GPART: Partitioning Maximal Redundant Rigid and Maximal Global Rigid Components in Generic Distance Graphs

YU ZHANG, QINHAN WEI, YONGCAI WANG, HAODI PING, and DEYING LI, Renmin University of China

Partitioning the *Maximal Redundant Rigid Components (MRRC)* and *Maximal Global Rigid Components (MGRC)* in generic 2D graphs are critical problem for network structure analysis, network localizability detection, and localization algorithm design. This article presents efficient algorithms to partition MRRCs and MGRCs and develops an open-sourced toolbox, GPART, for these algorithms to be conveniently used by the society. We firstly propose conditions and an efficient algorithm to merge the over-constrained regions to form the maximal redundant rigid components (MRRC). The detected MRRCs are proved to be maximal and all the MRRCs are guaranteed to be detected. The time to merge the over-constrained regions is linear to the number of nodes in the over-constrained components. To detect MGRCs, the critical problem is to decompose 3-connected components in each MRRC. We exploit SPQR-tree based method and design a local optimization algorithm, called MGRC_acce to prune the unnecessary decomposition operations so that the SPQR-tree functions can be called much less number of times. We prove the MGRCs can be detected inside MRRCs using at most $O(mn)$ time. Then a GPART toolbox is developed and extensively tested in graphs of different densities. We show the proposed MRRC and MGRC detection algorithms are valid and MGRC_acce greatly outperforms the direct SPQR-tree based decomposition algorithm. GPART is outsourced at https://github.com/inlab-group/gpart.

CCS Concepts: • **Networks** → **Network algorithms**; • **Software and its engineering** → *Software notations and tools*; • **Theory of computation** → *Design and analysis of algorithms;*

Additional Key Words and Phrases: Graph rigidity, component partition, graph algorithm, generic graph

ACM Reference format:
Yu Zhang, Qinhan Wei, Yongcai Wang, Haodi Ping, and Deying Li. 2023. GPART: Partitioning Maximal Redundant Rigid and Maximal Global Rigid Components in Generic Distance Graphs. *ACM Trans. Sensor Netw.* 19, 4, Article 86 (June 2023), 26 pages.
https://doi.org/10.1145/3594668

1 INTRODUCTION

Finding rigid, redundant rigid, and global rigid components in graphs is a crucial graph structure analysis problem, which is the foundation for a variety of structure inference and control

This work was supported in part by the National Natural Science Foundation of China Grant No. 61972404, 12071478, Public Computing Cloud, Renmin University of China.
Authors' address: Y. Zhang, Q. Wei, Y. Wang (corresponding author), H. Ping, and D. Li, Renmin University of China, Beijing 100872, China; emails: {zyyx2020, qhwei2021, ycw,haodi.ping, deyingli}@ruc.edu.cn.

1550-4859/2023/06-ART86 $15.00
https://doi.org/10.1145/3594668

applications, such as network localization in the **Internet of Things** (**IoT**) or formation control in **Unmanned Aerial Vehicle** (**UAV**) networks [11, 21, 25]. For example, finding the global rigid component is the foundation to identify the localizable sub-networks, i.e., the components that can be uniquely localized in the network [23]. Because applications generally require identifying the maximum number of nodes that satisfy the specific rigid conditions [12, 16, 23, 24], pursuing the maximal rigid, maximal redundant rigid, and **maximal global rigid components** (**MGRC**) is of great importance. Here, "maximal" means that the component will lose its desired level of the rigid property when any other neighboring node is added to this component.

Let $G = \{V, E, \mathbf{D}\}$ represent a distance graph in $\Re^d$, where d is the dimension of space; $\mathbf{D}$ is the partially measured distance matrix. If $(i, j) \in E$, it means the distance between nodes i, j, i.e., $d_{i,j}$ is measured in $\mathbf{D}$. The inter-node distance is assumed symmetric. Algorithms for graph rigidity analysis are mostly developed for two-dimensional systems, which is also the focus of this article.

For a graph G, it is said *generically rigid* in $\Re^2$, if "except the trivial rigid motions, one cannot continuously deform the graph's realization in $\Re^2$ while preserving all the inter-node distances constraints" [10]. Here, "trivial rigid motions" include transition, rotation, and reflection; The word "generically" means the nodes' coordinates are algebraically independent over the rationals, which is widely assumed in rigidity analysis [2]. A graph is generically globally rigid (globally rigid) if it has a unique realization in the 2D plane [4]. Jackson et al., [8] prove that a graph is globally rigid if and only if it is both 3-connected and redundantly rigid. The term "redundantly rigid" means the graph remains rigid after the removal of any single edge from the graph.

When the whole graph is not global rigid, redundant rigid, or even not rigid, finding subgraphs satisfying above different rigid conditions is the crucial problem to investigating the inner structure of the graph. More importantly, researchers want to partition the **maximal rigid components** (**MRCs**), **maximal redundant rigid components** (**MRRCs**), and MGRC in the graph. Note that a component is said "maximal rigid" if its "rigid" property will be broken by adding any other neighboring node into the component. The maximal redundant rigid component and the maximal global rigid component are defined in the same way.

From the application perspective, finding MRRCs and MGRCs also has great application values. Finding MGRCs is the foundation for the crucial problem of identifying localizable nodes [11]. In recently proposed component-based graph realization algorithms [13, 19], these algorithms desire to extract all the MGRCs at first so that these reliable components can be realized in priority to avoid the flexibility impacts of the sparse components.

Because of above great theoretical and application values, finding the subgraphs satisfying different kinds of rigidity properties has also attracted great research attention. Early results can be categorized into theoretical and algorithmic two categories:

(1) Early theoretical results focused on the conditions for identifying whether the whole graph is rigid, redundant rigid, or global rigid. In $\Re^d, d \leq 2$, the combinatorial necessary conditions are proposed for identifying rigid and redundant rigid graphs [4]. Later, for identifying generic global rigid graphs, Jackson and Jordan [8] state that a graph is generically globally rigid in the plane if and only if it is 3-connected and redundantly rigid. Servatius et al., [17] further present theories that in a redundantly rigid graph, the leaves of its 3-connected tree are globally rigid. But the combinatorial (connectivity and edge counting based) necessary and sufficient condition for identifying rigid and global rigid graphs in $\Re^d$, $d \geq 3$ is still open. This article therefore also focuses on efficient algorithms in $\Re^2$.
(2) Despite of above theoretical works, from algorithm perspective, PebbleGame [9] is a fundamental algorithm, which finds the MRCs in generic 2D graphs by using an efficient pebble searching algorithm. PebbleGame can also find redundant rigid components, which they

call *ORs* in their paper. All these works can be done in $O(mn)$ time where n is the number of nodes and m is the number of edges. But PebbleGame does not output the MRRCs. To the best of our knowledge, even in $\mathfrak{R}^2$, the efficient algorithms to extract the MRRCs and MGRCs from generic graphs are still open. An efficient and handy toolbox to carry out the above fundamental rigidity-based partition tasks is highly desired.

This article first presents conditions and algorithms to merge the ORs detected by PebbleGame to form MRRCs. If MRRCs can be successfully partitioned, an intuitive way to find MGRCs is to find **3-connected components (3CC)** in each MRRC. The 3-connected component partition problem can be conducted by SPQR-tree algorithm [3], which detects all binary vertex cuts in the graph in $O(m + n)$ time. But partitioning the cutting vertices found by SPQR-tree in MRRC may generate new binary vertex cuts in the decomposed components, so the SPQR-tree algorithm needs to be called recursively in the partitioned components, which is not efficient. We present an efficient local optimization algorithm to avoid recursive calls of the SPQR-tree algorithm.

The key contributions and novelties of this article are as follows.

(1) A pipeline to find the MRC, MRRC, 3CC, and MGRC in 2D generic graphs is presented.
(2) Conditions and an efficient algorithm of merging ORs returned by PebbleGame are presented. The algorithm is proven to detect all the MRRCs. We show the complexity of component merging is linear to the number of nodes in the components, so the complexity of MRRC partition is $O(mn)$, which is the same as the PebbleGame for MRC partition.
(3) An efficient algorithm exploiting SPQR-tree partition to decompose *3CC* from the MRRCs is proposed to extract the MGRCs. A local optimization method is proposed so the SPQR-tree algorithm needs to be called only a small number of times and all the MGRCs can be detected efficiently from an MRRC. The performances of the local optimization are verified in networks of different densities, which shows it contributes the most to the sparse networks.
(4) Above algorithms are aggregated into a handy toolbox called GPART for rigidity-based graph decomposition. GPART is outsourced at https://github.com/inlab-group/gpart. Extensive experiments in different kinds of network settings are conducted, which demonstrate the characteristics of MRRCs and MGRCs and verify the validity and effectiveness of GPART.

2 BACKGROUND AND PRELIMINARIES

We assume $G = (V, E, \mathbf{D})$ is a graph without multiple edges and self-loops, and its point set and edge set are V and E, respectively. $\mathbf{D}$ is the partially measured distance set. Let's suppose $|V| = n$ and $|E| = m$.

Definition 1 (Realization). If $\mathbf{p} = \{p_1, p_2, \ldots, p_n\} \in \mathfrak{R}^{2\times n}$ is a mapping from V to $\mathfrak{R}^2$ while satisfying all the edge constraints in $\mathbf{D}$, $\mathbf{p}$ is called a *realization* of G in $\mathfrak{R}^2$.

Note that we also assume G is generic, that the vertex coordinates are algebraic independent over rationals [1].

2.1 Maximal Rigid Component (MRC)

Rigidity is defined to judge whether the shape of the graph can be deformed continuously while respecting all edges' length constraints. Assuming $\mathbf{p}$ is a realization of G, and let $p(i)$ and $p(j)$ represent the coordinates of node i and j. Let v_i and v_j represent the instantaneous velocities of i and j in a motion. If (1) holds for every $(i, j) \in E$, the motion $v(G) = (v_1, v_2, \ldots, v_n)$ is called an *infinitesimal motion* of G [7].

$$(v_i - v_j)(p(i) - p(j)) = 0. \tag{1}$$

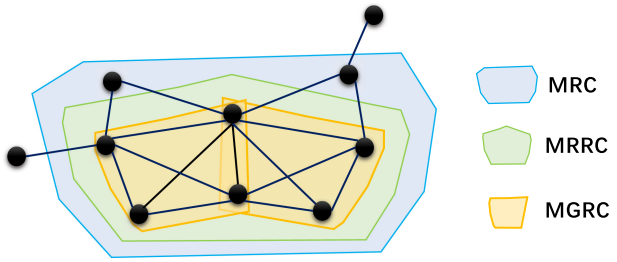


Fig. 1. An example to illustrate the MRC, MRRC, and MGRC partition.

Apparently, any translation or rotation of all vertices in G as a whole is an infinitesimal motion. These two types of motions are called *trivial flexing motions*. If a graph G contains only trivial flexing motions, we call G is *infinitesimal rigid*. Because only the generic frameworks are concerned, *infinitesimal rigid* is an equivalent concept of *rigid* [1].

Another important concept is *independent*. Consider n free points in $\Re^2$, they have $2n$ degrees of freedom in total. If an edge is added between two unconnected points, the degrees of freedom of these n points will be reduced by one. However, when degrees of freedom of points have been consumed, even if one more edge is added, the degrees of point freedom may not be decreased. The edge that cannot reduce the degree of freedom of the system is called *redundant edge*. Let F be a non-empty subset of E. If there are no redundant edges in F, then the edges in F are said *independent*.

Definition 2 (Maximal Rigid Component). A rigid component C with vertex set V_c is a *maximal rigid component (MRC)*, if $G[V_c \cup k]$ is no longer rigid by adding any neighboring node $k \notin V_c$ into C.

2.2 Maximal Redundant Rigid Component (MRRC)

A graph $G = (V, E)$ is redundant rigid if $G(V, E - e)$ is still rigid after removing any edge e from E so that we can define the concept of MRRC.

