Package: gor (via r-universe)

August 30, 2024

Title Algorithms for the Subject Graphs and Network Optimization

Version 1.0

Description Informal implementation of some algorithms from Graph Theory and Combinatorial Optimization which arise in the subject ``Graphs and Network Optimization" from first course of the EUPLA (Escuela Universitaria Politecnica de La Almunia) degree of Data Engineering in Industrial Processes. References used are: Cook et al (1998, ISBN:0-471-55894-X), Korte, Vygen (2018) <doi:10.1007/978-3-662-56039-6>, Hromkovic (2004) <doi:10.1007/978-3-662-05269-3>, Hartmann, Weigt (2005, ISBN:978-3-527-40473-5).

License GPL-3

Encoding UTF-8

RoxygenNote 7.2.3

Depends igraph, graphics, stats

NeedsCompilation no

Author Cesar Asensio [aut, cre] (<https://orcid.org/0000-0002-7538-1501>)

Maintainer Cesar Asensio <casencha@unizar.es>

Repository CRAN

Date/Publication 2023-05-03 18:40:02 UTC

Contents

apply_incidence_map *Apply incidence map of a graph to an edge vector*

Description

Apply incidence map of a graph to an edge vector. It uses the edgelist of the graph instead of the incidence matrix.

Usage

```
apply_incidence_map(eG, v)
```

Arguments

eG	Graph in edgelist representation, see as_edgelist.
v	Edge vector to which the incidence map will be applied.

Details

The incidence map is the linear transformation from the edge vector space to the vertex vector space of a graph associating to each edge its incident vertices. It is customarily represented by the incidence matrix, which is a very large matrix for large graphs; for this reason it not efficient to use directly the incidence matrix. This function uses the edgelist of the graph as returned by the as_edgelist function to compute the result of the incidence map on an edge vector, which is interpreted with respect to the same edgelist.

Value

A vertex vector, having the degree of each vertex in the subgraph specified by the edge vector.

Author(s)

Cesar Asensio

See Also

shave_cycle, for shaving hairy cycles, which makes use of this routine, and generate_fundamental_cycles, using the former.

Examples

```
g <- make_graph("Dodecahedron")
eG <- as_edgelist(g)
set.seed(1)
v <- sample(0:1, gsize(g), replace = TRUE) # Random edge vector
apply_incidence_map(eG, v) # 1 1 0 1 2 0 1 1 3 2 0 1 1 1 1 1 0 0 1 2
## Plotting the associated subgraph
h <- make_graph(t(eG[v==1,]))
```

```
z <- layout_with_gem(g)
plot(g, layout = z)
plot(h, layout = z, add = TRUE, edge.color = "red3", edge.width = 3)
```

---

bfs_tree *Breadth-first search tree*

Description

Computation of the breadth-first tree search in an undirected graph.

Usage

```
bfs_tree(g, r)
```

Arguments

g	Graph
r	Root: Starting vertex growing the tree.

Details

Starting from a root vertex, the tree is grown by adding neighbors of the first vertex added to the tree until no more neighbors are left; then it passes to another vertex with neighbors outside the tree. In this way, the tree has few levels and many branches and leaves.

Value

A directed spanning subgraph of g containing the edges of the BFS tree.

Author(s)

Cesar Asensio

Examples

```
g <- make_graph("Frucht")
T <- bfs_tree(g, 2)  # Root at v = 2
z <- layout_with_gem(g)
plot(g, layout = z, main = "Breadth-first search tree")
plot(T, layout = z, add = TRUE, edge.color = "cyan4", edge.width = 2)
plot(T, layout = layout_as_tree(T))
```

`build_cover_approx` *2-approximation algorithm for vertex cover*

Description

Gavril's 2-approximation algorithm to build a vertex cover.

Usage

```
build_cover_approx(G)
```

Arguments

`G`	Graph

Details

This algorithm computes a maximal matching and takes the ends of the edges in the matching as a vertex cover. No edge is uncovered by this vertex subset, or the matching would not be maximal; therefore, the vertex set thus found is indeed a vertex cover.

Since no vertex can be incident to two edges of a matching M, at least |M| vertices are needed to cover the edges of the matching; thus, any vertex cover X should satisfy |X| >= |M|. Moreover, the vertices incident to the matching are always a vertex cover, which implies that, if X* is a vertex cover of minimum sise, |X*| <= 2|M|.

Value

A list with two components: $set contains the cover, $size contains the number of vertices of the cover.

Author(s)

Cesar Asensio

References

Korte, Vygen *Combinatorial Optimization. Theory and Algorithms.*

See Also

is_cover checks if a vertex subset is a vertex cover, build_cover_greedy builds a cover using a greedy heuristic, improve_cover_flip improves a cover using local search, search_cover_random looks for a random cover of fixed size, search_cover_ants looks for a random cover using a version of the ant-colony optimization heuristic, find_cover_BB finds covers using a branch-and-bound technique, plot_cover plots a cover.

Examples

```
## Example with known vertex cover
K25 <- make_full_graph(25)   # Cover of size 24
X0 <- build_cover_approx(K25)
X0$size  # 24
plot_cover(X0, K25)

## Vertex-cover of a random graph
set.seed(1)
n <- 25
g <- sample_gnp(n, p=0.25)
X2 <- build_cover_approx(g)
X2$size   # 20
plot_cover(X2, g)
```

`build_cover_greedy` *Greedy algorithm for vertex cover in a graph*

Description

This routine uses a greedy algorithm to build a cover selecting the highest degree vertex first and removing its incident edges.

Usage

```
build_cover_greedy(G)
```

Arguments

`G`	Graph

Details

This algorithm builds a vertex cover since no edge remains to be covered when it returns. However, it is no guaranteed that the cover found by this algorithm has minimum cardinality.

Value

A list with two components: $set contains the cover, $size contains the number of vertices of the cover.

Author(s)

Cesar Asensio

References

Korte, Vygen *Combinatorial Optimization. Theory and Algorithms.*

See Also

is_cover checks if a vertex subset is a vertex cover, build_cover_approx builds a cover using a 2-approximation algorithm, improve_cover_flip improves a cover using local search, search_cover_random looks for a random cover of fixed size, search_cover_ants looks for a random cover using a version of the ant-colony optimization heuristic, find_cover_BB finds covers using a branch-and-bound technique, plot_cover plots a cover.

