Journal of Data Science 21 (3), 538–556
July 2023

DOI: 10.6339/23-JDS1110
Computing in Data Science

Generating General Preferential Attachment Networks with R Package *wdnet*

Yelie Yuan[1,*], Tiandong Wang[2], Jun Yan[1], and Panpan Zhang[3]

[1]*Department of Statistics, University of Connecticut, Storrs, Connecticut, USA*
[2]*Shanghai Center for Mathematical Sciences, Fudan University, Shanghai, China*
[3]*Department of Biostatistics, Vanderbilt University Medical Center, Nashville, Tennessee, USA*

Abstract

Preferential attachment (PA) network models have a wide range of applications in various scientific disciplines. Efficient generation of large-scale PA networks helps uncover their structural properties and facilitate the development of associated analytical methodologies. Existing software packages only provide limited functions for this purpose with restricted configurations and efficiency. We present a generic, user-friendly implementation of weighted, directed PA network generation with R package *wdnet*. The core algorithm is based on an efficient binary tree approach. The package further allows adding multiple edges at a time, heterogeneous reciprocal edges, and user-specified preference functions. The engine under the hood is implemented in C++. Usages of the package are illustrated with detailed explanation. A benchmark study shows that *wdnet* is efficient for generating general PA networks not available in other packages. In restricted settings that can be handled by existing packages, *wdnet* provides comparable efficiency.

Keywords *complete binary tree; heterogeneous reciprocity; multiple addition; user-specified preference function; weighted and directed network*

1 Introduction

Preferential attachment (PA) networks are important network models in scientific research. The standard PA model (Barabási and Albert, 1999) evolves under the mechanism that a new node is attached to an existing node with probability proportional to its degree. With the increasing needs of accommodating the heterogeneity and complexity of modern networks, a variety of extended PA network models have been proposed. Examples are directed PA models (Bollobás et al., 2003), generalized directed PA models (Britton, 2020), weighted PA models (Barrat et al., 2004), and PA models with reciprocal edges (Britton, 2020; Wang and Resnick, 2022a,b). In a general setting, the probability that a node gets a new edge is proportional to a preference function of some (node-specific) characteristics (e.g., node degree or strength). Due to their versatility, PA models have found a wide range of applications such as friendship networks (Momeni and Rabbat, 2015), scientific collaboration networks (Abbasi et al., 2012), Wikipedia networks (Capocci et al., 2006), and the World Wide Web (Kong et al., 2008), among others. Many of these networks are massive in scale.

*Corresponding author. Email: yelie.yuan@uconn.edu.

Received May 19, 2023; Accepted July 5, 2023

Table 1: Summary of packages generating PA networks.

Package	Undirected	Directed	Weighted	Reciprocal	Preference function
fastnet	✓	✓			Node degree
igraph	✓	✓			Power of node degree plus a positive constant
NetworkX	✓				Node degree
PAFit		✓			Power or logarithm of node in-degree
wdnet	✓	✓	✓	✓	General (user-specified) function of node degree/strength

Efficient generation of large-scale PA networks is critical to the investigations of their complex local and asymptotic properties. When the preference function is linear in node degree, Wan et al. (2017) developed a structured algorithm with complexity $O(n)$ for generating directed PA networks, where n is the number of generation steps. When the preference function is nonlinear in node degree, however, a naive extension of this algorithm requires visiting existing nodes one after another at each sampling step, leading to an increase in complexity to $O(n^2)$. Other node-degree-based techniques like stratified sampling or grouping (Hadian et al., 2016) cannot handle continuous edge weights. An algorithm based on a binary tree (Atwood et al., 2015) has complexity $O(n \log n)$ at the cost of additional storage of subtree information for each node. This algorithm is promising in handling weighted, directed PA networks with general preference functions. No user-friendly software package, however, has been available beyond the C implementation of Atwood et al. (2015).

Existing software packages only provide limited functions for PA network generation. Python package *NetworkX* (Hagberg et al., 2008) has a utility function for generating unweighted, undirected, linear PA networks. R packages *igraph* (Csardi and Nepusz, 2006), *PAFit* (Pham et al., 2020), and *fastnet* (Dong et al., 2020) contain functions for generating directed and/or undirected PA networks, but none of them allows edge weights. Both *igraph* and *PAFit* provide functions for preference functions that are not linear in node degrees, but they only cover a small class of power and logarithm functions. Further, no existing package implements the recently proposed PA models with reciprocity (Britton, 2020; Wang and Resnick, 2022a,b). See Table 1 for a brief summary of the functions for generating PA networks in these packages.

We introduce an R package *wdnet* (Yuan et al., 2023) for efficient generations of a general class of PA networks. The core algorithm is a generalization of the binary tree approach (Atwood et al., 2015). Our package contains substantial improvements in the flexibility for the generation of PA networks: It not only allows directed edges and edge weights, but also has additional features such as multiple edge additions, user-defined preference functions, and heterogeneous reciprocal edges, among others. See Table 1 for a summary of the features in comparison with existing packages. The engine under the hood is implemented in C++ for fast speed and then interfaced to R as facilitated by package *Rcpp* (Eddelbuettel and François, 2011).

The rest of the paper is organized as follows. In Section 2, we introduce the preliminaries of weighted, directed PA networks and present the core binary tree algorithm for generating PA networks with basic configurations. In Section 3, we illustrate the usage of the main generation function and how to control the PA network configurations for advanced features like

adding multiple edges and reciprocal edges, and defining user-specified preference functions. Performance comparisons are conducted in Section 4. Section 5 concludes with a summary of the paper and a brief introduction of other functions beyond PA network generation in package *wdnet*.

2 Generating Weighted, Directed PA Networks

We begin with an introduction to weighted, directed PA networks and a generic PA network generation framework in Section 2.1. The core of an efficient PA network generation algorithm in package *wdnet* is specified in Section 2.2.

2.1 Preliminaries

For discrete time $t = 0, 1, 2, \ldots$, let $G(t) := (V(t), E(t))$ be a weighted, directed network with node set $V(t)$ and edge set $E(t)$. For any $v_j, v_k \in V(t)$, let $(v_j, v_k, w_{jk}) \in E(t)$ denote a directed edge from v_j to v_k, where $w_{jk} > 0$ represents its weight. There can be more than one edges from v_j to v_k. For the special case of $j = k$, $(v_j, v_k, w_{jk}) \in E(t)$ is a self-loop. By convention, an initial (or seed) network $G(0)$ has at least one node and one edge.

We consider weighted, directed PA networks that allow adding multiple edges at each epoch. For illustration, we begin with a standard directed PA network that adds one edge at a time for now. There are three edge creation scenarios, respectively associated with probabilities $\alpha, \beta, \gamma \geqslant 0$, subject to $\alpha + \beta + \gamma = 1$. Note that we do not allow $\beta = 1$ to avoid degenerative situations. At each step $t \geqslant 1$, we flip a three-sided coin whose outcomes correspond to the three edge creation scenarios as follows:

(1) With probability α, add a new edge from a new node to an existing one from $G(t-1)$;
(2) With probability β, add a new edge between two existing nodes from $G(t-1)$ (self-loops are allowed);
(3) With probability γ, add a new edge from an existing node from $G(t-1)$ to a new one.

For convenience, we call these three scenarios α, β, and γ schemes, respectively; see Figure 1 for a graphical illustration.