Definition 3 (Maximal Redundant Rigid Component). A redundant rigid component C with vertex set V_c is an MRRC if $G[V_c \cup k]$ is no longer redundant rigid by adding any neighboring node $k \notin V_c$ into C.

2.3 Maximal Global Rigid Component (MGRC)

Considering $\mathbf{p}$ and $\mathbf{q}$ are two realizations of G, if $\|\mathbf{p}(u) - \mathbf{p}(v)\| = \|\mathbf{q}(u) - \mathbf{q}(v)\|$ for every pair of $(u, v) \in V$, $\mathbf{p}$ and $\mathbf{q}$ are called *congruent*. If any two realizations of G are congruent, G is called *globally rigid*. It can be seen from its definition that global rigidity means that the "shape" of a graph is certain. The concept of maximal global rigid component is therefore defined.

Definition 4 (Maximal Global Rigid Component). A global rigid component C with vertex set V_c is maximal, i.e., is an MGRC, if $C[V_c \cup k]$ is no longer global rigid after adding any neighboring node $k \notin V_c$ into C.

Figure 1 gives an example, in which the MRC, MRRC, and MGRC are partitioned and highlighted in different colors. It can be seen that the MRCs, MRRCs, and MGRCs have an inclusion relationship. MRRCs are subgraphs of MRCs, and MGRCs are subgraphs of MRRCs. Note that there are two MGRCs in one MRRC.

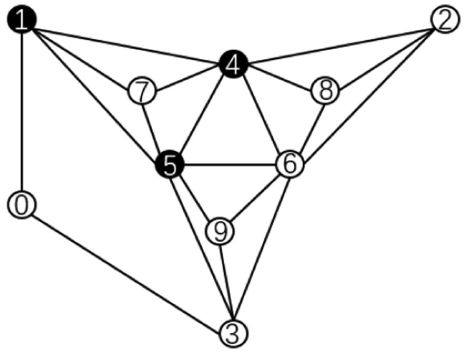


Fig. 2. An example of MRC, MRRC, and MGRCs.

2.4 An Example of Application of MRC, MRRC, and MGRC

The most typical application of MRC, MRRC, and MGRC is in the problem of network localization. One of the most important theorems in 2D network localization is that a network is localizable if and only if it is an MGRC and has at least three anchors. Figure 2 gives a model that abstracts a two-dimensional localization network into an undirected graph. The vertices in the graph represent sensor nodes that can conduct ranging to neighbors. The edge between two vertices indicates that the distance information between the two points is known after measurement. The black vertices represent the anchor points.

The whole graph of Figure 2 is rigid because it contains only trivial flexing motions. So there is only one MRC in Figure 2, which is itself. After removing vertex 0 in Figure 2, there is a graph composed of three K_4 (complete graphs with four vertices), which is an MRRC. But this MRRC is not an MGRC, because it is obvious that only knowing the world coordinates of the three anchor points 1, 4, and 5 is not enough to locate all vertices in other MRRCs. For example: by folding vertices 2 and 8 along the edge (4, 6), the result also satisfies all distance measurements. That is, there are two solutions to the world coordinates of vertices 2 and 8 and these two vertices cannot be uniquely localized. The MGRCs in Figure 2 are actually the three K_4s. So by finding the MGRCs, we know whether the current number of anchor points is sufficient, and also know which points cannot be uniquely localized. This is the value of dividing MGRCs.

2.5 Existing Detection Algorithms

PebbleGame in [9] proposed an efficient method to find MRCs. It can also find redundant rigid components. But it does not output the MRRC. The condition for being global rigid is proposed in [4], which requires the components to be both redundantly rigid and 3-connected. But it does not provide an algorithm to find the MGRCs.

In network localization society, some algorithms have been proposed to find localizable nodes. **Triangle protocols** (**TP**) [12], **Wheel extension** (**WE**) [24], **Triangle Bars** (**TB**) [16], and **Triangle Extension** (**TE**) [22] are distributed algorithms to find localizable nodes using extension-based methods. These distributed algorithms are based on sufficient trilateration or wheel graph conditions which cannot guarantee to detect all localizable nodes.

Algorithms have also been proposed for finding localizable nodes by maximum flowing algorithms. RR3P [23] requires the nodes to be inside redundantly rigid components and each node has at least three node-disjoint paths to anchors. The node-disjoint paths can be found by the maximum flow algorithm. In barycentric coordinate represented distributed linear localization, **Iterative maximum flow** (**IMF**) [14] proposes a maximum flow-based algorithm to detect localizable

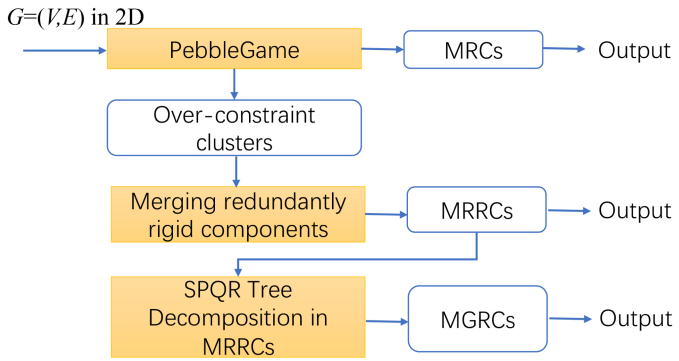


Fig. 3. The main pipeline of GPART toolkit.

Table 1. The Key Outputs of GPART and Time Complexity

Input	Output	Complexity
G^1 $(V, E, \mathbf{D}^2)$	MRCs in G	$O(mn^3)$
	MRRCs in G	$O(mn)$
	MGRCs in G	$O(mn)$
	3CCs in G	$O(m+n)$

[1]Graph G with nodes set V and edges set E.
[2]Distance matrix.
[3]n vertices and m edges.

nodes. Flipping free conditions [15] are proposed to find components without flipping ambiguities. But RR3P, IMF, and the flipping-free conditions still detect localizable nodes based on sufficient conditions. They cannot guarantee to detect all localizable nodes and cannot decompose the MGRC.

3 PARTITION METHODOLOGIES

We propose a pipeline to efficiently partition the MRRCs and MGRCs for given distance graphs and assemble the algorithms into a GPART framework. The pipeline of GPART is shown in Figure 3. Given a graph $G = (V, E)$ in $\Re^2$, GPART firstly processes the graph to detect MRCs by PebbleGame, which also detects the over-constrained regions, which are redundant rigid components. We propose conditions to merge these ORs to form the MRRCs. After that, in each MRRC, an efficient SPQR-tree algorithm is conducted to decompose an MRRC into 3CCs. We prove that the decomposed 3CCs in the MRRC must be redundantly rigid so that they are global rigid. We propose an efficient local optimization algorithm to exclude unnecessary neighboring nodes of the cutting pairs found by SPQR-tree so that the SPQR-tree algorithm needs to be run a much smaller number of times. We also prove that these decomposed global rigid components are maximal.

Table 1 summarizes the input, output, and complexity of computing each output in our developed GPART toolbox. In overall, GPART can output MRCs, MRRCs, and MGRCs for an input graph G with n vertices and m edges in $O(mn)$ time. We present these algorithms accordingly.

3.1 MRC Detection

Given $G = (V, E)$ in $\Re^2$, MRCs are detected by PebbleGame [9] through a *pebble covering* and a *labeling process*. In initialization, PebbleGame assigns two pebbles to each vertex, representing the two degrees of freedom.

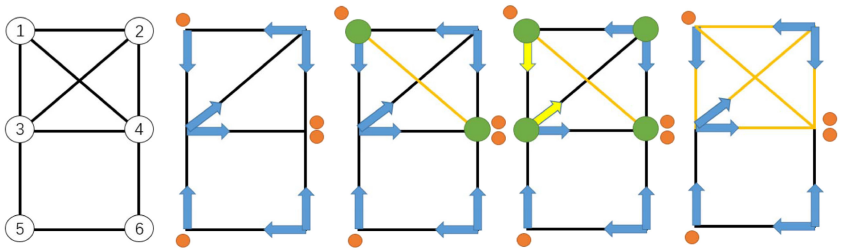


Fig. 4. An example of generating an over-constraint region.

3.1.1 Pebble Covering. The pebble covering step processes each edge to find independent edges and constructs a directed graph $H = (V, F)$.

It starts by setting $E' = E$ and takes any edge e from E'. Suppose the two vertices of e are u and v, respectively. Vertex u conducts a *seeking* operation until the number of free pebbles on u reaches two or a certain *seeking* ends in failure. The operation *seeking* starts from u and performs a depth-first search on graph $H = (V, F)$ to find a vertex w ($w \neq u, v$) with at least one free pebble. Suppose the edges on the path from u to w are $e_1, e_2, \ldots, e_k$, then the direction of $e_k, e_{k-1}, \ldots, e_1$ will be reversed in turn in F, and finally u will have one more pebble and w has one less pebble at the same time.

If both u and v successfully collect two pebbles, a new directed edge e' from u to v is added into F, and the edge e' consumes a pebble of u. Then $E' = E' - e$. If E' is not empty, the process will start over on the next unvisited edge in E until E is empty. For more details, see [9].

3.1.2 Labeling Process. Based on $H = (V, F)$ returned by pebble covering, a queue Q is created from empty. A new cluster label i is introduced, and an unmarked edge e in E is selected. Then e collects 3 pebbles from its two vertices through *pebble seeking* and the two vertices incident to e are added into Q. Then $Q.pop(x)$. Suppose the neighbors of x in G are V_x. For every $u \in V_x$, a depth-first search is started from u and performed on H to try to find a vertex with a free pebble or with a floppy mark. Then one of the following two situations must happen:

(1) If the search for free pebbles fails, all vertices passed in the depth-first search are marked as "rigid", and are added to the tail of Q.

(2) If a vertex that has free pebbles or has a floppy mark is found (let it be w), then all vertices **on the search path from u to w** are marked as floppy.

The process will be repeated until Q is empty. When Q is empty, for every $e \in E$, if both of its two vertices are marked as rigid, then e is labeled by i. All the **marks on the vertices** will be cleared, and the progress goes back to checking another unmarked edge in E until all edges in E have been marked. After completing the above process, all edges with the same label constitute a maximum rigid component.

3.1.3 Over-constrained Region. PebbleGame also finds the ORs, which are identical to the concept of redundant rigid components in this paper. In each *seeking* operation, if the fourth free pebble is not found by the depth-first search, it means the edge is redundant. Then all the vertices passed through during this *seeking* operation and the edges between these vertices constitute a redundant rigid subgraph. In PebbleGame, these redundant rigid subgraphs are called **over-constraint regions (ORs)**. Taking Figure 4 as an example, which shows the process of how the ORs are detected in PebbleGame. In Figure 4(b), all 8 edges in black have found pebble cover. In Figure 4(c), when the yellow edge wants to find a pebble cover, it seeks pebbles as shown in Figure 4(d) but fails to find a pebble cover. So all the vertices passed through during this seeking

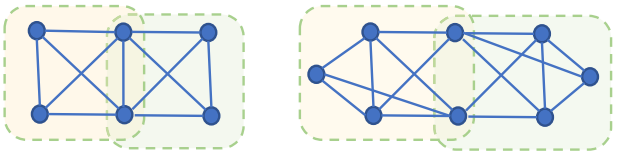


Fig. 5. Merging two redundant rigid components using Case 1 of Theorem 5.

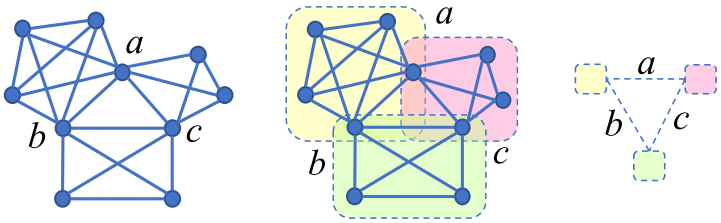


Fig. 6. Merging redundant rigid components using Case 2 of Theorem 5.

operation and the edges between them constitute a redundant rigid subgraph as shown by the subgraph with yellow edges.