Examples

```
## Example with known cover
K25 <- make_full_graph(25)   # Cover of size 24
X0 <- build_cover_greedy(K25)
X0$size  # 24
plot_cover(X0, K25)
plot_cover(list(set = c(1,2), size = 2), K25)

## Vertex-cover of a random graph
set.seed(1)
n <- 25
g <- sample_gnp(n, p=0.25)
X1 <- build_cover_greedy(g)
X1$size   # 17
plot_cover(X1, g)
```

---

build_cover_random *Random vertex covers*

Description

Random algorithm for vertex-cover.

Usage

```
build_cover_random(G, N, p = 0.75)
```

Arguments

G	Graph.
N	Number of random vertex set to try.
p	Probability of each element to be selected.

Details

It builds N random vertex sets by inserting elements with probability p, and it verifies if the subset so chosen is a vertex cover by running is_cover on it. It is very difficult to find a good vertex cover in this way, so this algorithm is very inefficient and it finds no specially good covers.

Currently, this function is *not* exported. The random sampling performed by search_cover_random is faster and more efficient.

Value

A list with four components: $set contains the subset of V(g) representing the cover and $size contains the number of vertices of the cover; $found is the number of vertex covers found and $failed is the number of generated subset that were not vertex covers.

Author(s)

Cesar Asensio

Examples

```
n <- 25
g <- sample_gnp(n, p=0.25)  # Random graph
X5 <- build_cover_random(g,10000,p=0.65)
X5$size            # 19
plot_cover(X5, g)
X6 <- improve_cover_flip(g, X5)   # Improved : 17
plot_cover(X6, g)
```

---

build_cut_greedy *Greedy algorithm aimed to build a large weight cut in a graph*

Description

This routine uses a greedy algorithm to build a cut with large weight. This is a 2-approximation algorithm, which means that the weight of the cut returned by this algorithm is larger than half the maximum possible cut weight for a given graph.

Usage

```
build_cut_greedy(G, w = NA)
```

Arguments

G	Graph
w	Weight matrix (defaults to NA). It should be zero for those edges not in G

Details

The algorithm builds a vertex subset S a step a a time. It starts with S = c(v1), and with vertices v1 and v2 marked. Then it iterates from vertex v3 to vn checking if the weight of the edges joining vi with marked vertices belonging to S is less than the weight of the edges joining vi with marked vertices not belonging to S. If the former weight is less than the latter, then vi is adjoined to S. At the end of each iteration, vertex vi is marked. When all vertices are marked the algorithm ends and S is already built.

Value

A list with four components: $set contains the subset of V(g) representing the cut, $size contains the number of edges of the cut, $weight contains the weight of the cut (which coincides with $size if w is NA) and $cut contains the edges of the cut, joining vertices inside $set with vertices outside $set.

Author(s)

Cesar Asensio

References

Korte, Vygen *Combinatorial Optimization. Theory and Algorithms.*

See Also

build_cut_random builds a random cut, improve_cut_flip uses local search to improve a cut obtained by other methods, compute_cut_weight computes cut size, weight and edges, plot_cut plots a cut.

Examples

```
## Example with known maximum cut
K10 <- make_full_graph(10)   # Max cut of size 25
c0 <- build_cut_greedy(K10)
c0$size  # 25
plot_cut(c0, K10)

## Max-cut of a random graph
set.seed(1)
n <- 25
g <- sample_gnp(n, p=0.25)
c2 <- build_cut_greedy(g)
c2$size   # 59
plot_cut(c2, g)
```

---

build_cut_random *Random cut generation on a graph*

Description

Random cut generation on a graph. This function generates a hopefully large cut on a graph by randomly selecting vertices; it does not attempt to maximize the cut size or weigth, so it is intended to be used as part of some smarter strategy.

Usage

```
build_cut_random(G, w = NA)
```

Arguments

G	Graph
w	Weight matrix (defaults to NA). It should be zero for those edges not in G

Details

It selects a random subset of the vertex set of the graph, computing the associated cut, its size and its weigth, provided by the user as a weight matrix. If the weight argument w is NA, the weights are taken as 1.

Value

A list with four components: $set contains the subset of V(g) representing the cut, $size contains the number of edges of the cut, $weight contains the weight of the cut (which coincides with $size if w is NA) and $cut contains the edges of the cut, joining vertices inside $set with vertices outside $set.

Author(s)

Cesar Asensio

See Also

build_cut_greedy builds a cut using a greedy algorithm, compute_cut_weight computes cut size, weight and edges, improve_cut_flip uses local search to improve a cut obtained by other methods, plot_cut plots a cut.

Examples

```
## Example with known maximum cut
K10 <- make_full_graph(10)   # Max cut of size 25
c0 <- build_cut_random(K10)
c0$size  # Different results: 24, 21, ...
plot_cut(c0, K10)

## Max-cut of a random graph
set.seed(1)
n <- 25
g <- sample_gnp(n, p=0.25)
c1 <- build_cut_random(g)   # Repeat as you like
c1$size  # Different results: 43, 34, 39, 46, 44, 48...
plot_cut(c1, g)
```

`build_tour_2tree` *Double-tree heuristic for TSP*

Description

Double-tree heuristic tour-building algorithm for the Traveling Salesperson Problem

Usage

```
build_tour_2tree(d, n, v0 = 1)
```

Arguments

`d`	Distance matrix defining the TSP instance
`n`	Number of cities to consider with respect to the distance matrix
`v0`	Initial vertex to find the eulerian walk; it defaults to 1.

Details

The **double-tree** heuristic is a 2-factor approximation algorithm which begins by forming a minimum distance spanning tree, then it forms the double-tree by doubling each edge of the spanning tree. The double tree is Eulerian, so an Eulerian walk can be computed, which gives a well-defined order of visiting the cities of the problem, thereby yielding the tour.

In practice, this algorithm performs poorly when compared with another simple heuristics such as nearest-neighbor or insertion methods.

Value

A list with two components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, and $distance contains the value of the distance covered by the tour.

Author(s)

Cesar Asensio

See Also

build_tour_nn uses the nearest heighbor heuristic, build_tour_nn_best repeats the previous algorithm with all possible starting points, compute_tour_distance computes tour distances, compute_distance_matrix computes a distance matrix, plot_tour plots a tour, find_euler finds an Eulerian walk.