Once an edge creation scenario is decided, we need to determine the corresponding source and/or target nodes. The probability of each candidate node from the current network being selected is proportional to its preference score, which is given by a function (called preference function) of node-specific characteristics. For unweighted, directed PA networks, the most commonly used characteristics are out- and in-degrees, whereas for weighted, directed PA networks,

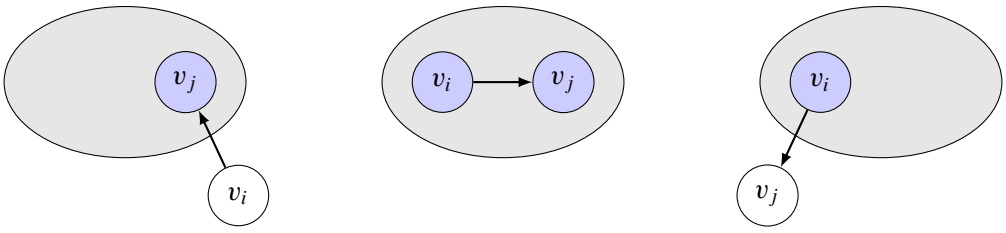


Figure 1: Three edge creation scenarios corresponding to α, β and γ schemes (from left to right), respectively.

out- and in-strengths are usually adopted. Let

$$\mathrm{O}(v_j, t) := \sum_{k:(v_j, v_k, w_{jk}) \in E(t)} w_{jk} \quad \text{and} \quad \mathrm{I}(v_j, t) := \sum_{k:(v_k, v_j, w_{kj}) \in E(t)} w_{kj}$$

represent the out- and in-strength of $v_j \in V(t)$, respectively. Let $\theta_1(v_j, t) := f_1(\mathrm{O}(v_j, t), \mathrm{I}(v_j, t))$ be the preference score for sampling v_j as a source node for a newly added edge at step $t+1$, with a non-negative function $f_1(\cdot)$ called source preference function. Then the probability of node $v_j \in V(t)$ being selected as a source node at time $t+1$ is given by

$$\frac{\theta_1(v_j, t)}{\sum_{v_k \in V(t)} \theta_1(v_k, t)}.$$

Similarly, with a non-negative target preference function $f_2(\cdot)$, one can define the preference score for sampling $v_j \in V(t)$ as a target node at time $t+1$ as well as the associated sampling probability. The default option for both f_1 and f_2 in the package is a power function:

$$f(x, y) := a_1 x^{a_2} + a_3 y^{a_4} + a_5, \tag{1}$$

where the parameters, a_i, $i = 1, \ldots, 5$, are specified by the users and can be different for f_1 and f_2. User-defined preference functions are also allowed; see Section 3.2 for details.

Once the source and target nodes of a new edge are selected, its weight is drawn independently from a distribution with probability density or mass function h on a positive support. The in- and out-strengths of the corresponding nodes are also updated, as well as their source and target preference scores. Then the algorithm proceeds to the next step.

Algorithm 1 summarizes the core structure of generating a weighted, directed PA network. The bottleneck of the algorithm is how to efficiently sample source or target nodes, i.e., the `Sample_Node()` function in Algorithm 1. We use the sampling procedure for source nodes as an illustration. At time $t+1$, the sampling step takes an updated vector of preference scores $\{\theta_1(v_j, t) : v_j \in V(t)\}$ as input. In fact, the task is straightforward. Given the grid of increasing breakpoints formed by cumulative sums of the current preference scores, find an appropriate interval that contains a uniform random variable U drawn from $\mathrm{Unif}(0, \sum_{v_j \in V(t)} \theta_1(v_j, t))$. This can be done by sequentially subtracting node preference scores from U until we find the node such that removing its preference score would cause $U \leqslant 0$. The above sampling method is a fundamental linear search, as it has to visit each of the existing nodes (one after another), and keeps updating their preference scores. This sampling approach is the `linear` method in the package, and the complexity of network generation by using this method is $O(n^2)$.

Fast sampling is possible for some special cases like when source and target preference functions are linear in node out- and in-degrees, respectively. Without loss of generality, consider a source preference function in the form of $f_1(x, y) = x + a_5$. When the edges are unweighted, the interval (containing U) can be determined by one uniform draw (Wan et al., 2017). This algorithm acts like by putting the node labels into a bag the same number of times as their out-degrees and then drawing a label from the bag, which is analogous to the Pólya urn theory (Mahmoud, 2008). This technique is called `bag` in package *igraph*, and the same name is adopted in package *wdnet*. When the edges are weighted, by a clever maneuver, the sampling step for the whole generation process can be done in one batch with a pre-set cumulative sum vector of the edge weights by using the base R function `findInterval()`. This is an extension of the `bag` algorithm, so we named it `bagx`. See Appendix A for more details about `bag` and `bagx`.

Algorithm 1: Generating a weighted, directed PA network.

Input: Number of steps n;
initial network $G(0) = (V(0), E(0))$;
probabilities for three edge creation scenarios α, β, γ;
probability density (or mass) function h for drawing edge weights from;
preference functions for source node f_1 and for target node f_2.

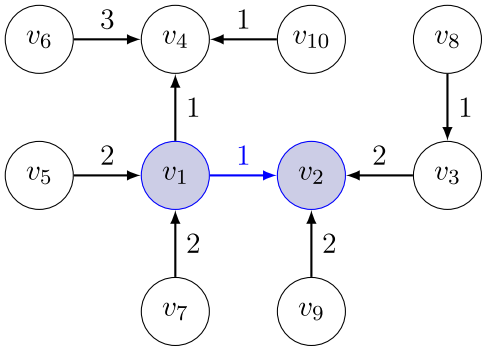
Output: $G(n) = (V(n), E(n))$.

Algorithm:

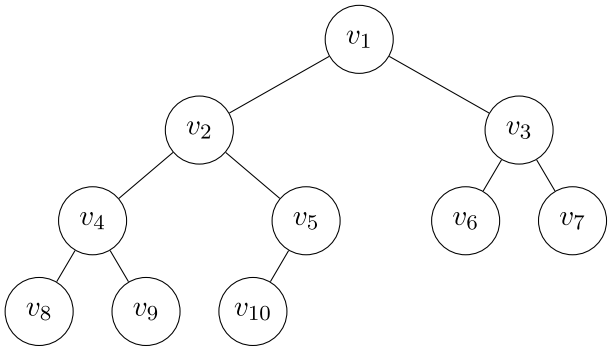
Initialize $(n + |V(0)|)$-dimensional zero vectors of out- and in-strengths, O and I;
Initialize $(n + |V(0)|)$-dimensional zero vectors of source and target preference scores $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$;
Update O, I, $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ with initial network $G(0)$;
$t \leftarrow 1$;
while $t \leqslant n$ **do**
  $N \leftarrow |V(t-1)|$; /* Number of nodes in $G(t-1)$ */
  Draw $\psi \sim \text{Unif}(0, 1)$;
  if $\psi \leqslant \alpha$ **then** /* α scheme */
    $j \leftarrow N + 1$; /* Source node index */
    $k \leftarrow$ Sample_Node$(V(t-1), 2)$; /* Target node index; the "2" as the second argument indicates target node sampling. */
    $V(t) \leftarrow V(t-1) \cup \{v_j\}$;
  else if $\alpha < \psi \leqslant \alpha + \beta$ **then** /* β scheme */
    $j \leftarrow$ Sample_Node$(V(t-1), 1)$; /* The "1" as the second argument indicates source node sampling */
    $k \leftarrow$ Sample_Node$(V(t-1), 2)$;
  else if $\psi > \alpha + \beta$ **then** /* γ scheme */
    $j \leftarrow$ Sample_Node$(V(t-1), 1)$;
    $k \leftarrow N + 1$;
    $V(t) \leftarrow V(t-1) \cup \{v_k\}$;
  Draw w from weight distribution h ; /* Sample edge weight */
  $E(t) \leftarrow E(t-1) \cup \{(v_j, v_k, w)\}$; /* Add the new edge to $G(t)$ */
  $\text{O}[j] \leftarrow \text{O}[j] + w$; /* Update preference function inputs */
  $\text{I}[k] \leftarrow \text{I}[k] + w$;
  $\boldsymbol{\theta}_1[j] \leftarrow f_1(\text{O}[j], \text{I}[j])$; /* Update preference scores */
  $\boldsymbol{\theta}_2[j] \leftarrow f_2(\text{O}[j], \text{I}[j])$;
  $\boldsymbol{\theta}_1[k] \leftarrow f_1(\text{O}[k], \text{I}[k])$;
  $\boldsymbol{\theta}_2[k] \leftarrow f_2(\text{O}[k], \text{I}[k])$;
  $t \leftarrow t + 1$;
return $G(n) = (V(n), E(n))$;

