running time of Algorithm 1 is $O(n^3 2^n)$. Suppose we drop the while loop and modify the if statement (at line 25) to check for the conditions of a satisfactory partition. In that case, the algorithm becomes literary a "generate and test" procedure that runs always (in all cases) in exponential time $O(2^n)$. Compare this to the running time of Algorithm 1 $O(n^3 2^{O(\ln n)})$ under the assumption that the recursion depth of the algorithm does not exceed the expected depth $O(\ln n)$. As discussed earlier in this article, this upper bound of the expected recursion depth is reached by approximating the sum

$$\sum_{i=1}^{n} \frac{1}{i}.$$

In fact,

$$\ln n + \frac{1}{n} \leq \sum_{i=1}^{n} \frac{1}{i} \leq \ln n + 1.$$

This means if the recursion depth of Algorithm 1 does not exceed the expectation, then the running time of Algorithm 1 is in

$$O(n^3 2^{\ln n}) \in O(n^3 n^{\ln 2}) \in O(n^{3.7})$$

Therefore, there is an obvious trade-off between achieving a logarithmic expected depth of the recursion of the algorithm versus an all-case exponential time of $O(2^n)$. Future research might investigate whether the amortized running time of the while loop is better than cubic time to mitigate this trade-off.

7. Conclusions

We studied an algorithm for computing a satisfactory partition of an undirected graph and showed that for a given undirected graph with n vertices, a satisfactory partition (if any exists) can be computed recursively with a recursion tree of depth $O(\ln n)$ in expectation. Likewise, we showed that a satisfactory partition for those undirected graphs, with recursion tree depth meeting the expectation, can be computed in time $O(n^3 2^{O(\ln n)})$. However, in the general case, we note that the algorithm runs in time $O(n^3 2^n)$, as it is known that the problem of computing a satisfactory partition is NP-complete.

Conflict of interest

The author declares no conflict of interest.

References

1. M. U. Gerber, D. Kobler, Algorithmic approach to the satisfactory graph partitioning problem, *Eur. J. Oper. Res.*, **125** (2000), 283–291. https://doi.org/10.1016/S0377-2217(99)00459-2
2. M. U. Gerber, D. Kobler, Algorithms for vertex-partitioning problems on graphs with fixed clique-width, *Theor. Comput. Sci.*, **299** (2003), 719–734. https://doi.org/10.1016/S0304-3975(02)00725-9
3. C. Bazgan, Z. Tuza, D. Vanderpooten, On the existence and determination of satisfactory partitions in a graph, In: T. Ibaraki, N. Katoh, H. Ono, *Algorithms and computation*, Springer-Verlag, 2003. https://doi.org/10.1007/978-3-540-24587-2_46
4. C. Bazgan, Z. Tuza, D. Vanderpooten, The satisfactory partition problem, *Discrete Appl. Math.*, **154** (2006), 1236–1245. https://doi.org/10.1016/j.dam.2005.10.014
5. C. Bazgan, Z. Tuza, D. Vanderpooten, Complexity and approximation of satisfactory partition problems, In: L. Wang, *Computing and combinatorics*, Springer-Verlag, 2005. https://doi.org/10.1007/11533719_84
6. A. Gaikwad, S. Maity, S. K. Tripathi, Parameterized complexity of satisfactory partition problem, *Theor. Comput. Sci.*, **907** (2022), 113–127. https://doi.org/10.1016/j.tcs.2022.01.022