Examples

```
## Regular example with obvious solution (minimum distance 48)
m <- 10   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
b <- build_tour_2tree(d, n)
b$distance    # Distance 57.86
plot_tour(z,b)

## Random points
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
b <- build_tour_2tree(d, n)
b$distance    # Distance 48.63
plot_tour(z,b)
```

`build_tour_greedy` *Building a tour for a TSP using the greedy heuristic*

Description

Greedy heuristic tour-building algorithm for the Traveling Salesperson Problem

Usage

```
build_tour_greedy(d, n)
```

Arguments

`d`	Distance matrix of the TSP.
`n`	Number of vertices of the TSP complete graph.

Details

The greedy heuristic begins by sorting the edges by increasing distance. The tour is constructed by adding an edge under the condition that the final tour is a connected spanning cycle.

Value

A list with two components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, and $distance contains the value of the distance covered by the tour.

Author(s)

Cesar Asensio

See Also

build_tour_nn uses the nearest heighbor heuristic, build_tour_nn_best repeats the previous algorithm with all possible starting points, compute_tour_distance computes tour distances, compute_distance_matrix computes a distance matrix, plot_tour plots a tour, build_tour_2tree constructs a tour using the double tree 2-factor approximation algorithm.

Examples

```
## Regular example with obvious solution (minimum distance 48)
m <- 10   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
b <- build_tour_greedy(d, n)
b$distance    # Distance 50
plot_tour(z,b)

## Random points
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
b <- build_tour_greedy(d, n)
b$distance    # Distance 36.075
plot_tour(z,b)
```

---

build_tour_nn *Building a tour for a TSP using the nearest neighbor heuristic*

Description

Nearest neighbor heuristic tour-building algorithm for the Traveling Salesperson Problem

Usage

```
build_tour_nn(d, n, v0)
```

Arguments

d	Distance matrix of the TSP.
n	Number of vertices of the TSP complete graph.
v0	Starting vertex. Valid values are integers between 1 and n.

Details

Starting from a vertex, the algorithm takes its nearest neighbor and incorporates it to the tour, repeating until the tour is complete. The result is dependent of the initial vertex. This algorithm is very efficient but its output can be very far from the minimum.

Value

A list with two components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, and $distance contains the value of the distance covered by the tour.

Author(s)

Cesar Asensio

See Also

build_tour_nn_best repeats this algorithm with all possible starting points, compute_tour_distance computes tour distances, compute_distance_matrix computes a distance matrix, plot_tour plots a tour, build_tour_greedy constructs a tour using the greedy heuristic.

Examples

```
## Regular example with obvious solution (minimum distance 48)
m <- 10   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
b <- build_tour_nn(d, n, 1)
b$distance    # Distance 50
plot_tour(z,b)
b <- build_tour_nn(d, n, 5)
b$distance    # Distance 52.38
plot_tour(z,b)

## Random points
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
b <- build_tour_nn(d, n, 1)
b$distance    # Distance 46.4088
plot_tour(z,b)
b <- build_tour_nn(d, n, 9)
b$distance    # Distance 36.7417
plot_tour(z,b)
```

---

build_tour_nn_best *Build a tour for a TSP using the best nearest neighbor heuristic*

Description

Nearest neighbor heuristic tour-building algorithm for the Traveling Salesperson Problem - Better starting point

Usage

```
build_tour_nn_best(d, n)
```

Arguments

d	Distance matrix of the TSP.
n	Number of vertices of the TSP complete graph.

Details

It applies the nearest neighbor heuristic with all possible starting vertices, retaining the best tour returned by build_tour_nn.

Value

A list with four components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, $distance contains the value of the distance covered by the tour, $start contains the better starting vertex found, and $Lall contains the distances found by starting from each vertex.

Author(s)

Cesar Asensio

See Also

build_tour_nn nearest neighbor heuristic with a single starting point, compute_tour_distance computes tour distances, compute_distance_matrix computes a distance matrix, plot_tour plots a tour.

Examples

```
## Regular example with obvious solution (minimum distance 48)
m <- 10   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
b <- build_tour_nn_best(d, n)
b$distance    # Distance 48.6055
b$start       # Vertex 12
plot_tour(z,b)

## Random points
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
b <- build_tour_nn_best(d, n)
b$distance    # Distance 36.075
b$start       # Vertex 13
plot_tour(z,b)
```

color_graph_greedy *Greedy coloring of a graph*

Description

Greedy algorithm for coloring the vertices of a graph

Usage

```
color_graph_greedy(g, ord = NULL, ran = FALSE)
```

Arguments

g	Graph to be colored
ord	Specified vertex ordering or NULL if natural vertex ordering is preferred
ran	Choose random vertex ordering; it defaults to FALSE. It is ignored if ord is non-NULL

Details

"Colors" are integers from 1 to the order of the graph to be colored. The greedy strategy assigns to each vertex v the least color not assigned to the neighbors of v.

Value

Vertex colors in a vector, with component i being the (integer) color of vertex i.

Author(s)

Cesar Asensio

Examples

```
library(igraph)
g <- make_graph("Petersen")
cg <- color_graph_greedy(g)
plot(g, vertex.color = rainbow(6)[cg])
max(cg)         # = 3: Number of colors used by the algorithm
sum(g[cg == 1, cg == 1])  # = 0: Color classes are stable sets

g <- make_graph("Dodecahedron")
cg <- color_graph_greedy(g)
plot(g, vertex.color = rainbow(6)[cg])
max(cg)   # = 4: Number of colors used by the algorithm
sum(g[cg == 1, cg == 1])  # = 0: Color classes are stable sets

## However, the dodecahedron has a 3-coloring:
cdod <- rep(1, 20)
```

```
cdod[c(1,3,7,12,13,20)] <- 2
cdod[c(5,8,9,11,17,19)] <- 3
plot(g, vertex.color = rainbow(6)[cdod])
sum(g[cdod == 1, cdod == 1]) # = 0
sum(g[cdod == 2, cdod == 2]) # = 0
sum(g[cdod == 3, cdod == 3]) # = 0

## Some vertex orderings can use less colors:
cg <- color_graph_greedy(g, ord = 20:1)
plot(g, vertex.color = rainbow(6)[cg])
max(cg)         # = 3: Number of colors used by the algorithm
```

`compute_cut_weight` *Compute cut weight and size*

Description

Compute cut weight and size from its associated vertex set. It can also return the edges in the cut.

Usage

```
compute_cut_weight(S, n, eG, w = NA, return.cut = FALSE)
```

Arguments

`S`	Subset of the vertex set of the graph.
`n`	Size of the graph.
`eG`	Edgelist of the graph as returned by as_edgelist, that is, a matrix with q rows and 2 columns. Note that this is the graph format used by the routine.
`w`	Weight matrix or NA if all edge weights are 1. It should be zero for those edges not in G
`return.cut`	Boolean. Should the routine return the edges in the cut? It defaults to FALSE. When TRUE, the routine also returns the input subset S, for easier cut plotting with plot_cut.