2.2 Node Sampling Based on a Binary Tree

Our recommended approach for `Sample_Node()` is a binary tree approach that extends the algorithm in Atwood et al. (2015) to weighted, directed PA networks with general preference functions. In a binary tree structure, each node has no more than two children nodes. The two



(a) A simple network. Edge weights are marked next to the edges.



(b) The binary tree structure.

Node	$\kappa(v_j)$	$l(v_j)$	$r(v_j)$	$\mathrm{O}(v_j)$	$\mathrm{I}(v_j)$	$\theta_1(v_j)$	$\theta_2(v_j)$	$\eta_1(v_j)$	$\eta_2(v_j)$
v_1	/	v_2	v_3	2	4	3	5	25	25
v_2	v_1	v_4	v_5	0	5	1	6	12	16
v_3	v_1	v_6	v_7	2	1	3	2	10	4
v_4	v_2	v_8	v_9	0	5	1	6	6	8
v_5	v_2	v_{10}	/	2	0	3	1	5	2
v_6	v_3	/	/	3	0	4	1	4	1
v_7	v_3	/	/	2	0	3	1	3	1
v_8	v_4	/	/	1	0	2	1	2	1
v_9	v_4	/	/	2	0	3	1	3	1
v_{10}	v_5	/	/	1	0	2	1	2	1

(c) Node attributes.

Figure 2: A generated network with initial graph colored with blue (panel a), its corresponding complete binary tree structure (panel b) and a summary of node attributes (panel c); the source and target preference functions are respectively given by $f_1(x, y) = x + 1$ and $f_2(x, y) = y + 1$.

children nodes of a parent node are distinguished by their positions, i.e., the left and the right child. Except for the root node, each node has only one parent node. A complete binary tree refers to a binary tree with all levels fully filled except for the last level. The last level is not necessarily completely filled, but has to be filled from left to right. A hypothetical example of complete binary tree is given in Figure 2b. An important application of binary trees is searching. The complexity of searching a specific node in a binary tree with n nodes is of order $O(\log n)$ (Mahmoud, 1992), which is more efficient than the linear search of complexity $O(n)$.

We translate a PA network to a binary tree as follows. Each node in a PA network corresponds to a node in the associated complete binary tree based on the time of its creation. Suppose that $G(0)$ contains only one node v_1 with a self-loop, then v_1 becomes the root of the complete binary tree. Then node v_2 that joins the PA network at $t = 1$ is the left child of v_1, and the next new node v_3, which joins the PA network at $t = 2$, is the right child of v_1. For the next newcomer v_4 (at time $t = 3$), it is attached to v_2 as a left child since the first level (consisting of v_2 and v_3) is fully filled. The transition continues in this fashion until all nodes in the PA

Algorithm 2: Node sampling function based on a binary tree storage structure.

Input: Node set $V(t-1)$;
$i \in \{1, 2\}$ for sampling a source or a target node, respectively.
Output: j, the index of the sampled node.

Function `Sample_Node`$(V(t-1), i)$**:**
  $j = 1$; `/* Start from the root` v_1 `*/`
  Draw $U \sim \text{Unif}(0, \eta_i(v_1, t-1))$;
  while $j \leqslant |V(t-1)|$ **do**
    $U \leftarrow U - \theta_i(v_j, t-1)$;
    `temp` $\leftarrow \eta_i(l(v_j), t-1)$;
    if $0 < U \leqslant$ `temp` **then** `/* Search in the subtree with root` $l(v_j)$ `*/`
      $j \leftarrow$ index of $l(v_j)$;
    else if $U >$ `temp` **then** `/* Search in the subtree with root` $r(v_j)$ `*/`
      $U \leftarrow U -$ `temp`;
      $j \leftarrow$ index of $r(v_j)$;
    else if $U \leqslant 0$ **then** `/* Return the index of the sampled node` v_j `*/`
      return j;

network are added to the complete binary tree. For an initial graph $G(0)$ containing more than one nodes, a node enumeration $\{v_1, v_2, \ldots, v_{|V(0)|}\}$ is required before constructing the binary tree; see Figures 2a and 2b for an example with an initial graph consisting of two nodes (connected by one edge which is colored with blue). Different enumeration orders of the initial network result in different binary trees and, consequently, different networks because of the underlying sampling mechanism.

Having built a complete binary tree, we augment the nodes therein according to a collection of node attributes. At step t, the binary tree node v_j stores the following information: parent (except for v_1) $\kappa(v_j)$, left child $l(v_j)$, right child $r(v_j)$, out-strength $\text{O}(v_j, t)$, in-strength $\text{I}(v_j, t)$, preference score as a source node $\theta_1(v_j, t)$, preference score as a target node $\theta_2(v_j, t)$. For now we consider θ_1 and θ_2 as functions of node out- and in-strengths, but in general, θ_1 and θ_2 can be functions of any node-level characteristics. Additionally, let $\eta_1(v_j, t)$ and $\eta_2(v_j, t)$ denote the total preference of source and target nodes of the subtree (a portion of the binary tree consisting of a node and all of its descendants) with root v_j, respectively, giving rise to the following relationship:

$$\eta_i(v_j, t) = \eta_i(l(v_j), t) + \eta_i(r(v_j), t) + \theta_i(v_j, t), \qquad i \in \{1, 2\}.$$

Figure 2c summarizes the node attributes, including η_1 and η_2, from the complete binary tree (in Figure 2b) that is constructed from the weighted network in Figure 2a.

Node sampling based on the binary tree structure, available as the `binary` method in *wdnet*, is summarized in Algorithm 2. Generally, the algorithm searches for the subtree to which the potentially sampled node belongs in a recursive manner until the root of the resulting subtree (or the node itself if it is at the bottom level) is returned. The subtree-based searching substantially reduces the complexity of the sampling step from $O(n)$ (for the linear search algorithm) to $O(\log n)$. The network generation algorithm with binary search thus has complexity $O(n \log n)$.

Upon the creation of a new edge (v_j, v_k, w_{jk}), the following quantities need to be updated: node strengths $\text{O}(v_j, t)$ and $\text{I}(v_k, t)$; preference scores $\theta_1(v_j, t)$, $\theta_1(v_k, t)$, $\theta_2(v_j, t)$ and $\theta_2(v_k, t)$;

total preference scores of subtrees $\eta_1(v_j, t)$, $\eta_2(v_j, t)$, $\eta_1(\kappa(v_j), t)$, $\eta_2(\kappa(v_j), t)$, etc. The update of total preference scores is not shown in the algorithm. It traces the growth path through subtrees (backwards), and has the same time complexity $O(\log n)$ as the sampling method.

