Tutorial on TreeSampleR

Summary

TreeSampleR is an R package that implements fast algorithms for sampling random spanning trees.

In particular, it provides fast C++ implementations of three random spanning tree samplers:

- The Aldous-Broder sampler
- The Wilson Sampler
- The Fast-forwarded cover sampler of Tam, Dunson and Duan (2024).

All function outputs are in standard adjacency matrix format, so users can easily transfer results to other environments for analysis.

Background Knowledge

A spanning tree of a given weighted, undirected graph $G = (V, E, w)$ is a subgraph of G that 1. is a tree and 2. spans all the nodes. It is a structure that maintains a small number of edges while still maintaining connectivity. Researchers are interested in sampling random spanning trees of a given graph for a broad range of statistical applications, such as networking sampling, dependency modeling in graphical models, Bayesian modeling, spatial statistics, graph signal processing etc.

A random spanning tree T has probability mass function $P(T) \propto \prod_e w_e$, where the probability assigned to the tree is proportional to the product of its edge weights. Uniform spanning trees is a special case of this distribution when all the edges have the same positive weight.

All 3 samplers that we implement gives exact samples of $P(T)$.

Usage

For all of our implemented algorithms to work, we require that:

- The graph is connected
- The graph has non-negative edge weights
- The graph does not have self loops
- The graph is undirected

In practice we have tests that will check these conditions for you and throw and error if they are not met.

Certain algorithms can be generalized to the directed case in theory, but this is not supported in our implementation.

Let's see a toy example:

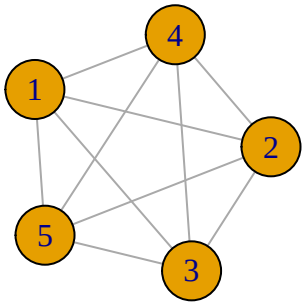
```
library(TreeSampleR)
library(igraph)
```

```
Attaching package: 'igraph'
```

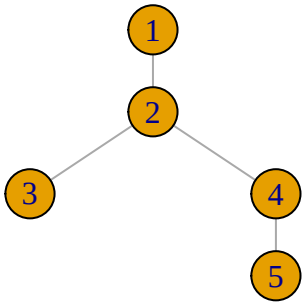
```
The following objects are masked from 'package:stats':

    decompose, spectrum
```

```
The following object is masked from 'package:base':

    union
```

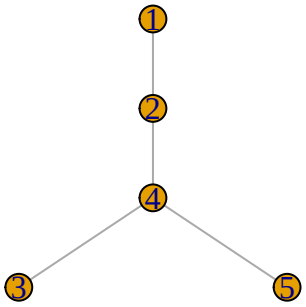
```
# generate a complete graph on 10 vertices
A = matrix(rep(1, 25), 5, 5)
diag(A) = rep(0, 5)

# visualize it
g <- graph_from_adjacency_matrix(A, "undirected")
plot_underlying(g)
```



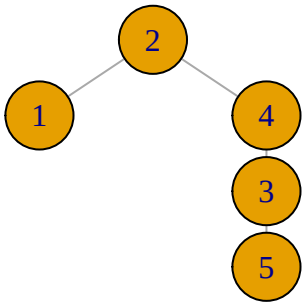

```
set.seed(1000)
# Let's generate some random spanning trees using all 3 functions

A1 <- aldous_broder(A, 1)
A2 <- fast_cover(A, 1, 2)
A3 <- wilson(A, 1)

# Visualize the trees

tree1 <- graph_from_adjacency_matrix(A1)
tree2 <- graph_from_adjacency_matrix(A2)
tree3 <- graph_from_adjacency_matrix(A3)

plot_as_tree(tree1, vertex_size = 40)
```




```
plot_as_tree(tree2, vertex_size = 20)
```



```r
plot_as_tree(tree3, vertex_size = 60, root = "2")
```




```r
## as you can observe, you can select different "roots" to plot the trees
## you can also change the size of the vertices in the plot for easier visualization
```

The choice of starting point does not affect the sampling properties of these samplers on undirected graphs.

Fast-forward cover sampler

All of the algorithms above are based on running a random walk on the underlying graph.

Sometimes, if there are bottlenecks in the underlying graph, then the random walk could get stuck in certain components for a very long time, leading to slow sample generation. This issue can occur for classical algorithms like the Aldous-Broder and Wilson's sampler.

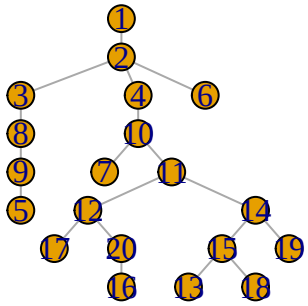
The fast-forwarded cover sampler is designed to tackle this issue. Everytime you get stuck for more than κ steps in the random walk without visiting a priorly unvisited node, a fast-forwarding step is deployed to break free of the bottleneck.

The choice of κ does not affect the sampling properties, only computational speed. You can try different choice of κ to see which one works fastest. If you have prior knowledge of clusters

of the underlying graph, and you know that the largest cluster has size K, a reasonable choice of κ can be $K \log K$.

```
## Generate graph with bottleneck structure

num_vert = 10
A_1<- matrix(rep(1, num_vert*num_vert), num_vert, num_vert)
A_2<- matrix(rep(1, num_vert*num_vert), num_vert, num_vert)

A <- matrix(0, 2*num_vert, 2*num_vert)
A[1:num_vert, 1:num_vert] <- A_1
A[(num_vert + 1):(2*num_vert), (num_vert + 1):(2*num_vert)] <- A_2
diag(A) <- 0

edge1 = c(num_vert, num_vert + 1)
edge2 = c(num_vert %/% 2, num_vert + num_vert %/% 2)

## Bottleneck edge
A[edge1[1], edge1[2]] <- 0.01
A[edge1[2], edge1[1]] <- 0.01

## Bottleneck edge
A[edge2[1], edge2[2]] <- 0.01
A[edge2[2], edge2[1]] <- 0.01
```

```
## apply fast-forwarded cover sampler
## since we know cluster size is 10,
## we choose threshold as roughly 10 * log 10 = 10.
A4 = fast_cover(A, 1, 10)
tree4 = graph_from_adjacency_matrix(A4)
plot_as_tree(tree4, vertex_size = 20)
```


1
2
3
4
6
8
10
9
7
11
5
12
14
17
20
15
19
16
13
18