Details

In a graph, a cut K is defined by means of a vertex subset S as the edges joining vertices inside S with vertices outside S. This routine computes these edges and their associated weight.

Value

A list with two components: $size is the number of edges in the cut, $weight is the weight of the cut, that is, the sum of the weights of the edges in the cut. If w=NA these two numbers coincide. When return.cut is TRUE, there are two additional components of the list: $cut, which contains the edges in the cut as rows of a two-column matrix, and $set, which contains the input set, as a convenience for plotting with plot_cut.

Author(s)

Cesar Asensio

See Also

build_cut_random builds a random cut, build_cut_greedy builds a cut using a greedy algorithm, improve_cut_flip uses local search to improve a cut obtained by other methods, plot_cut plots a cut.

Examples

```
K10 <- make_full_graph(10)
S <- c(1,4,7)
compute_cut_weight(S, gorder(K10), as_edgelist(K10))
cS <- compute_cut_weight(S, gorder(K10), as_edgelist(K10),
    return.cut = TRUE)
plot_cut(cS, K10)
```

---

compute_distance_matrix

p-distance matrix computation

Description

It computes the distance matrix of a set of n two-dimensional points given by a $n \times 2$ matrix using the distance-p with $p = 2$ by default.

Usage

```
compute_distance_matrix(z, p = 2)
```

Arguments

z	A $n \times 2$ matrix with the two-dimensional points
p	The p parameter of the distance-p. It defaults to 2.

Details

Given a set of n points $\{z_j\}_{j=1,\ldots,n}$, the distance matrix is a $n \times n$ symmetric matrix with matrix elements

$$d_{ij} = d(z_i, z_j)$$

computed using the distance-p given by

$$d_p(x, y) = \left(\sum_i (x_i - y_i)^p \right)^{\frac{1}{p}}$$

.

Value

The distance-p of the points.

Author(s)

Cesar Asensio

See Also

compute_p_distance computes the distance-p, compute_tour_distance computes tour distances. A distance matrix can also be computed using dist.

Examples

```
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
```

---

compute_gain_transp *Distance gain when transposing two cities in a tour*

Description

Distance gain when two cities in a TSP tour are interchanged, that is, the neighbors of the first become the neighbors of the second and vice versa. It is used to detect favorable moves in a Lin-Kernighan-based routine for the TSP.

Usage

```
compute_gain_transp(C, tr, d)
```

Arguments

C	Tour represented as a non-repeated vertex sequence. Equivalently, a permutation of the sequence from 1 to length(C).
tr	Transposition, represented as a pair of indices between 1 and length(C).
d	Distance matrix.

Details

It computes the gain in distance when interchanging two cities in a tour. The transformation is akin to a 2-interchange; in fact, if the transposed vertices are neighbors in the tour or share a common neighbor, the transposition is a 2-interchange. If the transposed vertices in the tour do not share any neighbors, then the transposition is a pair of 2-interchanges.

This gain is used in improve_tour_LinKer, where the transposition neighborhood is used instead of the variable k-opt neighborhood for simplicity.

Value

The gain in distance after performing transposition tr in tour C with distance matrix d.

Author(s)

Cesar Asensio

See Also

improve_tour_LinKer, a where this function is used.

Examples

```
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
compute_gain_transp(sample(n),c(4,23),d)  # -6.661
compute_gain_transp(sample(n),c(17,3),d)  #  4.698
```

`compute_lower_bound_1tree`

Computing the 1-tree lower bound for a TSP instance

Description

It computes the 1-tree lower bound for an optimum tour for a TSP instance.

Usage

```
compute_lower_bound_1tree(d, n, degree = FALSE)
```

Arguments

d	Distance matrix.
n	Number of vertices of the TSP complete graph.
degree	Boolean: Should the routine return the degree seguence of the internal minimum spanning tree? Defaults to FALSE.

Details

It computes the 1-tree lower bound for an optimum tour for a TSP instance from vertex 1. Internally, it creates the graph Kn-v1 and invokes mst from package igraph to compute the minimum weight spanning tree. If optional argument "degree" is TRUE, it returns the degree seguence of this internal minimum spanning tree, which is very convenient when embedding this routine in the Held-Karp lower bound estimation routine.

Value

The 1-tree lower bound – A scalar if the optional argument "degree" is FALSE. Otherwise, a list with the previous 1-tree lower bound in the $bound component and the degree sequence of the internal minimum spanning tree in the $degree component.

Author(s)

Cesar Asensio

See Also

improve_tour_2opt tour improving using 2-opt, improve_tour_3opt tour improving using 3-opt, compute_lower_bound_HK for Held-Karp lower bound estimates.

Examples

```
m <- 10   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
b <- build_tour_2tree(d, n)
b$distance           # Distance 57.868
bi <- improve_tour_2opt(d, n, b$tour)
bi$distance          # Distance 48 (optimum)
compute_lower_bound_1tree(d,n)       # 45

## Random points
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
compute_lower_bound_1tree(d,n)       # 31.4477
bn <- build_tour_nn_best(d,n)
b3 <- improve_tour_3opt(d,n,bn$tour)
b3$distance                          # 35.081
```

---

compute_lower_bound_HK

Held-Karp lower bound estimate

Description

Held-Karp lower bound estimate for the minimum distance of an optimum tour in the TSP

Usage

```
compute_lower_bound_HK(d, n, U, tsmall = 0.001, it = 0.2 * n, block = 100)
```

Arguments

d	Distance matrix of the TSP instance.
n	Number of vertices of the TSP complete graph.
U	Upper bound of the minimum distance.
tsmall	Stop criterion: Is the step size decreases beyond this point the algorithm stops. Defaults to 0.001.
it	Iterates inside each block. Some experimentation is required to adjust this parameter: If it is large, the run time will be larger; if it is small, the accuracy will decrease.
block	Number of blocks of "it" iterations. In each block the size of the multiplier is halved.

Details

An implementation of the Held-Karp iterative algorithm towards the minimum distance tour of a TSP instance. As the algorithm converges slowly, only an estimate will be achieved. The accuracy of the estimate depends on the stopping requirements through the number of iteration blocks "block" and the number of iterations per block "it", as well as the smallest allowed multiplier size "tsmall": When it is reached the algorithm stops. It is also crucial to a good estimate the quality of the upper bound "U" obtained by other methods.