Note that the ORs, i.e., redundant rigidity components detected by PebbleGame are not maximal. We propose a merging method to find all the MRRCs.

3.2 MRRC Detection

In order to get the MRRCs, we propose the following conditions for merging redundant rigid components returned by PebbleGame to form MRRCs.

Theorem 5 (Conditions to Merge Redundant Rigid Components). *Case 1: If two redundant rigid components G_1 and G_2 share more than two vertices, then $G' = G_1 \cup G_2$ is also redundant rigid.*

Case 2: If multiple redundant rigid graphs $\{G_1, G_2, \ldots, G_m\}$ each of which shares only one vertex with another, we treat each redundant rigid graph as a "shrinked vertex" and the shared vertices as edges between the shrinked vertices. This converts $\{G_1, G_2, \ldots, G_m\}$ into a new graph S. Then if S is rigid, $G' = G_1 \cup G_2 \cup \cdots \cup G_m$ is redundant rigid.

Proof. The correctness of the first case has been proved in Lemma 1 of [19]. While for the second case, all the vertices of S are redundant rigid components. The edges in S have shared vertices of redundant rigid components in the original graph. So if we remove one edge e from the merge graph G', the edge e must be removed from one of the redundant rigid components. Say it is G_i without loss of generality. Since G_i is redundant rigid, $G_i \setminus e$ must still be rigid. Since the removal of e will not affect the edges in S, so S will not be flexed. Since the removal of e will not affect other redundant rigid components, so $G' \setminus e$ must still be rigid. Since the selection of e is arbitrary, so G' is redundantly rigid. □

Figure 5 presents two examples that merge redundant rigid components using Case 1 of Theorem 5. When two redundant rigid components share more than two vertices, they can be merged to be a larger redundant rigid component.

Figure 6 gives an example that merges redundant rigid components using Case 2 of Theorem 5. In Figure 6, the regions with different colors in the figure represent redundant rigid components. We shrink each component into a vertex. The shared vertex between two redundant rigid components is converted to an edge connecting the two shrinked vertices. This converts (b) in Figure 6 to

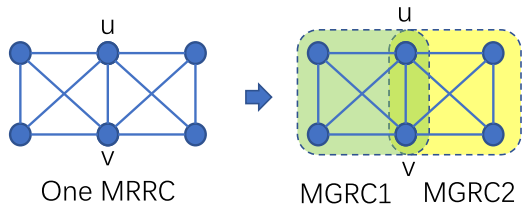


Fig. 7. An example of decomposing an MRRC to two MGRCs.

(c). The converted graph S is rigid, so the merged graph of the three redundant rigid components is redundant rigid.

Then we prove that by merging the ORs returned by PebbleGame, we can correctly find all the MRRCs in the graph.

Theorem 6. *Assuming set $N = \{N_1, N_2, \ldots N_l\}$ represents all the MRRC constructed by merging the ORs returned by PebbleGame and set $M = \{M_1, M_2, \ldots M_k\}$ represents all MRRCs in G, then there must be $N = M$.*

Proof. Convention: $E(G)$ represents the edge set of graph G, and $G(E)$ is a subgraph composed of E as the edge set and the vertices connected by the edges in E as the vertex set.

For every $M_i \in M$, suppose that $N_{j_1}, N_{j_2}, \ldots N_{j_k} (k \geq 0)$ are all MORs that satisfy $E(N_{j_t}) \cap E(M_i) \neq \emptyset (1 \leq t \leq k)$, then because the definition of 'maximum' in MRRC, there must be $E(N_{j_t}) \subseteq E(M_i)$. Let $N^* = N_{j_1} \cup N_{j_2} \cup \ldots N_{j_k}$, If $E(M_i) - E(N^*) \neq \emptyset$, according to the property of redundant rigid graph, there must be a redundant edge in $E(M_i) - E(N^*)$. This redundant edge must correspond to an over-constrained region, and this over-constrained region must also correspond to a MOR which is not equal to any one of $N_{j_1}, N_{j_2}, \ldots N_{j_k}$, this contradicts "$N_{j_1}, N_{j_2}, \ldots N_{j_k}$ are all MORs that satisfy $E(N_{j_t}) \cap E(M_i) \neq \emptyset (1 \leq t \leq k)$". So $E(M_i) - E(N^*) = \emptyset$ and k must be greater than 0. Meanwhile, because of the definition of 'maximum' in MOR, the union of any two MORs is not redundant rigid. But M_i is redundant rigid, so k must not be greater than 1. That means for each $M_i \in M$, there exists exactly one $N_j \in N$ such that $M_i = N_j$, therefore $M \in N$. Meanwhile, because $E(M_i) - E(N^*) = \emptyset$, there is not a N_j not matching to any M_i. So for each $N_j \in N$, there is exactly one $M_i \in M$ equal to N_j by previous proof. Therefore, we also prove $N \in M$, and so that $N = M$. □

So by Theorem 6, all MRRCs in G can be detected efficiently by merging the ORs returned by PebbleGame until no OR can be merged with others. We then study how to partition the MGRCs from the MRRCs.

3.3 MGRC Partition

3.3.1 Partition Methodologies. In $\mathfrak{R}^2$, a graph is global rigid if it is both redundant rigid and 3-connected [4]. Note that an MRRC must be 2-connected because it does not contain any flexing part, but it may not be 3-connected, for an example shown in Figure 7. The key idea to detect MGRCs is to decompose the MRRCs into 3CC. We will prove that the 3CC decomposed from an MRRC must still be redundant rigid. So these decomposed 3CC are both 3-connected and redundant rigid, and they are global rigid.

We first define a "Decomposition" operation by binary vertex cut in an MRRC. Suppose G is an MRRC and $\{u, v\}$ form a binary vertex cut set in G, which is also called a "separation pair". Note that a separation pair means the removal of these two vertices will disconnect the graph, as $\{u, v\}$ shown in Figure 7. If a 2-connected graph does not contain any separation pair, the graph

is 3-connected. To find MGRCs, G will be decomposed into subgraphs divided by the separation pairs.

Suppose $\{u, v\}$ is a separation pair, if $(u, v) \in E$, G is divided into G_1 and G_2 where the edge (u, v) is copied to be contained in both G_1 and G_2 and $G_1 \cap G_2 = (u, v)$. If $(u, v) \notin E$, the vertices u and v will be divided to be contained in both G_1 and G_2, so $G_1 \cap G_2 = \{u, v\}$, that they share only the two vertices. The "Decomposition" operation will be conducted recursively in G_1 and G_2, until no separation pair can be found in any decomposed subgraphs. Then all the decomposed subgraphs are 3-connected.

Theorem 7. *Suppose $G_1, G_2, \ldots, G_k$ are the 3CC decomposed from an MRRC by the recursive "Decomposition" operation, then each of these 3CC must still be redundant rigid.*

Proof. If $(u, v) \in E(G)$, by the "Decomposition" method, $G_1 \cap G_2 = (u, v)$ and $G_1 = (G \setminus G_2) \cup (u, v)$, $G_2 = (G \setminus G_1) \cup (u, v)$. After removing any edge e from G_1 except (u, v), $G_1 - e$ must still be rigid, because if it is not rigid, the same operation of $G - e$ must make $G - e$ not rigid. This contradicts with G is redundant rigid. If (u, v) is removed, since $G - (u, v)$ is rigid there must be $G_1 - (u, v)$ and $G_2 - (u, v)$ are rigid. Otherwise, $G - (u, v)$ cannot be rigid if it contains a flex subgraph. The same proof holds for G_2.

If $(u, v) \in E(G)$, by the "Decomposition" method, G_1 and G_2 share only the two vertices $\{u, v\}$. If $e \in G_1$, $G - e$ is rigid only when its subgraph $G_1 - e$ is rigid. Since $G - e$ is rigid, so $G_1 - e$ is rigid and the same holds for G_2. So after "Decomposition", G_1 and G_2 are still redundant rigid. The same process holds for further "Decomposition". So finally all the decomposed 3CC from an MRRC are redundant rigid. □

From the condition in [4], the decomposed 3CC are global rigid, since they are both 3-connected and redundant rigid.

Theorem 8. *The global rigid components decomposed from MRRCs by recursive decomposition are maximal.*

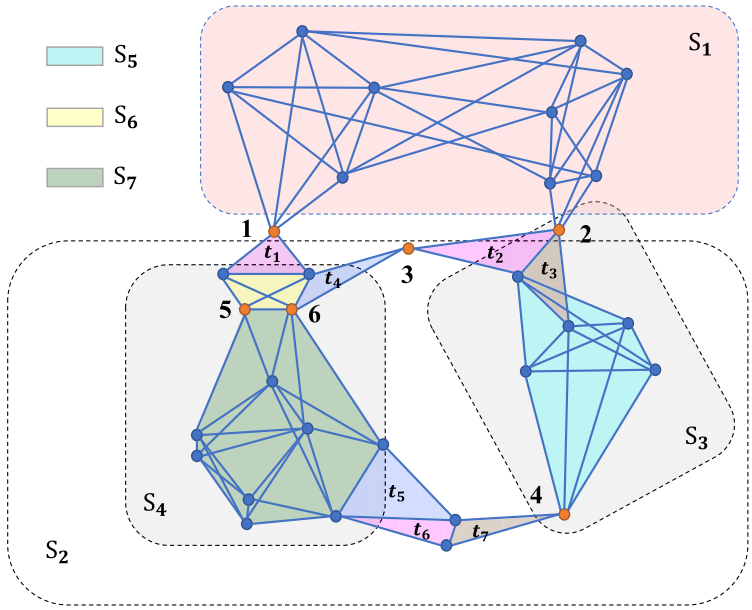
Proof. If any vertex outside such a global rigid component is added to it, the vertex is either outside the redundant rigid component or can be separated by a separation pair. The component after adding the vertex is no longer global rigid so the global rigid component is maximal. □

3.3.2 Naive Partition Algorithm. From the above discussions, the key problem is to find the separation pairs. Fortunately, all the separation pairs in an MRRC can be found efficiently by SPQR-tree algorithm [5, 6, 20] in $O(n + m)$ time. To be self-contained, the details of the SPQR-Tree algorithm for extracting 2-vertex cuts have been described in the Appendix.

We can then focus on the decomposition operations. For processing each separation pair, there are two possible cases:

(1) Case 1: If there is an edge between the two nodes of a separation pair, the "decomposition" operation will copy the edge into each decomposed component, so the decomposition will not generate new separation pairs in each decomposed component.
(2) Case 2: If there is not an edge between the two separating nodes, the decomposition may generate new separation pairs in the decomposed components. In this case, intuitively, the SPQR-tree method needs to be checked recursively in each decomposed component to check new separation pairs.

An example is shown in Figure 8. Nodes 1 and 2 are separation pairs in the graph G. By "decomposition" operation, G is decomposed into S_1 and S_2. Note that in S_2, $\{3, 4\}$ become new separation



S_i represents the component obtained by decomposition based on separation pair.
t_i represents a triangle.

Fig. 8. An example of cyclically generating new 2-vertex cuts in the process of decomposing MRRC.

pair. After decomposed by $\{3, 4\}$, node pair $\{5, 6\}$ becomes one of the new separation pairs in the decomposed component.

In Case 2 of the naive approach, the SPQR-tree algorithm needs to be called recursively in the decomposed components, until no more 2-vertex cuts can be found. The recursive call of SPQR-tree algorithm in the decomposed components of MRRC makes the MGRC partition inefficient. We present a local optimization method to accelerate the naive approach.