3 Usage

We start with the main function to generate PA networks with basic configurations in Section 3.1, and then introduce additional features in Section 3.2.

3.1 Main Generation Function

The function `rpanet()` is used to generate PA networks.

```
library("wdnet")
args(rpanet)
```

```
function (nstep, initial.network = list(edgelist = matrix(c(1,
    2), nrow = 1), edgeweight = 1, directed = TRUE), control,
    method = c("binary", "linear", "bagx", "bag"))
NULL
```

The first three arguments of `rpanet()` are: the number of steps (`nstep`), the initial network (`initial.network`), and a list of control parameters (`control`). Specifications of the control parameters are done through a collection of functions as we proceed. The `method` argument specifies which of the following four implemented methods is used to generate a PA network: `binary` (default), `linear`, `bagx`, and `bag`.

With respect to the required inputs in Algorithm 1, we elaborate the usage of `rpanet()` and its specifications via the `control` argument as follows.

Initial Network The `initial.network` is specified by a list containing a matrix of edges (`edgelist`) in the order of edge creations, a vector of edge weights (`edgeweight`), and a logical argument (`directed`) indicating whether the initial network as well as the generated network are directed. Each row of `edgelist` has two elements specifying the two nodes of an edge. The length of `edgeweight` is equal to the number of rows of `edgelist`. If `edgeweight` is not specified, all edges from the initial network are assumed to have weight 1. The default initial network has only one edge, $(1, 2, 1.0)$, corresponding to a network consisting of two nodes with a unit-weight edge from node 1 to node 2. The `initial.network` can also be specified by a `wdnet` object, which can be constructed via utility functions `edgelist_to_wdnet()` or `adj_to_wdnet()`. The following example sets up an initial network with two weighted edges, $(1, 2, 0.5)$ and $(3, 4, 2.0)$.

```
netwk0 <- list(edgelist = matrix(c(1, 2, 3, 4), nrow = 2, byrow = TRUE),
    edgeweight = c(0.5, 2.0), directed = TRUE)
```

Edge Scenarios The function `rpa_control_scenario()` is used to specify the probability of each edge creation scenario. Based on the real data analysis in Wan et al. (2017), we also include two additional edge creation scenarios to the α, β and γ schemes introduced in Section 2.1: (1)

the ξ scheme where a new edge is added between two new nodes, and (2) the ρ scheme where a new node with a self-loop is added. Self-loops are allowed in the β scheme by setting `beta.loop` to be `TRUE`. When `beta.loop = FALSE`, the order of sampling source and target nodes may affect the structure of generated PA network as controlled by the logical arguments `source.first`. The default settings of these arguments are $\alpha = 1$, $\beta = \gamma = \xi = \rho = 0$, `beta.loop = source.first = TRUE`. The following example sets up a configuration that excludes self-loops under the β scheme and samples target nodes before source nodes.

```
ctr1 <- rpa_control_scenario(alpha = 0.2, beta = 0.6, gamma = 0.2,
    beta.loop = FALSE, source.first = FALSE)
```

Edge Weights Edge weights are controlled by `rpa_control_edgeweight()` through its `sampler` argument. This argument accepts a function that takes a single parameter, representing the number of sampled values, and returns a vector of sampled edge weights. Note that the sampled values must be positive real numbers. The default setting is `sampler = NULL`, referring to the case where all new edges take unit weight. As shown in the following example, edge weights are sampled from a gamma distribution with shape 5 and scale 0.2; the "`+`" operator has been overloaded to concatenate multiple control lists.

```
my_rgamma <- function(n) rgamma(n, shape = 5, scale = 0.2)
ctr2 <- ctr1 + rpa_control_edgeweight(sampler = my_rgamma)
```

Preference Functions The default preference function is in the form of $f(x, y)$ given in Equation (1), which covers a wide range of sub-linear, linear and super-linear functions. The function `rpa_control_preference()` controls the configuration of this $f(x, y)$ with `ftype = "default"` along with two arguments `sparams` and `tparams` which specify the parameters of the source and target preference functions of Equation (1), respectively. For directed PA networks, the default source and target preference functions are, respectively, $f_1(x, y) = x+1$ and $f_2(x, y) = y+1$. This is controlled by default value of `sparams = c(1, 1, 0, 0, 1)` and `tparams = c(0, 0, 1, 1, 1)`. The following example sets the source preference function to $f_1(x, y) = x^2 + 1$ and the target preference function to $f_2(x, y) = y^2 + 1$:

```
ctr3 <- ctr2 + rpa_control_preference(ftype = "default",
  sparams = c(1, 2, 0, 0, 1), tparams = c(0, 0, 1, 2, 1))
```

For undirected networks, the default preference function has the form $g(x) = x^{b_1} + b_2$, and argument `params` specifies the preference parameters b_1 and b_2, with default values given by $b_1 = b_2 = 1$.

We further allow users to specify their own preference functions; see Section 3.2 for details.

Returned Value Function `rpanet()` returns a list of class `wdnet` containing the following components: `newedge` is a vector summarizing the number of new edges added at each step; `edge.attr` is a data frame containing edge weights and edge creation scenarios, where edges from schemes α, β, γ, ξ, ρ are respectively denoted as scenarios 1, 2, 3, 4, 5, and edges from the initial network are denoted as scenario 0; `node.attr` is a data frame containing node out- and in-strengths as well as source and target preference scores. Other items are self-explanatory.

```
set.seed(12)
netwk3 <- rpanet(nstep = 1e3, initial.network = netwk0, control = ctr3)
names(netwk3)
```

```
[1] "edgelist"  "newedge"   "control"   "directed"  "edge.attr" "weighted"
[7] "node.attr"
```

```
print(netwk3)
```

```
Weighted: TRUE
Directed: TRUE
Number of edges: 1002
Number of nodes: 402

Edges:
  source target    weight scenario
1      1      2 0.5000000        0
2      3      4 2.0000000        0
3      5      2 0.4591485        1
4      5      6 0.4347588        3
5      5      7 0.6565225        3
...omitted remaining edges

Node attributes:
      outs        ins     spref       tpref
1 4.523894  0.0000000 21.465613    1.000000
2 0.000000  1.7270612  1.000000    3.982741
3 2.000000  0.8624366  5.000000    1.743797
4 2.823930  2.0000000  8.974579    5.000000
5 2.386927 43.7566871  6.697419 1915.647667
...omitted remaining nodes
```

3.2 Additional Features

The package *wdnet* provides a few additional distinctive features in the PA network generation process that are not available in other software packages. These features are obtained by adapting the Algorithm 1.

Multiple Edge Addition The creation of multiple edges at one step is controlled by function `rpa_control_newedge()`. The first argument of this function, `sampler`, determines the distribution of the number of new edges to be added in the same step. This argument accepts a function that takes a single parameter, representing the number of values to be sampled, and returns a vector of sampled number of new edges. Note that the sampled values must be positive integers. By default, `sampler` is set to `NULL`, representing the addition of only one edge at each step.

When more than one edges are added at one step, we keep the node strengths and their preference scores unchanged until all edges at this step have been added. Users need to specify whether to sample the candidate nodes with replacements or not. For directed networks, the

logical arguments `snode.replace` and `tnode.replace` determine whether the source and target nodes are sampled with replacement, respectively. For undirected networks, only one logical argument `node.replace` needs to be specified.