The Held-Karp bound has the following uses: (1) assessing the quality of solutions not known to be optimal; (2) giving an optimality proof of a given solution; and (3) providing the "bound" part in a branch-and-bound technique.

Please note that recommended computation of the Held-Karp bound uses Lagrangean relaxation on an integer programming formulation of the TSP, whereas this routine uses the Cook algorithm to be found in the reference below.

Value

An estimate of the Held-Karp lower bound – A scalar.

Author(s)

Cesar Asensio

References

Cook et al. *Combinatorial Optimization* (1998) sec. 7.3.

See Also

compute_distance_matrix computes distance matrix of a set of points, build_tour_nn_best builds a tour using the best-nearest-neighbor heuristic, improve_tour_2opt improves a tour using 2-opt, improve_tour_3opt improves a tour using 3-opt, compute_lower_bound_1tree computes the 1-tree lower bound.

Examples

```
m <- 10   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
b <- build_tour_nn_best(d, n)
b$distance           # Distance 48.6055
bi <- improve_tour_2opt(d, n, b$tour)
bi$distance          # Distance 48 (optimum)
compute_lower_bound_HK(d,n,U=48.61)                    # 45.927
compute_lower_bound_HK(d,n,U=48.61,it=20,tsmall=1e-6)  # 45.791

## Random points
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
bn <- build_tour_nn_best(d,n)
b3 <- improve_tour_3opt(d,n,bn$tour)
b3$distance                                            # 35.08155
compute_lower_bound_HK(d, n, U=35.1)                   # 34.80512
compute_lower_bound_HK(d, n, U=35.0816, it=20)         # 35.02892
compute_lower_bound_HK(d, n, U=35.0816, tsmall = 1e-5) # 34.81119
compute_lower_bound_HK(d, n, U=35.0816, it=50, tsmall = 1e-9)  # 35.06789
```

---

compute_path_distance *Compute the distance of a TSP path*

Description

It computes the distance covered by a path in a Traveling Salesman Problem

Usage

```
compute_path_distance(h, d)
```

Arguments

h	A path specified by a vertex sequence
d	Distance matrix to use

Details

This function simply add the distances in a distance matrix indicated by a vertex sequence defining a path. It takes into account that, in a path, the last vertex is **not** joined to the first one by an edge, unlike compute_tour_distance.

Value

The path distance

$$d(\{v_1, ..., v_n\}) = \sum_{j=1}^{n-1} d(v_j, v_{j+1}).$$

Author(s)

Cesar Asensio

See Also

build_tour_nn nearest neighbor heuristic with a single starting point, build_tour_nn_best repeats the previous algorithm with all possible starting points, compute_distance_matrix computes a distance matrix, compute_tour_distance computes tour distances, plot_tour plots a tour.

Examples

```
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
h <- sample(1:n)              # A random tour
compute_path_distance(h, d)   # 107.246
compute_tour_distance(h, d) - compute_path_distance(h, d) - d[h[1], h[n]]
```

---

compute_p_distance *Distance-p between two-dimensional points*

Description

It computes the distance-p between two-dimensional points.

Usage

```
compute_p_distance(x, y, p = 2)
```

Arguments

x	A two-dimensional point
y	A two-dimensional point
p	The p-parameter of the distance-p

Details

The distance-p is defined by

$$d_p(x, y) = \left(\sum_i (x_i - y_i)^p \right)^{\frac{1}{p}}$$

.

Value

The distance-p between points x and y.

Author(s)

Cesar Asensio

See Also

compute_distance_matrix computes the distance matrix of a set of two-dimensional points, compute_tour_distance computes tour distances.

Examples

```
compute_p_distance(c(1,2),c(3,4))      # 2.8284
compute_p_distance(c(1,2),c(3,4),p=1)  # 4
```

---

compute_tour_distance *Compute the distance of a TSP tour*

Description

It computes the distance covered by a tour in a Traveling Salesman Problem

Usage

```
compute_tour_distance(h, d)
```

Arguments

h	A tour specified by a vertex sequence
d	Distance matrix to use

Details

This function simply add the distances in a distance matrix indicated by a vertex sequence defining a tour. It takes into account that, in a tour, the last vertex is joined to the first one by an edge, and adds its distance to the result, unlike compute_path_distance.

Value

The tour distance

$$d(\{v_1, ..., v_n\}) = \sum_{j=1}^{n} d(v_j, v_{(jmodn)+1}).$$

Author(s)

Cesar Asensio

See Also

build_tour_nn nearest neighbor heuristic with a single starting point, build_tour_nn_best repeats the previous algorithm with all possible starting points, compute_distance_matrix computes a distance matrix, compute_path_distance computes path distances, plot_tour plots a tour.

Examples

```
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
h <- sample(1:n)              # A random tour
compute_tour_distance(h, d)   # 114.58
```

---

crossover_sequences *Crossover of sequences*

Description

Crossover sequence operation for use in the genetic cut-search algorithm.

Usage

```
crossover_sequences(s1, s2, cpoint = NA)
```

Arguments

s1	Sequence
s2	Sequence of the same lenght as s1
cpoint	Crossover point, an integer between 1 and length(s1)-1. Defaults to NA, in which case it will be randomly chosen

Details

This operation takes two sequences of the same lenght "n" and splits them in two at a crossover point between 1 and "n-1". Then it produces two "offsprings" by interchanging the pieces and gluing them together.

The crossover point can be specified in argument cpoint. By providing NA (the default), cpoint is chosen randomly.

Note that this crossover operation is the "classic" crossover included in the original genetic algorithm, and it is adequate when applied to binary sequences. However, when applied to permutations, the result of this function can have repeated elements; hence, it is not adequate for the TSP.

Value

A two-row matrix. Rows are the offsprings produced by the crossover

Author(s)

Cesar Asensio

See Also

search_cut_genetic genetic algorithm cut-search, mutate_binary_sequence binary sequence mutation

Examples

```
set.seed(1)
s1 <- sample(0:1, 10, replace = TRUE)
s2 <- sample(0:1, 10, replace = TRUE)
crossover_sequences(s1, s2)

set.seed(1)
s1 <- sample(1:10, 10)
s2 <- sample(1:10, 10)
crossover_sequences(s1, s2, cpoint = 5)
```

---

crossover_tours *Crossover operation used by the TSP genetic algorithm*

Description

Crossover operation used by the TSP genetic algorithm. It takes two tours and it computes two "offsprings" trying to exploit the structure of the cycles, see below.