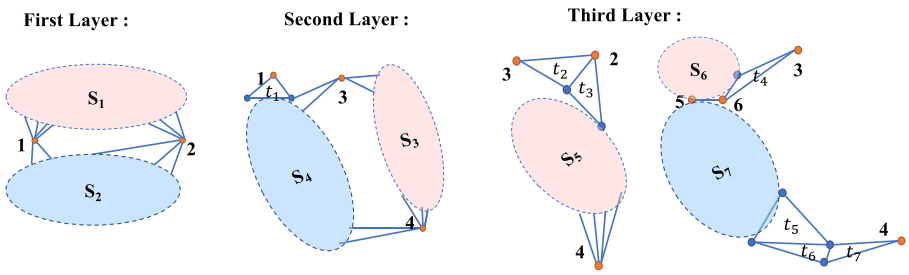
3.3.3 Local Acceleration Method. We observe that the decomposition operation in Case 2 has a special property.

Theorem 9 (Properties After Decomposition). *Suppose u, v are the two cutting vertices of Case 2. After decomposition by u, v, the decomposition will only reduce the degrees of the cutting vertices, i.e., the copies of u and v in each component, which are on the boundary of each decomposed component, and will not affect other nodes inside the components.*

We can see this property clearly from Figure 8. The separation by 1 and 2 only affects the degrees of 1 and 2 and will not affect the degrees of other nodes in S_1 and S_2.

Utilizing this property, we develop the following three local optimization methods to accelerate the decomposition algorithm to extract MGRCs from each MRRC. After the local optimization, we show that the SPQR-tree needs to be called much fewer times.

(1) Recursive deletion of nodes with degree 2 After decomposition by Case 2, if the cutting vertex u or v has degree 2, it obviously does not belong to the 3-connected component in the final MGRCs. So we can confidently delete the cutting vertex with degree 2, and recursively delete the new node with degree 2 generated by the deletion. Only the nodes with degrees equal to or higher than 3 will be retained in each decomposed component. The deletion operation is local. In Figure 8, nodes 1 and 2 in the decomposition sub-graph S_2 can be deleted.



	Computation in Layer 1	Computation in Layer 2	Computation in Layer 3
MGRC_naive	$\mathbf{P}_S + \mathbf{P}_{S_1} + \mathbf{P}_{S_2}$	$\mathbf{P}_{t_1} + \mathbf{P}_{S_3} + \mathbf{P}_{S_4}$	$\mathbf{P}_{t_2} + \mathbf{P}_{t_3} + \mathbf{P}_{t_4} + \mathbf{P}_{t_5} + \mathbf{P}_{t_6} + \mathbf{P}_{t_7} + \mathbf{P}_{S_5} + \mathbf{P}_{S_6} + \mathbf{P}_{S_7}$
MGRC_acce	$\mathbf{P}_S + \mathbf{P}_{S_1} + \mathbf{P}_{S_2}$	$\mathbf{P}_{S_3} + \mathbf{P}_{S_4}$	$\mathbf{P}_{S_5}$

Fig. 9. The running steps comparison before and after acceleration of network in Figure 8. P_i represents the operation of finding separation pairs and conducting decomposition.

(2) Components decomposed by only Case 1 do not need further judgment.
When there is an edge between the two cutting vertices, each sub-component after decomposing contains this edge, which can ensure that the interior of the sub-component maintains 3-connectivity. This is easy to deduce because all 2-vertex cuts are detected all at once during decomposition. If there are all Case 1 vertex cuts, the decomposed components preserve these edges to ensure that no new 2-vertex cuts can be generated. In addition, since no edge is deleted, the sub-components obtained by decomposition also maintain redundant rigid. Therefore, the components decomposed by only Case 1 can be output as MGRC directly.

(3) Ignore K3 (triangle) in the decomposed components.
In addition, the sub-graph obtained by MRRC through decomposition is not necessarily MRRC. For example, in Figure 8, the sub-component S_2 obtained by decomposing $\{1, 2\}$ is obviously not redundant rigid. After the decomposition and deletion of nodes with degree two, it is a triangle. The triangle is globally rigid. We skip the decomposition of K3 and directly output it when the number of remaining nodes in the component is 3 after the deletion of nodes with degree 2.

3.3.4 Naive Partition vs. Local Optimization. Let's take Figure 8 as an example to analyze the speed improvement brought by the acceleration method. Figure 8 is an MRRC as a whole. After executing SPQR-tree, $\{1, 2\}$ is detected as a separation pair. Since there is not an edge between node 1 and node 2, it follows the Case 2 in the decomposition process. We compare the number of calling SPQR-tree functions in the naive partition algorithm and that after local optimization. We denote the algorithm before using the acceleration method by *MGRC_naive* and denote the algorithm after using the acceleration method by *MGRC_acce*.

We use the letter $\mathbf{P}$ to represent the operation of calling SQPR-tree for finding separation pairs and conducting decomposition. The subscript of $\mathbf{P}$ indicates the component that performs the corresponding operations, for example, $\mathbf{P}_S$ indicates that the operation is performed in the S component. Note that this operation can also check whether a component is 3-connected. If no separation pair is returned by $\mathbf{P}_S$ then S is 3-connected.

Figure 8 shows the original graph S and its sub-modules. Figure 9 shows the comparison of the executed steps when extracting MGRCs from the example in Figure 8. We divide the decomposition step into the following three layers, corresponding to the layers in Figure 9.

ALGORITHM 1: Find Maximal Global Rigid Component

Input: An MRRC $G = (V, E)$
Output: Global rigid components, i.e., *MGRCs*

```
1  Initialize: Stack S; ;
2  Put G into S; r = −3 ;
3  while ¬S.empty do
4      Component r = S.pop() ;
5      if r.flag == −1||r.flag == −2 then
6          MGRC ← r ;
7          Continue ;
8      Find 2-vertex cuts of Case 1 or Case 2 using SPQR − tree ;
9      if there is no 2-vertex cut in r then
10         r is 3-connected component ;
11         r.flag = −1 ;
12         Continue ;
13     Separate r into M = {M_1, M_2, ..., M_n} ;
14     if M_i is decomposed by only separation pair in Case 1 then
15         M_i.flag = −2 ;
16     else
17         M_i.flag = −3 ;
18     S.push(M) ;
19 Return MGRC ;
```

(1) The first layer. At first, we perform $\mathbf{P}_S$ to obtain S_1 and S_2 according to the 2-vertex cut $\{1, 2\}$ in S. Since $\{1, 2\}$ is a separation pair in Case 2, it is difficult to determine whether there is a newly generated 2-vertex cut in S_1 and S_2. So further confirmation is required. In this layer, the operation steps of the algorithms *MGRC_naive* and *MGRC_acce* are the same, i.e., $\mathbf{P}_S, \mathbf{P}_{S_1}, \mathbf{P}_{S_2}$.
(2) The second layer. It is found that S_1 is 3-connected in $\mathbf{P}_{S_1}$, so it can be directly output as MGRC. Since there exist new 2-vertex cuts in S_2, in *MGRC_naive* algorithm, S_2 is further decomposed into three parts, t_1, S_4, and S_3, so *MGRC_naive* calls $\mathbf{P}_{t_1}$, $\mathbf{P}_{S_3}$, $\mathbf{P}_{S_4}$ to further decompose each component. In *MGRC_acce*, t_1 is deleted by optimization rule 1, so it only calls $\mathbf{P}_{S_3}, \mathbf{P}_{S_4}$.
(3) The third layer. In side S_3, and S_4, *MGRC_naive* still need further decomposition and verification in every submodule t_2 to t_7 and S_5, S_6, S_7. But *MGRC_acce* recognizes the submodules from t_2 to t_7 are triangles. They are directly output as MRGCs using the third optimization rule without calling SPQR-tree. In $\mathbf{P}_{S_4}$, $\{5, 6\}$ is a separation pair of Case 1, so S_6 and S_7 are directly output as MGRCs. In this layer, *MGRC_acce* conducts only $\mathbf{P}_{S_5}$.

Algorithm 1 presents the *MGRC_acce* algorithm which takes an MRRC as input and outputs the MGRCs. In Algorithm 1, the stack S is initialized to store the MRRC to be decomposed. After we complete the decomposition of an MRRC (line 8), the obtained sub-graph needs to be restored in S for 3-connectivity verification and further decomposition unless it is a 3-connected component marked -1 (line 9–12) or a component decomposed by only separation pair in Case 1 marked -2 (line 14–17). The 3-connectivity verification and further decomposition are mainly because the sub-graph obtained by the decomposition is not necessarily 3-connected and may exist a new 2-vertex cut generated during the decomposition.

4 PERFORMANCE ANALYSIS

The time complexity and the partition performances of GPART are presented in this section.

4.1 Time Complexity

Theorem 10 (Time Complexity of MRRC Decomposition). *The time complexity of partitioning MRRCs is $O(mn)$.*

Proof. According to the analysis of the article [18], the time complexity of implementing a pebble game in a sparse graph is $O(n^2)$, and in a dense graph (m is close to n^2) it will reach $O(mn)$, where n is the number of nodes and m is the number of edges. The time complexity of MRRC detection is to check and merge the over-constrained region labels for nodes, which is in $O(n)$. Therefore the overall complexity is at most $O(mn)$. □

Theorem 11 (Time Complexity of MGRC Decomposition). *The time complexity of partitioning MGRCs is $O(mn)$.*

Proof. For the MGRC partition, in the worst case, the entire graph is an MRRC and at every time of the decomposition, the smallest decomposed redundant rigid component contains at least two nodes and five edges. Otherwise, if the decomposed component is less than two nodes and five edges, the original graph will not be redundant rigid. So in the worst case, the numbers of decomposition operation is at most $\min\{\frac{m}{5}, \frac{n}{2}\}$. Since the time complexity of verifying 3-connectivity and finding binary vertex cuts by SPQR-tree in each decomposed component is $O(m + n)$ for the component with m edges and n nodes, the worst case overall time complexity for partitioning MGRCs is, therefore, $\min\{\frac{m}{10}(m + n), \frac{n}{4}(m + n)\}$. So the total time complexity of MGRC partition can be upper bounded by $O(mn)$. □

4.2 Visualization

We conduct simulations in Matlab2019 to visualize the MRC, MRRC, and MGRC decomposition processes. For showing the partitioned components clearly, the graph partition results for a network with 60 nodes are visually shown from Figure 10(a) to Figure 10(d). Different components are marked with lines of different colors. It can be seen that a large MRC and several small ones are detected as shown in Figure 9(b). The MRC contains only one MRRC inside it shown in Figure 10(c). Finally, the MRRC is partitioned into an MGRC with several triangle components as shown in Figure 10(d). A larger example is shown from Figure 10(e) to Figure 10(h), which contains 210 nodes. The graph is partitioned into 12 MRCs, in which more MRRCs are partitioned, and finally, a large set of MGRCs are detected.

Statistical analysis is further conducted. We firstly evaluate the edge coverage ratio by the MRCs, MRRCs, and MGRCs in different network settings, which is the number of edges in the components over the total number of edges. From sparse networks with average node degree 2, to dense networks with average node degree 9, two hundred networks are randomly generated for each node degree setting. The mean edge coverage ratio and the variance are shown in Figure 11. MRCs have much larger coverage ratio than MRRCs and MGRCs. The coverage ratio of MGRCs is only slightly lower than that of MRRCs.

The number of extracted components in each graph is an important indicator to the component size, which is statistically counted in different networks. The mean value and the variance are shown in Figure 12. In sparse networks, many small-size MRCs are detected and the number of MRRCs and MGRCs is much lower since they require stricter connectivity conditions. In dense networks, the number of components becomes much smaller, which means the size of these components has become quite large for the good connectivity of the graph.