The code below updates the setting from `ctr3` by letting the number of new edges follows a unit-shifted Poisson distribution (Wang and Resnick, 2023) with probability mass function

$$\Pr(X = k) = e^{-2}\frac{2^{k-1}}{(k-1)!}, \qquad k \geqslant 1.$$

Both source and target nodes are sampled *without* replacement.

```
ctr4 <- ctr3 + rpa_control_newedge(sampler = function(n) rpois(n, 2) + 1,
    snode.replace = FALSE, tnode.replace = FALSE)
```

Reciprocal Edges Reciprocal edges are mutual links between two nodes. We allow reciprocal edges under a heterogeneous setting (Wang and Resnick, 2022b) where each node belongs to one of the $K \geqslant 1$ groups. With the emergence of each new node, its group label is given according to a user-specified probability vector $\boldsymbol{\pi} := (\pi_1, \pi_2, \ldots, \pi_K)$, where $0 \leqslant \pi_k \leqslant 1$ represents the probability that the node belongs to group $k \in \{1, 2, \ldots, K\}$. Similar to stochastic block models, there is a probability block matrix $\boldsymbol{q} := (q_{k\ell})_{K\times K}$ (not necessarily symmetric), which is also specified by the users, to determine the probability of adding a reciprocal edge for each new edge joining the network. For example, consider a new edge (v_i, v_j, w_{ij}) where v_i and v_j are respectively labeled with $k = 2$ and $\ell = 3$, then its reciprocal correspondence (v_j, v_i, w_{ji}) is added to the network instantaneously with probability q_{32}. The weight of the reciprocal edge (if added), w_{ji}, is independently sampled with configurations specified in `rpa_control_edgeweight()`. When more than one new edges are added at a step, the reciprocal edge for each of them is added independently, one after another.

The function `rpa_control_reciprocal()` gives the configurations of reciprocal edges. The arguments `group.prob` and `recip.prob` specify the probability vector $\boldsymbol{\pi}$ and the block probability matrix $\boldsymbol{q}$, respectively. In addition, the logical argument `selfloop.recip` determines whether reciprocal edges for self-loops are allowed. Their default settings are `group.prob = NULL`, `recip.prob = NULL` and `selfloop.recip = FALSE`, respectively, referring to the case of no immediate reciprocal edges. The following example creates a configuration with $\boldsymbol{\pi} = (0.4, 0.6)$ and

$$\boldsymbol{q} = \begin{pmatrix} 0.4 & 0.1 \\ 0.2 & 0.5 \end{pmatrix}.$$

```
ctr5 <- ctr4 + rpa_control_reciprocal(group.prob = c(0.4, 0.6),
    recip.prob = matrix(c(0.4, 0.1, 0.2, 0.5), nrow = 2, byrow = TRUE))
```

By default, all nodes in the seed network are assumed to be from group 1. This configuration can be easily customized as shown in the following example, where nodes 1 and 4 are from group 1, while nodes 2 and 3 are from group 2.

```
netwk0 <- list(edgelist = matrix(c(1, 2, 3, 4), nrow = 2, byrow = TRUE),
    edgeweight = c(0.5, 2), directed = TRUE, nodegroup = c(1, 2, 2, 1))
netwk5 <- rpanet(1e3, control = ctr5, initial.network = netwk0)
```

Node groups are recorded in the data frame `node.attr`; immediate reciprocal edges are denoted as scenario 6 in the data frame `edge.attr`.

Customized Preference Functions User-defined preference functions in C++ syntax are allowed by setting `ftype = "customized"` in `rpa_control_preference()`. This is implemented in C++ through the utility functions in R package *RcppXPtrUtils* (Ucar, 2022). For directed networks, one-line C++ expressions can be passed to arguments `spref` and `tpref` to define the source and target preference functions, respectively. The expressions are strings in R but with valid C++ syntax as transformations of `outs` and `ins`. The strings are passed to the function `cppXPtr()` in package *RcppXPtrUtils*, which compiles the source code and returns an `XPtr` (external pointer) that points to the compiled preference function. The default preference functions $f_1(x, y) = x + 1$ and $f_2(x, y) = y + 1$ can be equivalently achieved by setting `spref = "outs + 1"` and `tpref = "ins + 1"`. The following example sets the preference functions to $f_1(x, y) = \ln(x + 1) + 1$, $f_2(x, y) = \ln(y + 1) + 1$:

```
ctr6 <- ctr5 + rpa_control_preference(ftype = "customized",
    spref = "log(outs + 1) + 1", tpref = "log(ins + 1) + 1")
```

For undirected networks, argument `pref` specifies a one-line C++ expression as a transformation of node strength `s`. The default preference function $g(x) = x + 1$ could be equivalently achieved by `pref = "s + 1"`. Users need to ensure the non-negativity of the preference functions. The generation process will be terminated if a negative preference value is encountered.

For more advanced preference functions which may take multiple lines of C++ code, see examples in Appendix B.

4 Benchmarks

In this section, we generate weighted and unweighted PA networks with different sizes and preference functions via our package (*wdnet*, version 1.2.0), *igraph* (version 1.3.5) and *PAFit* (version 1.2.5), and compare their performance. All simulations were run on a single core of Intel Xeon Gold 6150 CPU @ 2.70GHz with 16 GB of RAM.

Weighted Networks Since the other two packages (i.e., *igraph* and *PAFit*) do not admit edge weights, the comparison of weighted PA network generation is between the `linear` and `binary` methods in our package *wdnet*. Specifically, we assign the same probabilities to edge creation scenarios (i.e., $\alpha = \beta = \gamma = 1/3$), set the source and target preference functions respectively to $f_1(x, y) = x^k + 0.1$ and $f_2(x, y) = y^k + 0.1$ with $k \in \{0.5, 1, 2\}$ (where $k = 0.5$ and $k = 2$ respectively refer to sub-linearity and super-linearity). Draw the edge weights independently from Gamma$(5, 0.2)$. For each k, we generate PA networks of various evolutionary steps (i.e., $n \in \{10^3, \ldots, 10^7\}$) with a simple initial network consisting of two nodes and one edge $(1, 2, 1)$ (default).

The top three panels of Figure 3 compare the median runtimes of generating 100 independent weighted, directed PA networks via `binary` and `linear` methods. When preference functions are sub-linear ($k = 0.5$) or linear ($k = 1$), the `binary` method is much more efficient than the `linear` method. Besides, the larger the number of steps is, the more advantageous it is to use the `binary` method. Some simulations for the `linear` method are omitted because they