Usage

```
crossover_tours(C1, C2, d, n)
```

Arguments

C1	Vertex numeric vector of the first parent tour.
C2	Vertex numeric vector of the second parent tour.
d	Distance matrix of the TSP instance. It is used in the computation of the low-weight perfect matching.
n	The number of vertices of the TSP complete graph.

Details

In the genetic algorithm, the crossover operation is a generalization of local search in which two tours are combined somehow to produce two tours, hopefully different from their parents and with better fitting function values. Crossover widens the search while trying to keep the good peculiarities of the parents. However, in practice crossover almost never lowers the fitting function when parents are near the optimum, but it helps to explore new routes. Therefore, it is always a good idea to complement crossover with some deterministic local search procedure which can find another local optima; crossover also helps in evading local minima.

In this routine, crossover is performed as follows. Firstly, the edges of the parents are combined in a single graph, and the repeated edges are eliminated. Then, the odd degree vertices of the resulting graph are matched looking for a low-weight perfect matching using a greedy algorithm. Adding the matching to the previous graph yields an Eulerian graph, as in Christofides algorithm, whose final step leads to the first offspring tour. The second tour is constructed by recording the second visit of each vertex by the Eulerian walk, and completing the resulting partial tour with the nearest neighbor heuristic.

Value

A two-row matrix containing the two offsprings as vertex numeric vectors.

Author(s)

Cesar Asensio

See Also

search_tour_genetic implements a version of the genetic algorithm for the TSP.

Examples

```
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
c1 <- sample(1:n)
c2 <- sample(1:n)
c12 <- crossover_tours(c1, c2, d, n)
compute_tour_distance(c1, d)        # 114.5848
compute_tour_distance(c2, d)        # 112.8995
compute_tour_distance(c12[1,], d)   # 116.3589
compute_tour_distance(c12[2,], d)   # 111.5184
```

dfs_tree *Depth-first search tree*

Description

Computation of the depth-first tree search in an undirected graph.

Usage

```
dfs_tree(g, r)
```

Arguments

g	Graph
r	Root: Starting vertex growing the tree.

Details

Starting from a root vertex, the tree is grown by adding neighbors of the last vertex added to the tree. In this way, the tree has many levels and few branches and leaves. When the tree cannot grow further, it backtracks to previously added vertices with neighbors outside the tree, adding them until none is left.

Value

A directed spanning subgraph of g containing the edges of the DFS tree.

Author(s)

Cesar Asensio

See Also

bfs_tree breadth-first search tree; bfs and dfs in the igraph package.

Examples

```
g <- make_graph("Frucht")
T <- dfs_tree(g, 5)  # Root at v = 5
z <- layout_with_gem(g)
plot(g, layout = z, main = "Depth-first search tree")
plot(T, layout = z, add = TRUE, edge.color = "cyan4", edge.width = 2)
plot(T, layout = layout_as_tree(T))
```

`dijk` *Dijkstra' algorithm for shortest paths*

Description

Dijkstra's algorithm finding the sorthest paths from a root vertex to the remaining vertices of a graph using a spanning tree

Usage

```
dijk(g, d, r = 1)
```

Arguments

`g`	An igraph Graph
`d`	Weights (lengths) of the edges of g
`r`	Starting vertex — root of the output tree

Details

An implementation of Dijkstra's algorithm.

Value

A list with components: $tree, which is a sequence of pairs of vertices parent-son; $distances, which is a $2 \times n$ matrix with distances from the root vertex to the remaining vertices, and $parents, which contains the parent of each vertex in the tree, except for the root which has no parent, so its entry is NA.

Author(s)

Cesar Asensio

See Also

shortest_paths in the igraph package.

Examples

```
library(igraph)
g <- make_graph("Frucht")
n <- gorder(g)
set.seed(1);
d <- matrix(round(runif(n^2, min = 1, max = 10)), nrow = n)  # Distances
d <- d + t(d); for (i in 1:n) { d[i,i] <- 0 }          # Distance matrix
Td <- dijk(g, d, r = 1)
Td$distances
Td$parents
```

```
gTd <- make_graph(Td$tree, n = gorder(g))   # igraph tree
Eg <- as_edgelist(g)
dl <- c()   # We convert the matrix in a list:
for (e in 1:nrow(Eg)) { dl <- c(dl, d[Eg[e,1], Eg[e,2]]) }
z <- layout_with_kk(g)
plot(g, layout = z, edge.label = dl)
plot(gTd, layout = z, edge.color = "red3", add = TRUE)
```

`find_cover_BB` *Branch-and-Bound algorithm for the Vertex-Cover problem*

Description

This routine performs a version of the Branch-and-Bound algorithm for the VCP. It is an exact algorithm with exponential worst-case running time; therefore, it can be run only with a small number of vertices.

Usage

```
find_cover_BB(
  g,
  verb = TRUE,
  save.best.result = FALSE,
  filename.best.result = "best_result_find_cover_BB.Rdata",
  nu = gorder(g),
  X = c(),
  Xmin = c(),
  marks = rep("F", gorder(g)),
  call = 0
)
```

Arguments

`g`	Graph.
`verb`	Boolean: Should echo each newly found cover to the console? Defaults to TRUE.
`save.best.result`	Boolean: Should the algorithm save the result of the algorithm in a file? It defaults to FALSE. When save.best.result = TRUE, a file is created with the variable "Xbest" being the best result achieved by the algorithm before its termination.
`filename.best.result`	Name of the file created when save.best.result = TRUE. It defaults to "best_result_find_cover_BB.Rdata".
`nu`	Size of the best cover currently found.
`X`	Partial cover.

`Xmin`	Best cover found so far.
`marks`	Mark sequence storing the current state of the algorithm, see details.
`call`	Number of recursive calls performed by the algorithm.

Details

The algorithm traverses a binary tree in which each bifurcation represents if a vertex is included in or excluded from a partial cover. The leaves of the tree represent vertex subsets; the algorithm checks if at some point the partial cover cannot become a full cover because of too many uncovered edges with too few remaining vertices to decide. In this way, the exponential complexity is somewhat reduced. Furthermore, the vertices are considered in decreasing degree order, as in the greedy algorithm, so that some cover is found in the early steps of the algorithm and thus a good upper bound on the solution can be used to exclude more subsets from being explored. The full algorithm has been extracted from the reference below.