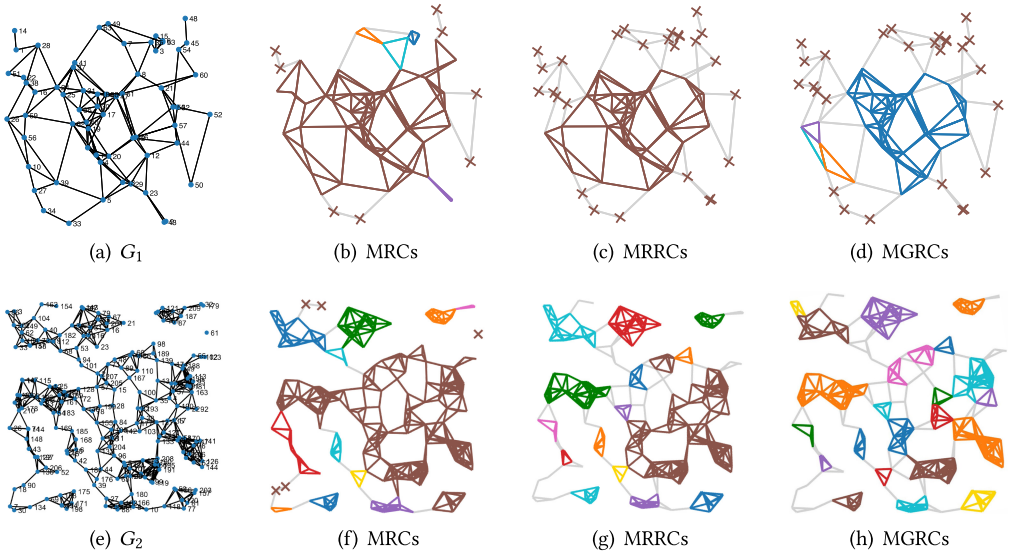


Fig. 10. Visualization of the MRC, MRRC, and MGRC detection in two graphs.

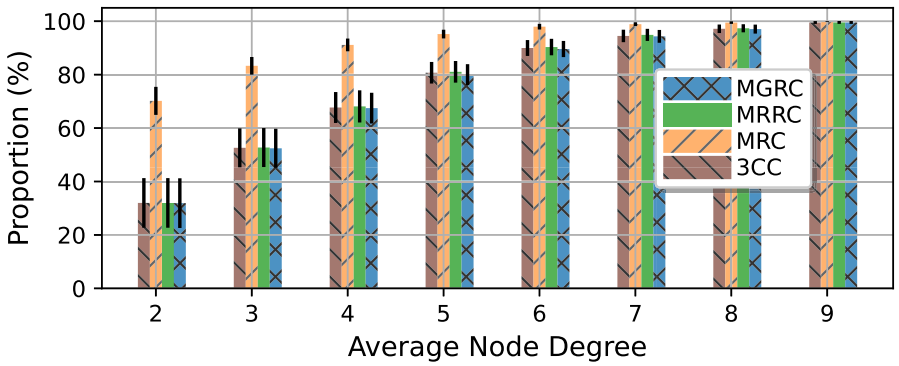


Fig. 11. Edge coverage ratios by different kinds of components.

4.3 Algorithm Efficiency

To evaluate the efficiency of the toolset in networks with different sparsity, we evaluate the running time of the proposed algorithms for networks whose node number varies from 75 to 4525 using 50 nodes as the growth step. Ten graphs with random node positions are generated at each node number and the average running time is counted and shown in Figure 13 for sparse networks with average node degrees from 2–3. Figure 14 shows the running time of different algorithms for these networks when the node degrees are from 2 to 3.

Then we increase the average node degree to about 4.5 to 5.5. Figure 15 shows the running time of different algorithms for these networks when the node degrees are from 4.5 to 5.5.

Figure 16 shows the running times of different algorithms for dense networks with degrees ranging from 8 to 9. Matlab's tic and toc functions were used to time the algorithms. In the figures, the MGRC_acce stands for accelerated MGRCs extraction algorithm while the MGRC_naive stands for unaccelerated MGRCs extraction algorithm.

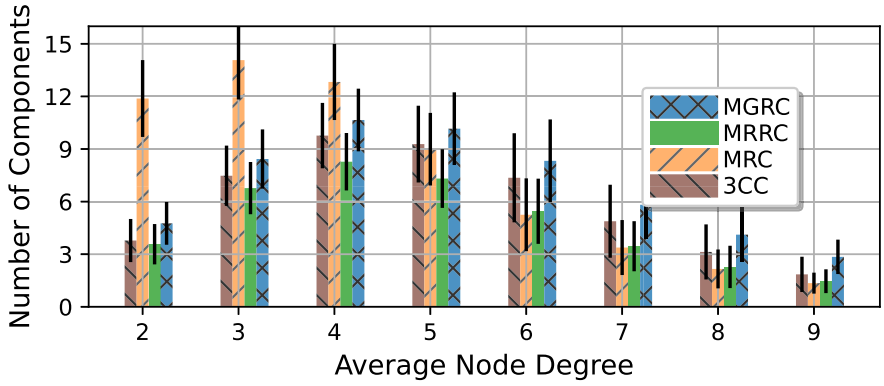


Fig. 12. Average number of components extracted in different networks.

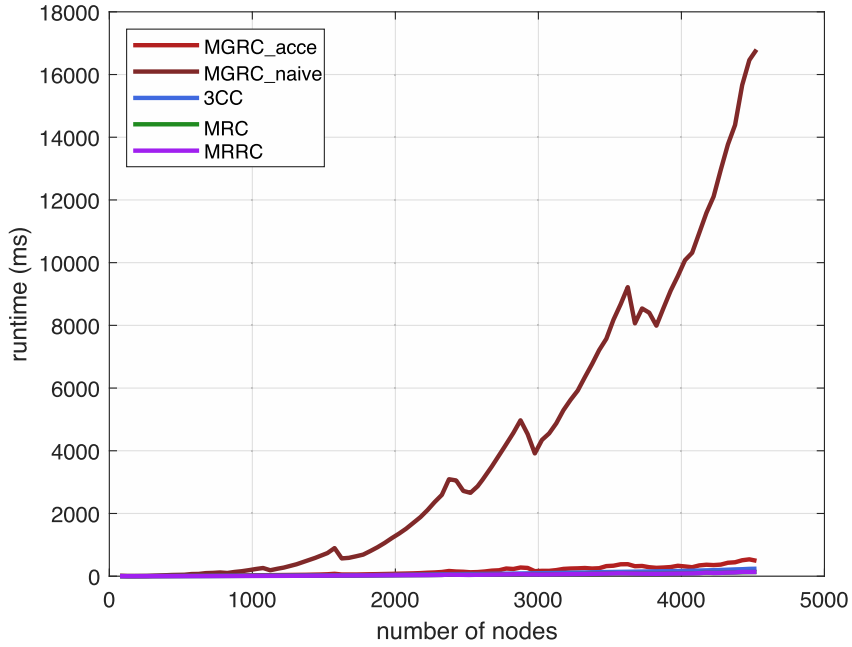
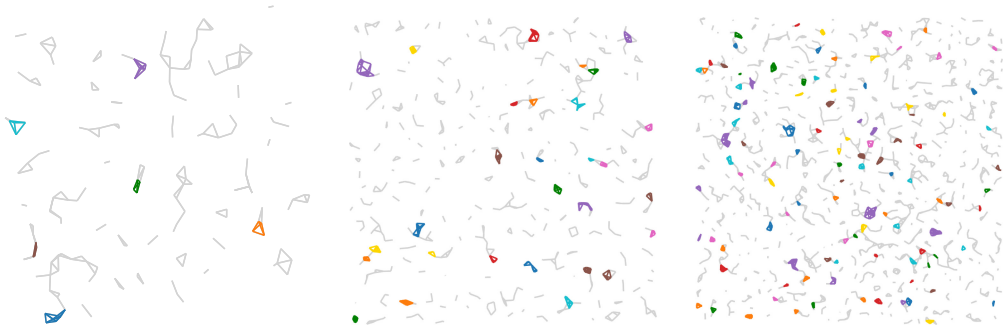


Fig. 13. The average running time as the number of network nodes increases in a network with sparsity 2–3.

Figure 13 shows the variation of the running time of each algorithm with the increase of the number of nodes when the node degree is controlled to be ranging from 2–3. It can be seen that the running time of MGRC_acce is much shortened than MGRC_naive. In order to more intuitively see the change of the partitioned MRRCs as the number of nodes increases, Figure 14 highlights the partitioned MRRCs in a set of networks with different numbers of nodes. It can be seen that in an extremely sparse network, as the number of nodes increases, the size of a single MRRC does not change much, but the overall number of MRRCs continues to increase. At the same time, due to the sparseness of the network, it is almost impossible to have separation pair of Case 2 in MRRC and there will be some triangles or non-triconnected components composed of multiple triangles in MRRC. In this case, the proposed local optimization rules help to greatly reduce the number of computation costs, to avoid calling SPQR-tree on the sparse triangles.



(a) MRRC for 225 node network (b) MRRC for 825 node network (c) MRRC for 2025 node network

Fig. 14. Visualization of the MRRCs for different node networks with degrees from 2–3.

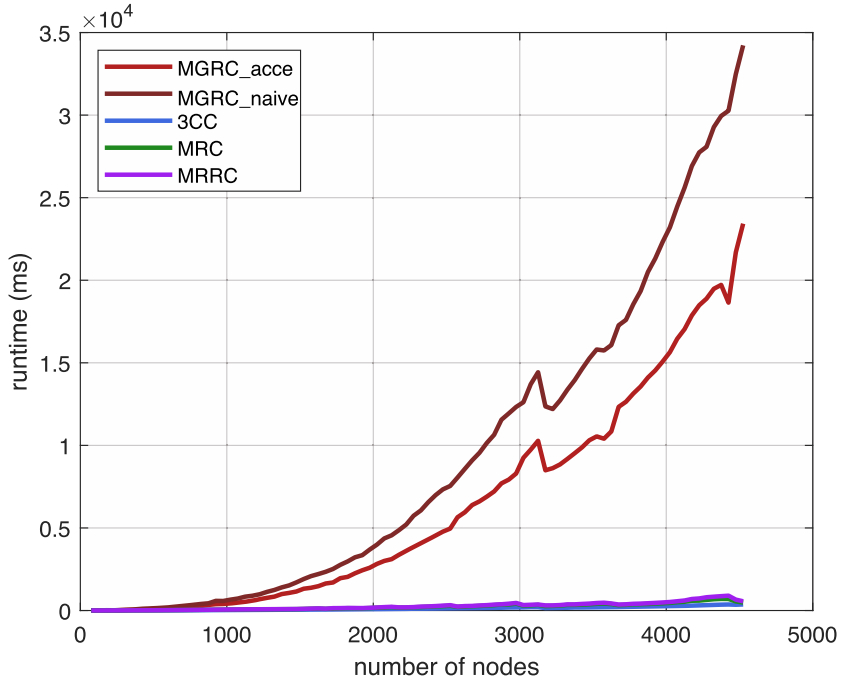


Fig. 15. The average running time as the number of network nodes increases in a network with sparsity 4.5–5.5.

In dense networks when the average node degree ranges from 8–9, in MRRCs, there is a high probability that there are no or only a few new 2-vertex cuts in the subcomponents. In such case, the accelerated MGRC_acce algorithm only needs to output directly after deleting the node with degree 2 in the subcomponent, while the MGRC_naive algorithm still needs to perform SPQR-Tree decomposition in each subcomponent to detect whether there is a new 2-vertex cut in it. This difference causes the running time difference between the two to gradually increase with the growth of the network size.

Figure 17 further gives an example to show the partitioned MRRCs for networks with different numbers of nodes while the average node degree is controlled to be ranging from 8–9. We can intuitively see the correctness of our previous discussion.