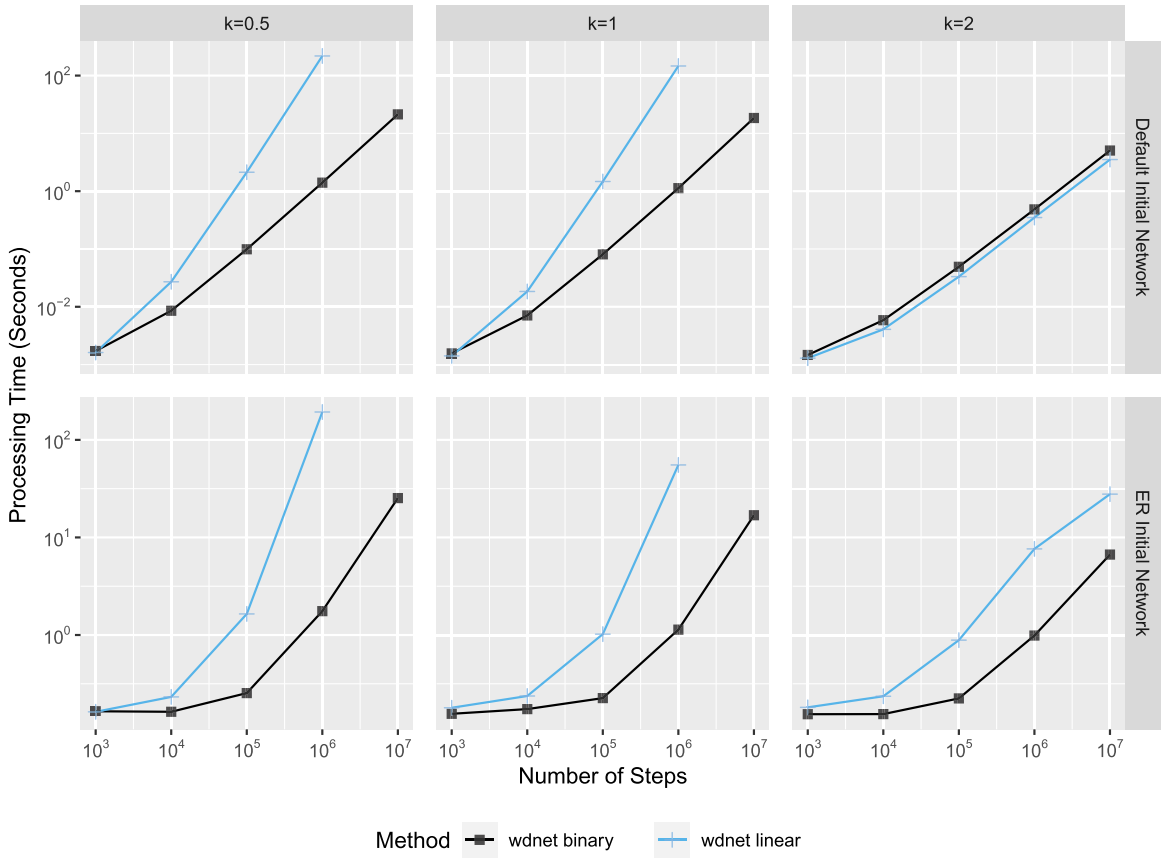


Figure 3: Algorithm runtime for weighted networks with default initial network (upper) of one edge $(1, 2, 1)$ and with initial weighted ER networks (lower) of 10^4 nodes and 10^6 edges. Edge weights, including those in the initial weighted ER networks, are drawn from Gamma(5, 0.2). Probability of edge schemes are $\alpha = \beta = \gamma = 1/3$. Preference functions are $f_1(x, y) = x^k + 0.1$, $f_2(x, y) = y^k + 0.1$ with $k \in \{0.5, 1, 2\}$. Each point represents the median runtime of 100 replications.

are excessively time-consuming. For a super-linear preference function ($k = 2$), the difference in generation speed between the two methods becomes subtle. A further investigation reveals that the sum of source (and target) preference scores of the 20 earliest created nodes, $\{v_1, v_2, \ldots, v_{20}\}$, take 99% of the total (for all nodes), making them dominant in the sampling process. Those early created nodes are always quickly selected under whichever edge addition scenario since linear search visits those "ancestors" first. Consequently, the time cost of using `linear` method is significantly reduced.

To further investigate the impact of early created nodes in the sampling process, we consider a modified (i.e., weighted and directed) Erdös–Rényi (ER) network (Erdös and Rényi, 1959; Gilbert, 1959) as an initial network. For each simulation run, we generate an ER network with 10^4 nodes and 10^6 edges, where edge weights are drawn independently from Gamma(5, 0.2). We keep all other parameters same as in the previous experiment, and give the median runtime (of 100 independently generated PA network replica) in the bottom three panels of Figure 3. We observe similar patterns for $k = 0.5$ and $k = 1$, so focus on $k = 2$ only. Here the `binary` algorithm

outperforms the `linear` method, since the large seed network alleviates the domination of old nodes in the subsequent sampling process.

In fact, most nodes that are sampled throughout the process are those with high strengths in the seed graph. Owing to the feature of ER network, a few hundred nodes (out of 10^4) in the seed network are repeatedly sampled. However, a larger pool (compared to 20 in the previous experiment) results in longer runtime when using the `linear` method. On the other hand, we believe the performance of `linear` algorithm will improve as n gets larger since fewer nodes will continue to be dominant in the sampling process. Since we have added a sorting procedure in the `linear` algorithm, those nodes will be quickly selected. Accordingly, the `linear` method may finally outperform the `binary` method for extremely large networks. Last but not least, we find the tracing curves between 10^3 and 10^4 become flatter in the bottom plots when compared to their upper counterparts (for each k). This is due to the large seed graph, which requires a certain amount of time to initialize the sampling process. Consequently, there is a small difference in the total generation time for relatively small n.

Unweighted Networks Next, we compare the performance of generating unweighted PA networks using our package *wdnet* and the other two popular packages *PAFit* and *igraph*. Since *PAFit* and *igraph* allow the α scheme only, we now set $\alpha = 1$ in the `rpa_control_scenario()` function. Under such setting, we only need to define a target preference function in the form of $f_2(x, y) = y^k + 0.1$, where we again assume $k \in \{0.5, 1, 2\}$. Similar to the previous experiments, we generate unweighted PA networks with $n \in \{10^3, \ldots, 10^7\}$ for each k. A default initial network is adopted for each simulation run. The results are given in the top three panels of Figure 4. For $k = 0.5$ and $k = 1$, we find the `binary` algorithm in our package and *igraph* (`psumtree` method) are almost equally efficient, and outperform the rest. The performance of `linear` method in our package is better than *PAFit* (similar to linear search) since the former is implemented in C++ whereas the latter is implemented in R. For $k = 2$, we do not see much difference among the two methods in our package *wdnet* and that in *igraph*, which is consistent with our conclusions for the weighted PA network generation experiment. Overall, *PAFit* is the least efficient, especially for generating large PA networks.

Lastly, we repeat our simulations by considering large ER networks as the initial graphs in order to alleviate the impact of few old nodes during the generation process. The setup is the same as that for weighted network simulations. Noticing that *PAFit* does not accept arbitrary initial networks, we exclude it from this set of simulation comparisons. The corresponding results are shown in the bottom three panels of Figure 4. Similar to the conclusions drawn for weighted PA networks, the `binary` method outperforms the `linear` method in our package owing to the less influence from old nodes when $k = 2$. Moreover, there is little performance difference between the `binary` method and *igraph* across all considered k values.