In this routine, the binary tree search is implemented by recursive calls (that is, a dynamic programming algorithm). Although the worst case time complexity is exponential (recall that the Minimum Vertex Cover Problem is NP-hard), the approach is fairly efficient for a branch-and-bound technique.

The tree node in which the algorithm is when it is called (by the user or by itself) is encoded in a sequence of vertex marks. Marks come in three flavors: "C" is assigned to "Covered" vertices, that is, already included in the partial cover. "U" is asigned to "Uncovered" vertices, that is, those excluded from the partial cover. Mark "F" is assigned to "Free" vertices, those not considered yet by the algorithm; one of them is considered in the actual function call, and the subtree under this vertex is explored before returning. This mark sequence starts and ends with all vertices marked "F", and is used only by the algorithm, which modifies and passes it on to succesive calls to itself.

When the verb argument is TRUE, the routine echoes to the console the newly found cover only if it is better than the last. This report includes the size, actual cover and number call of the routine.

The routine can drop the best cover found so far in a file so that the user can stop the run afterwards; this technique might be useful when the full run takes too much time to complete.

Value

A list with five components: $set contains the subset of V(g) representing the cover and $size contains the number of vertices of the cover. Component $call is the number of calls the algorithm did on itself. The remaining components are used to transfer the state of the algorithm in the search three from one call to the next; they are $partial, the partially constructed cover, and $marks, a sequence encoding the tree node in which the algorithm is when this function is called, see details.

Author(s)

Cesar Asensio

See Also

is_cover checks if a vertex subset is a vertex cover, build_cover_greedy builds a cover using a greedy heuristic, build_cover_approx builds a cover using a 2-approximation algorithm, improve_cover_flip improves a cover using local search, search_cover_random looks for a random cover of fixed size,

search_cover_ants looks for a random cover using a version of the ant-colony optimization heuristic, plot_cover plots a cover.

Examples

```
set.seed(1)
g <- sample_gnp(25, p=0.25)        # Random graph
X7 <- find_cover_BB(g)
X7$size                            # Exact result: 16
X7$call                            # 108 recursive calls!
plot_cover(X7, g)

## Saving best result in a file (useful if the algorithm takes too
## long and should be interrupted by the user)
## It uses tempdir() to store the file
## The variable is called "Xbest"
find_cover_BB(g, save.best.result = TRUE, filename.best.result = "BestResult_BB.Rdata")
```

---

find_euler *Constructing an Eulerian Cycle*

Description

It finds an **Eulerian cycle** using a $O(q)$ algorithm where q is the number of edges of the graph.

Usage

```
find_euler(G, v)
```

Arguments

G	Eulerian and connected graph
v	Any vertex as starting point of the cycle

Details

Recursive algorithm for undirected graphs. The input graph should be Eulerian and connected.

Value

A two-element list: the $walk component is a $q \times 2$ matrix edgelist, and the $graph component is the input graph with no edges; it is used in intermediate steps, when the function calls itself.

Disclaimer

This function is part of the subject "Graphs and Network Optimization". It is designed for teaching purposes only, and not for production.

- It is introduced in the "Connectivity" section.
- It is used as a subroutine in the "TSP" section.

Author(s)

Cesar Asensio (2021)

References

Korte, Vygen: Combinatorial Optimization (Springer) sec 2.4.

See Also

build_tour_2tree double-tree algorithm.

Examples

```
library(igraph)
find_euler(make_full_graph(5), 1)$walk   # Walk of length 10
```

---

find_tour_BB *Branch-and-Bound algorithm for the TSP*

Description

This routine performs a version of the Branch-and-Bound algorithm for the Traveling Salesperson Problem (TSP). It is an exact algorithm with exponential worst-case running time; therefore, it can be run only with a very small number of cities.

Usage

```
find_tour_BB(
  d,
  n,
  verb = FALSE,
  plot = TRUE,
  z = NA,
  tour = rep(0, n),
  distance = 0,
  upper = Inf,
  col = c(1, rep(0, n - 1)),
  last = 1,
  partial = c(1, rep(NA, n - 1)),
  covered = 0,
  call = 0,
  save.best.result = FALSE,
  filename.best.result = "best_result_find_tour_BB.Rdata",
  order = NA
)
```

Arguments

d	Distance matrix of the TSP instance
n	Number of vertices of the TSP complete graph
verb	If detailed operation of the algorithm should be echoed to the console. It defaults to FALSE
plot	If tours found by the algorithm should be plotted using plot_tour. It defaults to TRUE
z	Points to plot the tours found by the algorithm. It defaults to NA; it should be set if plot is TRUE or else plot_tour will not plot the tours
tour	Best tour found by the algorithm. If the algorithm has ended its complete run, this is the optimum of the TSP instance. This variable is used to store the internal state of the algorithm and it should not be set by the user
distance	Distance covered by the best tour found. This variable is used to store the internal state of the algorithm and it should not be set by the user
upper	Upper bound on the distance covered by the optimum tour. It can be provided by the user or the routine will use the result found by the heuristic build_tour_nn_best
col	Vectors of "colors" of vertices. This variable is used to store the internal state of the algorithm and it should not be set by the user
last	Last vertex added to the tour being built by the algorithm. This variable is used to store the internal state of the algorithm and it should not be set by the user
partial	Partial tour built by the algorithm. This variable is used to store the internal state of the algorithm and it should not be set by the user
covered	Partial distance covered by the partial tour built by the algorithm. This variable is used to store the internal state of the algorithm and it should not be set by the user
call	Number of calls that the algorithm performs on itself. This variable is used to store the internal state of the algorithm and it should not be set by the user
save.best.result	The time needed for a complete run of this algorithm may be exponentially large. Since it only will return its results if it ends properly, we can save to a file the best result found by the routine at a given time when save.best.result = TRUE (default is FALSE). Then, the user will be allowed to stop the run of the algorithm without losing the (possibly suboptimal) result.
filename.best.result	The name of the file used to store the best result found so far when save.best.result = TRUE. It defaults to "best_result_find_tour_BB.Rdata". When loaded, this file will define the best tour in variable "Cbest".
order	Numeric vector giving the order in which vertices will be search by the algorithm. It defaults to NA, in which case the algorithm will take the order of the tour found by the heuristic build_tour_nn_best. If the user knows in advance some good tour and he/she wishes to use the order of its vertices, it should be taken into account that the third vertex used by the algorithm is the last vertex of the tour!