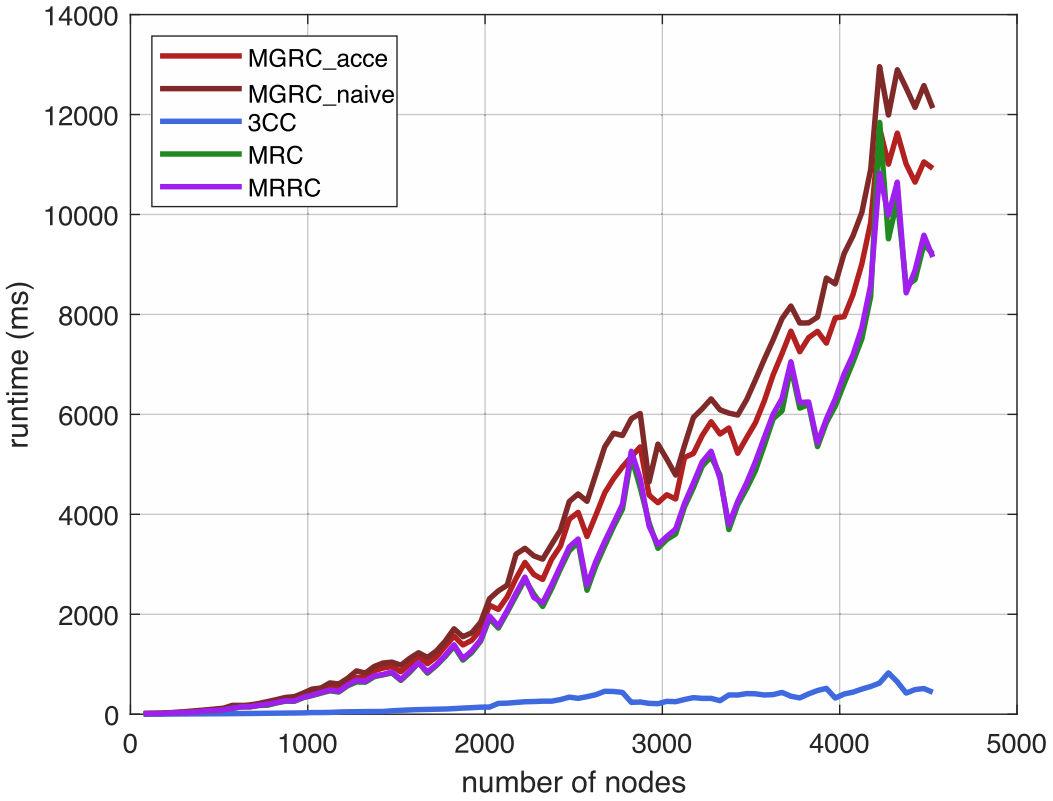
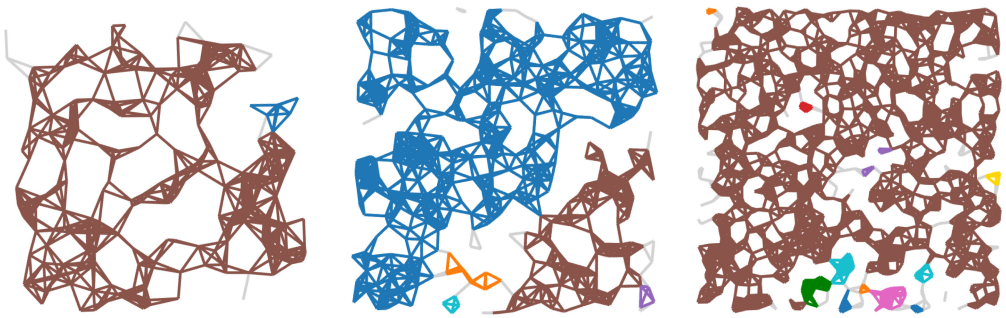


Fig. 16. The average running time as the number of network nodes increases in a network with sparsity 8–9.



(a) MRRC for 225 node network (b) MRRC for 525 node network (c) MRRC for 1425 node network

Fig. 17. Visualization of the MRRCs for different node networks with degrees from 8–9.

To evaluate the average efficiency of the toolkit, the network degree is varied from 2–12 when the network has 1,000 nodes. Fifty random graphs with a certain node degree and node number are generated and the average running time is counted and shown in Figure 18. The time of extracting MRCs, 3CCs, MRRCs, and MGRCs are compared. As the network density increases, the difference between MGRC_acce and MGRC_naive gradually shrinks, while the running times of partitioning 3CCs, MRCs, and MRRCs gradually increase.

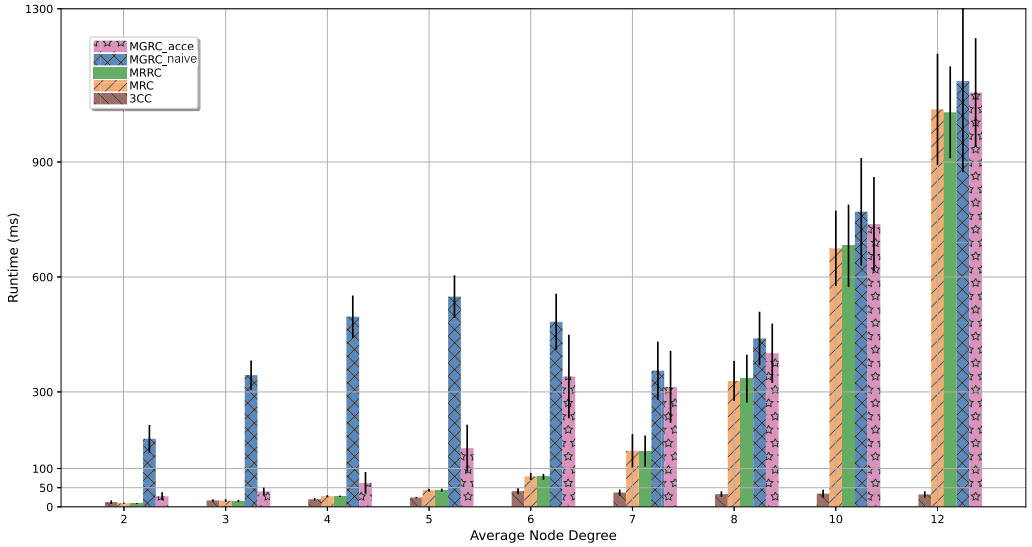


Fig. 18. The average running time of the network with different node degrees and fixed number of nodes.

5 CONCLUSION

This article presents an effective and efficient toolkit, i.e., GPART for partitioning the MRCs, MRRCs, and MGRCs in 2D graphs. The algorithm details are given and theoretical explanations of how the algorithm guarantees the properties of each component are presented. Extensive evaluations show the validity and high efficiency of GPART. The toolkit is also outsourced on GitHub. In future work, we will further study rigidity-based partition algorithms in 3D and for redundant global rigid components.

A APPENDIX

For being self-contained, we give a detailed introduction to the SPQR-Tree algorithm and a detailed example of MRC partition by PebbleGame.

A.1 SPQR-Tree

Two types of separation pairs are defined in [5]. All can be detected delicately by three times of **Depth-First Search** (**DFS**). The algorithm routine will be interpreted using an example. Considering a node pair $\{a, b\}$, it is a:

(1) **Type-1 pair**, if some segment S_i with at least two edges has only a and b in common with a cycle c, and there is any vertex v does not lie in S_i.
(2) **Type-2 pair**, if each segment connects to a cycle c only through either vertex on path p or vertices on path q, together with a and b. (p and q are the two disjoint paths connecting vertex a and b on cycle c).

For a type-1 pair, as shown in the Figure 19(a), cycle c is divided into paths p and q with a and b as endpoints. Cycle c is connected to Segment S_i only through node a and b and there is at least a vertex v that is not in S_i. Then, when the node pair $\{a, b\}$ and its corresponding edges are removed, the graph G is broken into at least two parts, vertex v with path p and segment S_i. In the type-2 condition, any segment S_i is required to be connected to the cycle c only by the vertices on the path p or path q with the vertices a and b. The positive and negative examples are given in Figure 19(b).

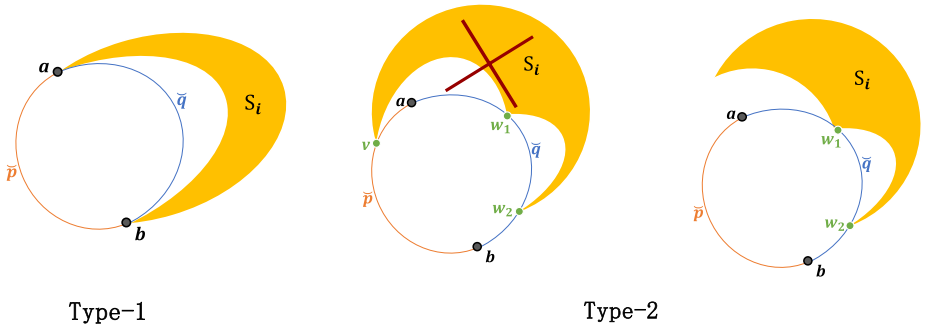
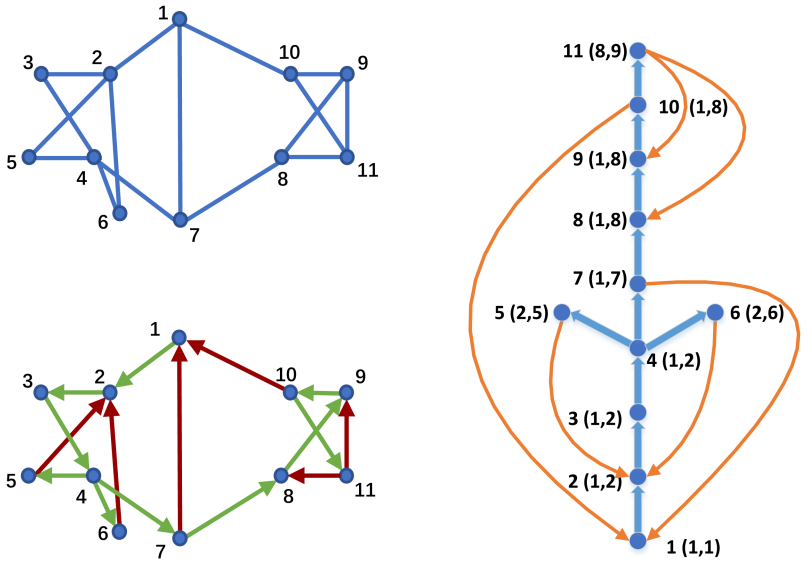


Fig. 19. Schematic diagram of the type-1 and type-2 conditions in the theorem.



(a) G and P (b) P with numbers

Fig. 20. (a) The original graph G and palm tree P; (b) The palm tree P with the $LowPT1$ and $LowPT2$ value of each node marked in brackets in sequence.

We denote the intersections of S_i with path p as w_1 and w_2 and the intersection with path q as v. $v, w_1, w_2 \neq a, b$. In the left picture, since S_i is connected to both paths p and q, respectively, disconnecting $\{a, b\}$ does not separate the graph G, while in the right picture, S_i is only connected to path q, removing $\{a, b\}$ breaks G into S_i and p two parts.

In order to find the separation pairs, the DFS-based searching method is mainly divided into four steps:

Step 1: Use DFS to convert the original graph G into a directed graph, called Palm tree P. The Palm tree contains two types of arcs. (1) *tree arcs* which are set up during DFS, point from nodes with small numbers to nodes with larger numbers. They are shown by green arcs in Figure 20(a). (2) *frond arcs* which are pointing from nodes with large numbers to those with small numbers, shown in red. At the same time, several node attributes, i.e., the *Number*, $LowPT1$, $LowPT2$, and *Father* of each node are calculated during DFS. *Number* is the order of being visited and *Father* is the preceding node in DFS. *LowPT1* and *LowPT2* of a node v record the lowest and the second lowest node number that can be reached from v on the directed Palm tree using the tree arcs and

v	A(v)
1	2
2	3
3	4
4	7, 5, 6
5	2
6	2
7	8, 1
8	9
9	10
10	1, 11
11	8, 9

(a) $A(v)$

generate path:

1. $1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 4 \longrightarrow 7 \longrightarrow 8 \longrightarrow 9 \longrightarrow 10 \longrightarrow 1$
2. $10 \longrightarrow 11 \longrightarrow 8$
3. $11 \longrightarrow 9$
4. $7 \longrightarrow 1$
5. $4 \longrightarrow 5 \longrightarrow 2$
6. $4 \longrightarrow 6 \longrightarrow 2$

(b) Generated Paths

Fig. 21. (a) The sorted neighbors of each node; (b) generated paths.

one frond arc. Figure 20(b) gives an example. The *Number (LowPT1, LowPT2)* are given beside each node. All these attributes are calculated by one DFS in [5].