7. N. Kim, Z. Zheng, S. Elmetwaly, T. Schlick, Rna graph partitioning for the discovery of rna modularity: a novel application of graph partition algorithm to biology, *Plos One*, **9** (2014), e106074. https://doi.org/10.1371/journal.pone.0106074
8. L. Jäntschi, M. V. Diudea, Subgraphs of pair vertices, *J. Math. Chem.*, **45** (2009), 364–371. https://doi.org/10.1007/s10910-008-9411-6
9. C. Guada, E. Zarrazola, J. Yáñez, J. T. Rodríguez, D. Gómez, J. Montero, A novel edge detection algorithm based on a hierarchical graph-partition approach, *J. Intell. Fuzzy Syst.*, **34** (2018), 1875–1892. https://doi.org/10.3233/JIFS-171218
10. M. Li, H. Cui, C. Zhou, S. Xu, Gap: genetic algorithm based large-scale graph partition in heterogeneous cluster, *IEEE Access*, **8** (2020), 144197–144204. https://doi.org/10.1109/ACCESS.2020.3014351
11. H. Cui, D. Yang, C. Zhou, A large-scale graph partition algorithm with redundant multi-order neighbor vertex storage, *Inf. Sci.*, **667** (2024), 120473. https://doi.org/10.1016/j.ins.2024.120473
12. H. Li, R. Fu, X. Ma, Forbidden subgraphs in reduced power graphs of finite groups, *AIMS Math.*, **6** (2021), 5410–5420. https://doi.org/10.3934/math.2021319
13. S. Fidanova, P. C. Pop, An improved hybrid ant-local search algorithm for the partition graph coloring problem, *J. Comput. Appl. Math.*, **293** (2016), 55–61. https://doi.org/10.1016/j.cam.2015.04.030
14. S. Fidanova, P. C. Pop, An ant algorithm for the partition graph coloring problem, In: I. Dimov, S. Fidanova, I. Lirkov, *Numerical methods and applications*, Springer-Verlag, 2014. https://doi.org/10.1007/978-3-319-15585-2_9
15. S. Zhang, Z. Jiang, X. Hou, Z. Guan, M. Yuan, H. You, An efficient and balanced graph partition algorithm for the subgraph-centric programming model on large-scale power-law graphs, *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021. https://doi.org/10.1109/ICDCS51616.2021.00016
16. L. Chen, Y. Chen, Y. Wang, An improved spectral graph partition intelligent clustering algorithm for low-power wireless networks, *J. Ambient Intell. Humanized Comput.*, 2019. https://doi.org/10.1007/s12652-019-01508-7
17. Y. Leng, H. Wang, F. Lu, Artificial intelligence knowledge graph for dynamic networks: an incremental partition algorithm, *IEEE Access*, **8** (2020), 63434–63442. https://doi.org/10.1109/ACCESS.2020.2982652
18. X. Heng, Y. Chen, L. Liu, Medical intelligent system and orthopedic clinical nursing based on graph partition sampling algorithm, *Comput. Intell. Neurosci.*, **2022** (2022), 2764157. https://doi.org/10.1155/2022/2764157
19. L. Wang, S. Ding, Y. Wang, L. Ding, A robust spectral clustering algorithm based on grid-partition and decision-graph, *Int. J. Mach. Learn. Cybern.*, **12** (2021), 1243–1254. https://doi.org/10.1007/s13042-020-01231-2
20. J. Wang, Y. Guo, X. Wen, Z. Wang, Z. Li, M. Tang, Improving graph-based label propagation algorithm with group partition for fraud detection, *Appl. Intell.*, **50** (2020), 3291–3300. https://doi.org/10.1007/s10489-020-01724-1

21. M. Fu, Y. Zhang, Results on monochromatic vertex disconnection of graphs, *AIMS Math.*, **8** (2023), 13219–13240. https://doi.org/10.3934/math.2023668

22. W. Zhou, H. Tang, Z. Ji, A task partition algorithm based on grid and graph partition for distributed crowd simulation, *2014 Fourth International Conference on Instrumentation and Measurement, Computer, Communication and Control*, 2014. https://doi.org/10.1109/IMCCC.2014.113

23. J. W. Zhan, A novel sports video background segmentation algorithm based on graph partition, *2015 8th International Conference on Intelligent Computation Technology and Automation (ICICTA)*, 2015. https://doi.org/10.1109/ICICTA.2015.25

24. H. Cui, Y. Wu, S. Lv, Property graph partition algorithm based on improved barnacle mating optimization, *J. Phys.*, **2832** (2024), 012005. https://doi.org/10.1088/1742-6596/2832/1/012005

25. Y. Chen, Q. Wang, X. Cai, N. Wang, A new text mining method of dispatching operation ticket system based on graph partition spectral clustering algorithm, *2023 6th International Conference on Energy, Electrical and Power Engineering (CEEPE)*, 2023, 1517–1521. https://doi.org/10.1109/CEEPE58418.2023.10166981

26. B. Ma, C. Yang, Distinguishing colorings of graphs and their subgraphs, *AIMS Math.*, **8** (2023), 26561–26573. https://doi.org/10.3934/math.20231357

27. S. Luo, L. Liu, H. Wang, B. Wu, Y. Liu, Implementation of a parallel graph partition algorithm to speed up bsp computing, *2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2014. https://doi.org/10.1109/FSKD.2014.6980928

28. P. C. Pop, B. Hu, G. R. Raidl, A memetic algorithm with two distinct solution representations for the partition graph coloring problem, In: R. M. Díaz, F. Pichler, A. Q. Arencibia, *Computer aided systems theory-EUROCAST*, Springer-Verlag, 2013. https://doi.org/10.1007/978-3-642-53856-8_28

29. L. Jäntschi, S. D. Bolboacă, Informational entropy of b-ary trees after a vertex cut, *Entropy*, **10** (2008), 576–588. https://doi.org/10.3390/e10040576

30. W. Zhao, Y. Li, R. Lin, The existence of a graph whose vertex set can be partitioned into a fixed number of strong domination-critical vertex-sets, *AIMS Math.*, **9** (2024), 1926–1938. https://doi.org/10.3934/math.2024095

31. J. Gómez-Gardeñes, E. Estrada, Network bipartitioning in the anti-communicability euclidean space, *AIMS Math.*, **6** (2021), 1153–1174. https://doi.org/10.3934/math.2021070

32. S. D. Bolboacă, L. Jäntschi, Nanoquantitative structure-property relationship modeling on c_{42} fullerene isomers, *J. Chem.*, **2016** (2016), 1791756. https://doi.org/10.1155/2016/1791756

33. D. M. Joiţa, L. Jäntschi, Extending the characteristic polynomial for characterization of c_{20} fullerene congeners, *Mathematics*, **5** (2017), 84. https://doi.org/10.3390/math5040084