5 Discussions

Our R package *wdnet* provides useful tools to efficiently generate large-scale PA networks. The package admits a wide range of PA network specifications such as multiple edge addition scenarios, weighted and directed edges, and reciprocal edges. Our implementations extend those discussed in Wan et al. (2017) and Britton (2020), most of which are not available in other existing packages. A distinctive feature of the package is that it allow users to define their own preference functions. Our `binary` algorithm is efficient for general situations. Our `linear` algo-

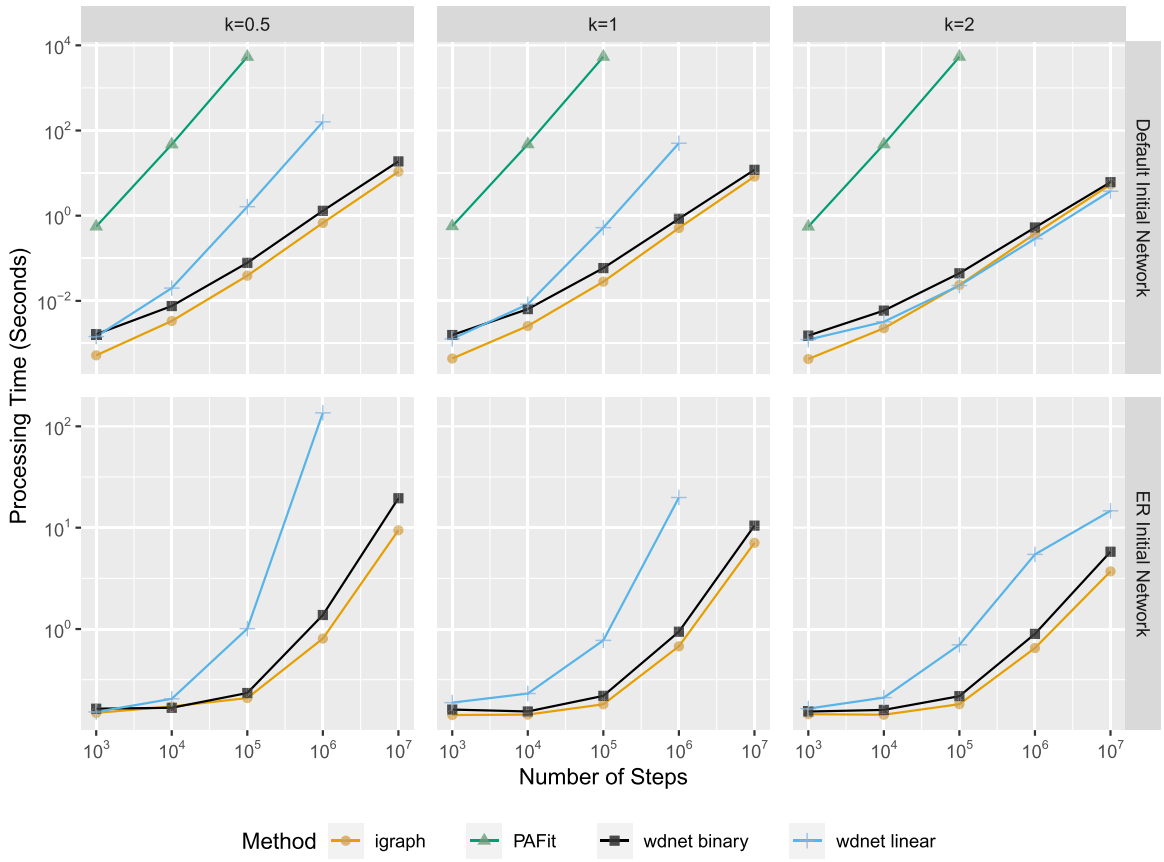


Figure 4: Algorithm runtime for unweighted networks with default initial network (upper) of one edge from v_1 to v_2 and with initial unweighted ER networks (lower) of 10^4 nodes and 10^6 edges. Probability of edge schemes are $\alpha = 1$, $\beta = \gamma = 0$. The target preference function is $f_2(x, y) = y^k + 0.1$. Each point represents the median runtime of 100 replications.

rithm outperforms implementations of the same algorithm in other packages due to its sorting step. The core implementation is in C++ for fast speed.

Efficient generation of PA networks facilitates investigation of PA networks properties and goodness-of-fit diagnosis in real applications. A PA network is controlled by many parameters. When theoretical properties, for example, transitivity and clustering coefficients, are challenging to derive, their empirical versions can be easily learned from generating many realizations given the model parameters. When the initial network size is large relative to the desired PA network size, its impact may not be ignorable and could be studied through simulations. In real applications, the goodness-of-fit diagnosis of a PA network can be done by generating many replicates from the fitted PA model and comparing the observed network statistics with the empirical distribution of the same statistics from the replicates. Such goodness-of-fit check may motivate modification of the PA networks to fit the real data better (e.g., Wang et al., 2022).

Beyond general PA network generation, *wdnet* also provides a collection of other functions. Specifically, several centrality measures are available via function `centrality()`, including the recently proposed weighted PageRank centrality (Zhang et al., 2022). Assortativity measures for weighted directed networks discussed in (Yuan et al., 2021) are available via func-

tion `assortcoef()`. A degree-preserving rewiring algorithm for generating networks with predetermined assortativity coefficients (Wang et al., 2022) is available via function `dprewire()`. All these functions are derived from recent research, so they are not available in other packages.

Supplementary Materials

(1) The code used for benchmarks and the R Markdown source for the paper can be found at https://github.com/Yelie-Yuan/code-sharing/tree/main/generating-pa.
(2) The development version of the package is available at https://gitlab.com/wdnetwork/wdnet.

Funding

Dr. Wang and Dr. Yan's works were partially supported by the NSF grant DMS2210735.

A Alternative Sampling Method for Special Cases

Fast sampling is available when source (target) preference functions are linear. We demonstrate this approach through an example of sampling source nodes with a preference function $f_1(x, y) = x + a_5$.

As shown in Section 2.1, the main idea of sampling is to make draws from a bag of node labels, where the number of labels is equal to the out-degrees. At step $t+1$, generate a random variable $U \sim \text{Unif}(0, \sum_{v_j \in V(t)} \theta_1(v_j, t))$, then the source node (for the new edge) is randomly drawn from the bag if $U \leqslant \sum_{v_j \in V(t)} \text{O}(v_j, t)$. Otherwise, the source node is uniformly drawn from all existing nodes (regardless of their out-degrees). The sampling at each step has complexity $O(1)$, thus giving complexity $O(n)$ for the entire network generation process. The sampling of target nodes can be done in an analogous manner, and we call this approach `bag` in our package.

This idea can be generalized to weighted networks. At step $t+1$, the source preference score of node v_j is

$$\sum_{k:(v_j,v_k,w_{jk}) \in E(t)} w_{jk} + a_5,$$

where total source preference of all existing nodes in the network is

$$\sum_{v_j \in V(t)} \big(\text{O}(v_j, t) + a_5\big) := W(t) + a_5 \, |V(t)|,$$

where $W(t)$ is the total weight and $|V(t)|$ is the cardinality of $V(t)$. The sampling process proceeds as follows:

(1) At step $t+1$, create a vector, $\boldsymbol{\nu}(t)$, of cumulative sum of edge weights (according to the emergence order of edges), where the first element is 0 and the last element is $W(t)$;
(2) Compute $\tau(t+1) = (W(t) + a_5|V(t)|)X$ for some random variable $X \sim \text{Unif}(0, 1)$, independent from the network generation process;
(3) If $\tau(t+1) > W(t)$, a node is randomly sampled from $V(t)$; otherwise, find an index ℓ such that $\tau(t+1) \in (\boldsymbol{\nu}_\ell(t), \boldsymbol{\nu}_{\ell+1}(t)]$, then select the source node of the edge corresponding to the ℓ-th addition in $\boldsymbol{\nu}(t)$.