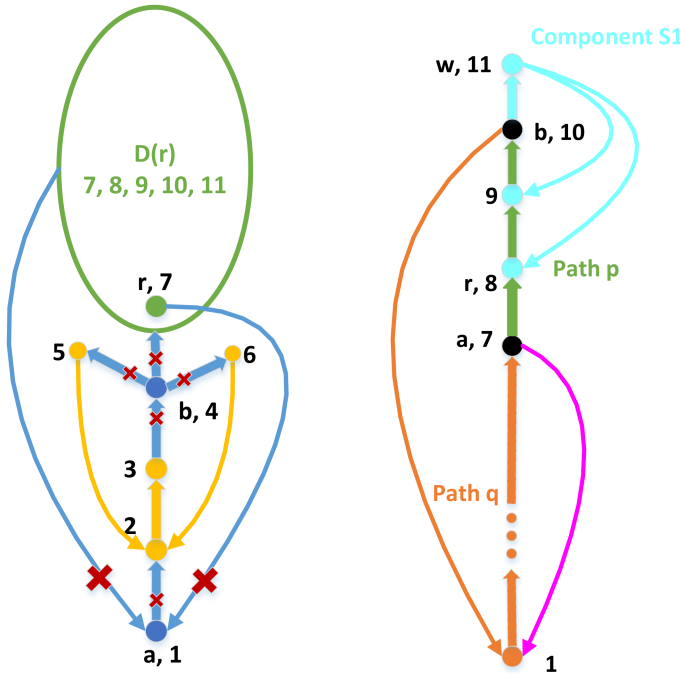
Step 2: The adjacency list $A(v)$ of each node v is processed, so that for each node v, its tree arc neighbors are arranged in ascending order according to the *LowPT1* values of the neighbors, and its frond neighbors are sorted in ascending order according to the Numbers of the neighbors. In this way, for a node v, suppose its neighbor nodes $A(v)$ are $w_1, w_2, \ldots, w_n$. Then, the neighbors satisfy $LowPT1(w_1) < LowPT1(w_2) < \cdots < LowPT1(w_n)$ if (v, w_i) are tree arcs and $w_1 < w_2 < \cdots < w_n$ if (v, w_i) are fronds. The updated adjacency list A of P is demonstrated in Figure 21(a).

Step 3: The updated adjacency list A is used to traverse the Palm tree to generate a cycle c and its corresponding segments. Still taking Figure 20(b) as an example, starting by DFS from node 1, until a frond (10,1) is found. Then the current path $1 \rightarrow 2 \rightarrow \cdots \rightarrow 10 \rightarrow 1$ is the detected cycle. After that, other neighbor edges of 10 are visited recursively to generate segment paths intersecting with the cycle, e.g., $10 \rightarrow 11 \rightarrow 8$. If all neighboring edges of 10 have been visited, the procedure will go to vertex 11 until all edges in P have been visited. The generated cycle c and its segments (other paths beyond cycle c) are shown in Figure 21(b).

Step 4: Finally, another DFS is used to traverse the generated path and verify all separation pair candidates. Let $v \rightarrow w$ denotes (v, w) is a tree arc in P. Let $v \overset{*}{\rightarrow} w$ denote w is a descendant of v. Let $D(v)$ denotes all descendant of v, and any vertex is the descendant of itself.

It has been proved in [5] that if there exists a separation pair $\{a, b\}$ in graph G, then both node a and b belong to cycle c or segment Si. Since each segment and cycle can be converted into each other (S_i can add a segment on cycle c to form a new cycle), the above separation pair determination is only for cycle c. Although the above conditions are intuitive and easy to understand, it is difficult to directly check whether the node pair is a separation pair in the algorithm. [5] proposes a further judgment condition:

(1) Type-1: there is a vertex $r \neq a, b$ such that $a \overset{*}{\rightarrow} b \rightarrow r$. $LowPT1(r) = a$, $LowPT2(r) \geq b$ and there is vertex $s \neq r, a, b$.



(a) Type-1 (b) Type-2

Fig. 22. Examples of type-1 separation pair and type-2 separation pair.

(2) Type-2: there is vertex $r \neq a, b$ such that $a \rightarrow r \overset{*}{\rightarrow} b$, a, b, r lie on a common path. $\{a, b\}$ is a separation pair if any of the following cases holds:
   (a) If for every frond (x, y) with $r \leq x < b$, there is $y \geq a$.
   (b) If for every frond (x, y) with $a < y < b$ $(a \neq 1)$ and $b \rightarrow w \overset{*}{\rightarrow} x$, there is $LowPT1(w) \geq a$.

For Type-1 pair $\{a, b\}$, according to the definition in the theorem, we only need to find a segment S_i, so that S_i and cycle c are only connected at vertex a and b, and there exists another vertex s that does not belong to S_i, then $\{a, b\}$ is a separation pair for removing ab would disconnect S_i from vertex s.

In Figure 23, we denote the child node of b as r, and $D(r)$ as all descendant nodes of r (including r itself). Assuming that there is a node v, consider the following two cases, $a < v < b$ and $v \in D(r)$. For the case of $a < v < b$ in Figure 23(b), there is $LowPT1(v) = a$ and $LowPT2(v) \leq minb, LowPT2(r)$ since $LowPT1(r) = a$, $LowPT2(r) \geq b$, which implies the frond arc of node v can only connect to nodes between a and b. For the case of $v \in D(r)$ in Figure 23(c), if the node v has a front arc connected between a and b, then there must be $LowPT2(r) < b$, which contradicts the condition of Type-1. In other words, the front arc of node v can only connect to nodes belonging to $D(r)$ or node a or node b. So as shown in Figure 23(a), the type-1 condition can guarantee that removing $\{a, b\}$ breaks G into $D(r)$ and the set of nodes between nodes a and b including node s, $D(r)$ is S_i. Figure 22(a) gives an example of detecting Type-1 pair. For node pair $\{1, 4\}$, a=1, $b = 4, r = 7$ and $D(7) = \{7, 8, 9, 10, 11\}$ (note that $r \neq 5, 6$, since $LowPT1(r) = a = 1$). Then there is $s = 5$ *or* 6 and $s \notin D(7)$. So $\{1, 4\}$ is a separation pair breaking the graph into two components: s (in yellow) and $D(r)$ (in green).

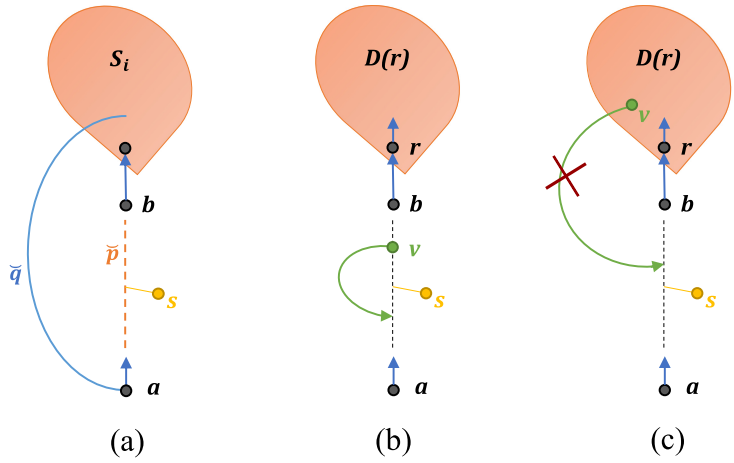


Fig. 23. The connection between the two judgment conditions of the type-1 separation pair.

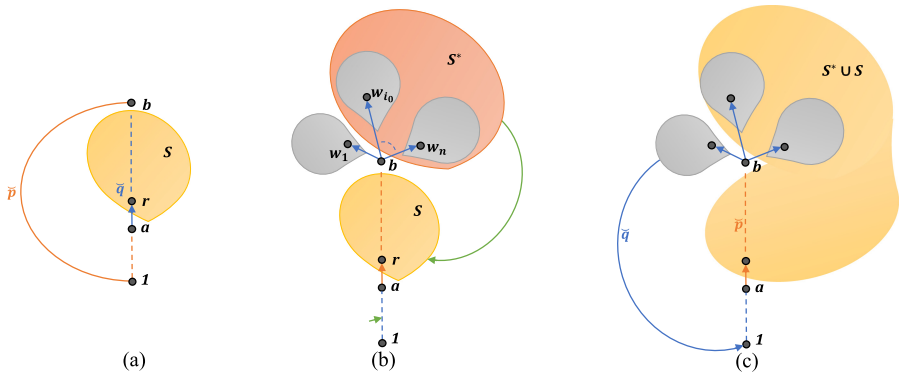


Fig. 24. The connection between the two judgment conditions of the type-2 separation pair.

Type-2 conditions include two cases, which will be divided into two points to explain below.

(1) For condition $r \leq x < b, y \geq a$, we define $S = D(b) \backslash D(r) = \{x | r \leq x < b\}$ shown in Figure 24(a) colored yellow, If $x \in S$, then $y \geq a$. The condition $y \geq a$ can guarantee that S will not connect to a node outside of $S \cup a$. As shown in Figure 23 and 24, the cycle c is divided into two parts $a \overset{pathp}{\rightarrow} b$ and $b \overset{pathq}{\rightarrow} a$ by the separation pair $\{a, b\}$, S_i is all the frond arc in S. Figure 22(b)(a) gives an example of detecting Type-2(a) pair. For node pair $\{1, 7\}$, $a = 1, r = 2, b = 7$ and $S = \{2, 3, 4, 5, 6\}$. Then all frond arcs in S are connected to vertices greater than a, which is nodes in S. When explaining Type-1, we also used the node pair $\{1, 7\}$ as an example. In the case of type-1, the separation pair $\{a, b\}$ divides G into $D(r)$ $(x \geq 7)$ and several other parts, while in the example of type-2(a), G is divided into $S(2 \leq x < 7)$ and several other parts. Separation pair $\{1, 7\}$ can satisfy both conditions because they just divide G into two parts S and $D(r)$.

(2) For condition $a < y < b$ in Figure 24(b), we consider the child of b is $w_1, w_2, \ldots, w_n$, which is the candidate of w. Then there is a index i_0, for $i < i_0$, $LowPT1(w_i) < a$ and for $i > i_0$, $LowPT1(w_i) \geq a$. We denote $D(w_i)$ as all the descendant nodes of w_i and S^* as the union $D(w_i)$ for $LowPT1(w_i) \geq a$. If $x \in S^*$, then $y \in S$ with at least one node r. The condition

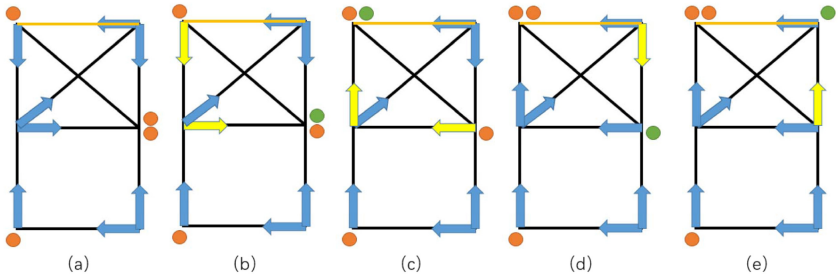


Fig. 25. The process of collecting 3 free pebbles on the two vertices of the edge (1, 2).

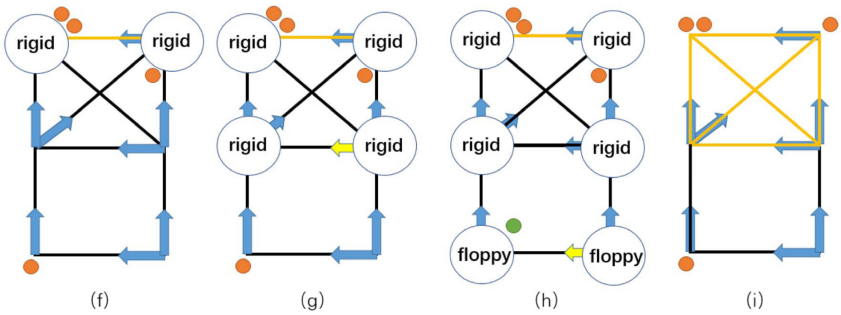


Fig. 26. The process of finding the MRC from the edge (1, 2).

$LowPT1(w) \geq a$ and $a < y < b$ can guarantee that S^* will not connect to a node outside S. In another words, for $x \in S^*$ there is no frond (x, y) with $1 \leq y \leq a$, Then removing $\{a, b\}$ will separate $S^* \cup S$ from graph.

Furthermore, if we divide cycle c into two parts, $a \overset{pathp}{\rightarrow} b$ (colored in orange) and $b \overset{pathq}{\rightarrow} a$ (colored in blue) in Figure 24(c) (path p must be in $w_i (i < i_0)$ because $i > i_0$ has $LowPT1(w_i) > a$), then the type-2 condition is to make sure the segment $S \cup S^*$ will only be connected to vertices in path p through frond arc and the segment between $\{1, a\}$ will only connect to path q.

We give an example in Figure 22(b) that $a = 7, r = 8, b = 10, w = 11$ which is eligible, and S, S^* is marked with bright blue.

A.2 An Example of Identifying MRCs by PebbleGame

This section takes Figure 4 as an example to show the process of identifying MRCs.

Figure 25 is the process of collecting 3 free pebbles on the two vertices of the edge (1, 2).

Figure 26 is the process of finding the MRC from the edge (1, 2): in the figure (f), the two vertices of (1, 2) are first marked as "rigid". Figure (g) represents the search for free pebbles starting from the neighbors of {1, 2}, since it is not found, the vertices 3 and 4 passed through in the search process are also marked as "rigid", and the rigid set is expanded to {1, 2, 3, 4} at this time. Figure (h) shows that starting from the neighbor of {1, 2, 3, 4} - vertex 5, a free pebble is found at vertex 6, and all the vertices on the search path 5->6 are marked as "floppy", At this time, all the neighbors of {1, 2, 3, 4} have been marked and cannot be expanded any more. Figure (i) represents the identified MRC= $G[\{1, 2, 3, 4\}]$.

Figure 27 shows the process of collecting three free pebbles at the two vertices of the edge (3, 5).

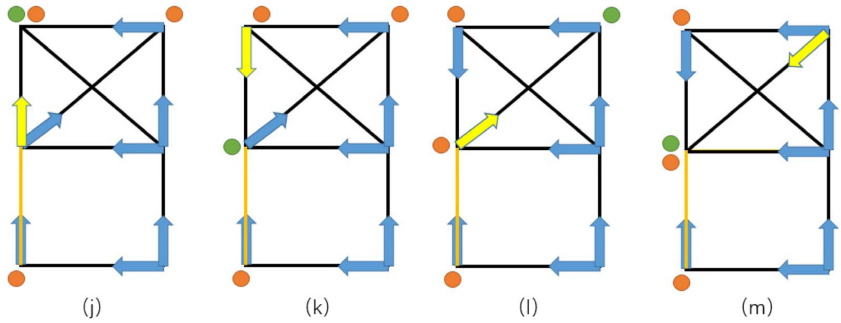


Fig. 27. The process of collecting three free pebbles at the two vertices of the edge (3, 5).

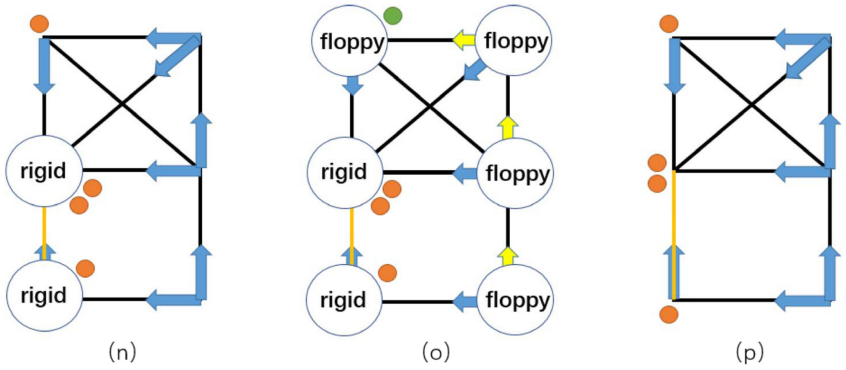


Fig. 28. The process of finding the MRC from the edge (3, 5).

Figure 28 shows the process of finding the MRC from the edge $(3, 5)$: In graph (n), the two vertices of $(3, 5)$ are first marked as "rigid". Figure (o) represents the search for free pebbles starting from the neighbors of $\{3, 5\}$. The yellow arrows represent the search path. Since a free pebble is found at vertex 1, all the vertices on the search path 6->4->2->1 are marked as "floppy", At this time, all neighbors of $\{3, 5\}$ have been marked and cannot be extended anymore, and Figure (p) represents the identified MRC = $G[\{3, 5\}]$.

REFERENCES

[1] Robert Connelly, Tibor Jordán, and Walter Whiteley. 2013. Generic global rigidity of body–bar frameworks. *Journal of Combinatorial Theory, Series B* 103, 6 (2013), 689–705.
[2] Steven J. Gortler, Alexander D. Healy, and Dylan P. Thurston. 2010. Characterizing generic global rigidity. *American Journal of Mathematics* 132, 4 (2010), 897–939.
[3] Carsten Gutwenger and Petra Mutzel. 2001. A linear time implementation of SPQR-Trees. In *Graph Drawing*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Joe Marks (Eds.). Vol. 1984. Springer Heidelberg, Heidelberg, 77–90. DOI: https://doi.org/10.1007/3-540-44541-2_8. Series Title: Lecture Notes in Computer Science.
[4] Bruce Hendrickson. 1992. Conditions for unique graph realizations. *SIAM Journal on Computing* 21, 1 (1992), 65–84.
[5] J. E. Hopcroft and R. E. Tarjan. 1972. Finding the triconnected components of a graph. Technical Report TR 140, Dept. of Computer Science, Cornell Univ.
[6] J. E. Hopcroft and R. E. Tarjan. 1973. Dividing a graph into triconnected components. *SIAM J. Comput.* 2, 3 (1973), 135–158. DOI: https://doi.org/10.1137/0202012. arXiv:https://doi.org/10.1137/0202012
[7] Bill Jackson. 2007. Notes on the rigidity of graphs. In *Levico Conference Notes*, Vol. 4.
[8] Bill Jackson and Tibor Jordán. 2005. Connected rigidity matroids and unique realizations of graphs. *Journal of Combinatorial Theory, Series B* 94, 1 (May 2005), 1–29. DOI: https://doi.org/10.1016/j.jctb.2004.11.002

[9] Donald J. Jacobs and Bruce Hendrickson. 1997. An algorithm for two-dimensional rigidity percolation: The Pebble game. *J. Comput. Phys.* 137, 2 (1997), 346–365.
[10] G. Laman. 1970. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics* 4, 4 (Oct. 1970), 331–340. DOI: https://doi.org/10.1007/BF01534980
[11] Yunhao Liu, Zheng Yang, Xiaoping Wang, and Lirong Jian. 2010. Location, localization, and localizability. *Journal of Computer Science and Technology* 25, 2 (2010), 274–297.
[12] David C. Moore, J. Leonard, D. Rus, and S. Teller. 2004. Robust distributed network localization with noisy range measurements. In *SenSys'04*. DOI: https://doi.org/10.1145/1031495.1031502
[13] Haodi Ping, Yongcai Wang, and Deying Li. 2020. HGO: Hierarchical graph optimization for accurate, efficient, and robust network localization. In *Proceedings of the 29th International Conference on Computer Communications and Networks, ICCCN*. 1–9. DOI: https://doi.org/10.1109/ICCCN49398.2020.9209620
[14] Haodi Ping, Yongcai Wang, Xingfa Shen, Deying Li, and Wenping Chen. 2022. On node localizability identification in barycentric linear localization. *ACM Trans. Sen. Netw.* 19, 1, Article 19 (Dec 2022), 26 pages. https://doi.org/10.1145/3547143
[15] H. Ping, Y. Wang, D. Li, and T. Sun. 2022. Flipping free conditions and their application in sparse network localization. *IEEE Transactions on Mobile Computing* 21, 3 (2022), 986–1003.
[16] Buddhadeb Sau and Krishnendu Mukhopadhyaya. 2013. Localizability of wireless sensor networks: Beyond wheel extension. In *Stabilization, Safety, and Security of Distributed Systems* , Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita (Eds.). Lecture Notes in Computer Science, Springer International Publishing, Cham, 326–340. DOI: https://doi.org/10.1007/978-3-319-03089-0_23
[17] Brigitte Servatius and Herman Servatius. 2010. Rigidity, global rigidity, and graph decomposition. *European Journal of Combinatorics* 31, 4 (2010), 1121–1135.
[18] Ileana Streinu and Audrey Lee. 2008. Pebble game algorithms and (k, l)-sparse graphs. *Discrete Mathematics & Theoretical Computer Science* 308, 8 (2008), 1425–1437. DOI: https://doi.org/10.1016/j.disc.2007.07.104
[19] T. Sun, Y. Wang, D. Li, Z. Gu, and J. Xu. 2018. WCS: Weighted component stitching for sparse network localization. *IEEE/ACM Transactions on Networking* 26, 5 (2018), 2242–2253.
[20] Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 114–121. DOI: https://doi.org/10.1109/SWAT.1971.10
[21] Y. Wang, T. Sun, G. Rao, and D. Li. 2018. Formation tracking in sparse airborne networks. *IEEE Journal on Selected Areas in Communications* 36, 9 (2018), 2000–2014.
[22] Hejun Wu, Ao Ding, Weiwei Liu, Lvzhou Li, and Zheng Yang. 2017. Triangle extension: Efficient localizability detection in wireless sensor networks. *IEEE Transactions on Wireless Communications* 16, 11 (Nov. 2017), 7419–7431. DOI: https://doi.org/10.1109/TWC.2017.2748563
[23] Zheng Yang and Yunhao Liu. 2010. Understanding node localizability of wireless Ad-hoc networks. In *Proceedings of the 2010 IEEE INFOCOM*. 1–9. DOI: https://doi.org/10.1109/INFCOM.2010.5462060
[24] Z. Yang, Y. Liu, and X.-Y. Li. 2009. Beyond trilateration: On the localizability of wireless ad-hoc networks. In *Proceedings of the IEEE INFOCOM 2009*. 2392–2400. DOI: https://doi.org/10.1109/INFCOM.2009.5062166
[25] H. Zhao, J. Wei, S. Huang, L. Zhou, and Q. Tang. 2019. Regular topology formation based on artificial forces for distributed mobile robotic networks. *IEEE Transactions on Mobile Computing* 18, 10 (2019), 2415–2429.

Received 12 September 2022; revised 30 March 2023; accepted 13 April 2023