The sampling becomes efficient if we apply the above approach to all steps simultaneously. Notice that edge weights are independently drawn from h, and they are also independent of

other network generation components. Hence, to efficiently generate networks after n steps of evolution, vector $\boldsymbol{\nu}(n)$ can be determined independently in advance. Moreover, given a generated list of edge scenario parameters (i.e., α, β and γ), $|V(t)|$ can be obtained for $1 \leqslant t \leqslant n$ as well. Therefore, we collect all information that we need for node sampling in the entire network generation process, i.e., $W(t)$ and $|V(t)|$ for $1 \leqslant t \leqslant n$, with complexity $O(n)$. It remains to find the exact interval covering $\tau(t+1)$ in $\boldsymbol{\nu}(t)$ (a subset of $\boldsymbol{\nu}(n)$) for $\tau(t+1) \leqslant W(t)$, which can be done efficiently using the `findInterval()` function (with time complexity $O(n \log n)$).

We wrap up the above sampling approach as `bagx` in our package. Although `bagx` is not as efficient as `bag` for generating unweighted, linear PA networks, it provides a competitive alternative to generating weighted PA networks compared with the standard algorithm.

B Advanced Customized Preference Functions

Users can define customized preference functions by utilizing `cppXPtr` from package *RcppXPrtUtils*. The returned (external) pointer, `XPtr`, can be passed to `spref` and/or `tpref`. For instance, we fix the target preference function as $f_2(x, y) = y + 1$, and set the source preference function to be

$$f_1(x, y) = \begin{cases} 1 & \text{if } x < 1; \\ x^2 & \text{if } 1 \leqslant x \leqslant 100; \\ 200(x - 50) & \text{otherwise.} \end{cases}$$

The corresponding codes are given as follows:

```
my_spref <- RcppXPtrUtils::cppXPtr(code =
  "double foo(double x, double y) {
    if (x < 1) {
      return 1;
    } else if (x <= 100) {
      return pow(x, 2);
    } else {
      return 200 * (x - 50);
    }
  }")
ctr7 <- rpa_control_preference(ftype = "customized", spref = my_spref,
  tpref = "ins + 1")
```

External pointers cannot be shared across different R sessions. Therefore, we recommend that users save the source code of the customized preference functions for recompilation.

References

Abbasi A, Hossain L, Leydesdorff L (2012). Betweenness centrality as a driver of preferential attachment in the evolution of research collaboration networks. *Journal of Informatics*, 6(3): 403–412.

Atwood J, Ribeiro B, Towsley D (2015). Efficient network generation under general preferential attachment. *Computational Social Networks*, 2(1): 7. https://doi.org/10.1186/s40649-015-0012-9

Barabási AL, Albert R (1999). Emergence of scaling in random networks. *Science*, 286(5439): 509–512. https://doi.org/10.1126/science.286.5439.509

Barrat A, Barthélemy M, Vespignani A (2004). Weighted evolving networks: Coupling topology and weight dynamics. *Physical Review Letters*, 92(22): 228701. https://doi.org/10.1103/PhysRevLett.92.228701

Bollobás B, Borgs C, Chayes J, Riordan O (2003). Directed scale-free graphs. In: *SODA'03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 132–139. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Britton T (2020). Directed preferential attachment models: Limiting degree distributions and their tails. *Journal of Applied Probability*, 57(1): 122–136. https://doi.org/10.1017/jpr.2019.80

Capocci A, Servedio VDP, Colaiori F, Buriol LS, Donato D, Leonardi S, et al. (2006). Preferential attachment in the growth of social networks: The internet encyclopedia Wikipedia. *Physical Review E*, 74(3): 036116. https://doi.org/10.1103/PhysRevE.74.036116

Csardi G, Nepusz T (2006). The *igraph* software package for complex network research. *InterJournal, Complex Systems*, 1695.

Dong X, Castro L, Shaikh N (2020). *fastnet*: An R package for fast simulation and analysis of large-scale social networks. *Journal of Statistical Software*, 96(7): 1–23. https://doi.org/10.18637/jss.v096.i07

Eddelbuettel D, François R (2011). *Rcpp*: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8): 1–18. https://doi.org/10.18637/jss.v040.i08

Erdös P, Rényi A (1959). On random graphs I. *Publicationes Mathematicae Debrecen*, 6: 290–297. https://doi.org/10.5486/PMD.1959.6.3-4.12

Gilbert EN (1959). Random graphs. *Annals of Mathematical Statistics*, 30(4): 1141–1144. https://doi.org/10.1214/aoms/1177706098

Hadian A, Nobari S, Minaei-Bidgoli B, Qu Q (2016). ROLL: Fast in-memory generation of gigantic scale-free networks. In: *SIGMOD'16: Proceedings of the 2016 International Conference on Management of Data*, 1829–1842. Association for Computing Machinery, New York, NY, USA.

Hagberg AA, Schult DA, Swart PJ (2008). Exploring network structure, dynamics, and function using *NetworkX*. In: *Proceedings of the 7th Python in Science Conferencee (SciPy 2008)*, 11–15. Pasadena, CA, USA.

Kong JS, Sarshar N, Roychowdhury VP (2008). Experience versus talent shapes the structure of the web. *Proceedings of the National Academy of Sciences*, 105(37): 13724–13729. https://doi.org/10.1073/pnas.0805921105

Mahmoud HM (1992). *Evolution of Random Search Trees*. John Wiley & Sons, Hoboken, NJ, USA.

Mahmoud HM (2008). *Pólya Urn Models*. CRC Press, Boca Raton, FL, USA.

Momeni N, Rabbat MG (2015). Measuring the generalized friendship paradox in networks with quality-dependent connectivity. In: *Complex Networks VI: Proceedings of the 6th Workshop on Complex Networks CompleNet 2015* (G Mangioni, F Simini, SM Uzzo, D Wang, eds.), 45–55. Springer-Verlag.

Pham T, Sheridan P, Shimodaira H (2020). *PAFit*: An R package for the non-parametric estimation of preferential attachment and node fitness in temporal complex networks. *Journal of Statistical Software*, 92(3): 1–30. https://doi.org/10.18637/jss.v092.i03

Ucar I (2022). *RcppXPtrUtils*: XPtr Add-Ons for '*Rcpp*'. R package version 0.1.2.

Wan P, Wang T, Davis RA, Resnick SI (2017). Fitting the linear preferential attachment model.

Electronic Journal of Statistics, 11(2): 3738–3780.

Wang T, Resnick SI (2022a). Asymptotic dependence of in-and out-degrees in a preferential attachment model with reciprocity. *Extremes*, 25(3): 417–450. https://doi.org/10.1007/s10687-022-00439-5

Wang T, Resnick SI (2022b). Random networks with heterogeneous reciprocity. ArXiv e-prints.

Wang T, Resnick SI (2023). Poisson edge growth and preferential attachment networks. *Methodology and Computing in Applied Probability*, 25(1): 8. https://doi.org/10.1007/s11009-023-09997-y

Wang T, Yan J, Yuan Y, Zhang P (2022). Generating directed networks with predetermined assortativity measures. *Statistics and Computing*, 32(5): 91. https://doi.org/10.1007/s11222-022-10161-8

Yuan Y, Wang T, Yan J, Zhang P (2023). *wdnet*: Weighted and Directed Networks. University of Connecticut. R package version 1.1.1.

Yuan Y, Yan J, Zhang P (2021). Assortativity measures for weighted and directed networks. *Journal of Complex Networks*, 9(2): cnab017. https://doi.org/10.1093/comnet/cnab017

Zhang P, Wang T, Yan J (2022). PageRank centrality and algorithms for weighted, directed networks. *Physica A: Statistical Mechanics and its Applications*, 586: 126438.