Details

The algorithm starts at city 1 (to avoid the cyclic permutation tour equivalence) and the "branch" phase consists on the decision of which city follows next. In order to avoid the equivalence between a tour and its reverse, it only considers those tours for which the second city has a smaller vertex id that the last. With n cities, the total number of tours explored in this way is $(n-1)!/2$, which clearly is infeasible unless n is small. Hence the "bound" phase estimates a lower bound on the distance covered by the tours which already are partially constructed. When this lower bound grows larger than an upper bound on the optimum supplied by the user or computed on the fly, the search stops in this branch and the algorithm proceeds to the next. This complexity reduction does not help in the worst case, though.

This routine represents the tree search by iterating over the sucessors of the present tree vertex and calling itself when descending one level. The leaves of the three are the actual tours, and the algorithm only reaches those tours whose cost is less than the upper bound provided. By default, the algorithm will plot the tour found if the coordinates of the cities are supplied in the "z" input argument.

When the routine takes too much time to complete, interrupting the run would result in losing the best tour found. To prevent this, the routine can store the best tour found so far so that the user can stop the run afterwards.

Value

A list with nine components: $tour contains a permutation of the 1:n sequence representing the best tour constructed by the algorithm, $distance contains the value of the distance covered by the tour, which if the algorithm has ended properly will be the optimum distance. Component $call is the number of calls the algorithm did on itself. The remaining components are used to transfer the state of the algorithm in the search three from one call to the next; they are $upper for the current upper bound on the distance covered by the optimum tour, $col for the "vertex colors" used to mark the vertices added to the partially constructed tour, which is stored in $partial. The distance covered by this partial tour is stored in $covered, the last vertex added to the partial tour is stored in $last, and the "save.best.result" and "filename.best.result" input arguments are stored in $save.best.result and $filename.best.result.

Author(s)

Cesar Asensio

Examples

```
## Random points
set.seed(1)
n <- 10
z <- cbind(runif(n, min=1, max=10), runif(n, min=1, max=10))
d <- compute_distance_matrix(z)
bb <- find_tour_BB(d, n)
bb$distance                         # Optimum 26.05881
plot_tour(z,bb)
## Saving tour to a file (useful when the run takes too long):
## Can be stopped after tour is found
## File is stored in tempdir(), variable is called "Cbest"
```

```
find_tour_BB(d, n, save.best.result = TRUE, z = z)
```

---

gauge_tour *Gauging a tour*

Description

Gauging a tour for easy comparison.

Usage

```
gauge_tour(To, n)
```

Arguments

To	Tour to be gauged, a vector containing a permutation of the 1:n sequence
n	Number of elements of the tour T

Details

A tour of n vertices is a permutation of the ordered sequence 1:n, and it is represented as a vector containing the integers from 1 to n in the permuted sequence. As a subgraph of the complete graph with n vertices, it is assumed that each vertex is adjacent with the anterior and posterior ones, with the first and last being also adjacent.

With respect to the TSP, a tour is invariant under cyclic permutation and inversion, so that there exists $(n-1)!/2$ different tours in a complete graph of n vertices. When searching for tours it is common to find the same tour under a different representation. Therefore, we need to establish wheter two tours are equivalent or not. To this end, we can "gauge" the tour by permuting cyclically its elements until the first vertex is at position 1, and fix the orientation so that the second vertex is less than the last. Two equivalent tours will have the same "gauged" representation.

This function is used in search_tour_genetic to discard repeated tours which can be found during the execution of the algorithm.

Value

The gauged tour.

Author(s)

Cesar Asensio

See Also

search_tour_genetic implements a version of the genetic algorithm for the TSP.

Examples

```
set.seed(2)
T0 <- sample(1:9,9)   #        T0 = 2 6 5 9 7 4 1 8 3
gauge_tour(T0, 9)     # gauged T0 = 1 4 7 9 5 6 2 3 8
```

`generate_fundamental_cycles` *Generate fundamental cycles in a connected graph*

Description

Generation of a system of fundamental cycles in a connected graph with respect of a given spanning tree.

Usage

```
generate_fundamental_cycles(eT, eG)
```

Arguments

`eT`	Spanning tree of the graph in edgelist representation, see as_edgelist.
`eG`	Graph in edgelist representation, see as_edgelist.

Details

The routine loops through the edges of the graph outside the spanning tree (there are |E| - |V| + 1 of them); in each step, it adds an edge to the tree, thus closing a cycle, which has some "hair" in it in the form of dangling vertices. Then all those dangling vertices are removed from the cycle (the "hair" is "shaven").

Value

A matrix with the fundamental cycles in its rows, in edge vector representation, that is, a binary vector with 1 if the edge belongs to the cycle and 0 otherwise. This interpretation of the edge vectors of each fundamental cycle refers to the edgelist of the graph given in eG.

Author(s)

Cesar Asensio

See Also

shave_cycle shaves hairy cycles, apply_incidence_map applies the incidence map of a graph to an edge vector.

Examples

```
g <- make_graph("Dodecahedron")
n <- gorder(g)
b <- bfs(g, 1, father = TRUE)                 # BFS tree
T <- make_graph(rbind(b$father[2:n], 2:n), n) # Tree as igraph graph
eT <- as_edgelist(T)
eG <- as_edgelist(g)
C <- generate_fundamental_cycles(eT, eG)      # Fundamental cycles
mu <- gsize(g) - gorder(g) + 1                # Cyclomatic number
z <- layout_with_gem(g)
for (i in 1:mu) {                             # Cycle drawing
    c1 <- make_graph(t(eG[which(C[i,] == 1),]) , dir = FALSE)
    plot(g, layout = z)
    plot(c1, layout = z, add = TRUE, edge.color = "cyan4",
         edge.lty = "dashed", edge.width = 3)
    title(paste0("Cycle ", i, " of ", mu))
    #Sys.sleep(1) # Adjust time to see the cycles
}
```

---

gor *Graphs and Network Optimization algorithms*

Description

This is a hardly complete collection of algorithms from the subject "Graphs and Network Optimization".

Details

Functions in this package have been written for teaching purposes. Therefore, no attempt at production versions of the algorithms has been made. They are neither complete nor completely correct, although a great effort has been invested in their construction and debugging. All of them pass a series of tests given in the examples sections of the corresponding help pages. Comments and suggestions are welcome, even if I cannot guarantee that they will be incorporated to the package.

This package makes extensive use of the igraph package functions, which should be loaded before using "gor".

Some functions in this package perform tasks which can be found in other well-tested packages such as igraph or TSP. As said before, these functions have been written with teaching in mind, and I do not claim that gor functions are better than any other in any way whatsoever.

Author(s)

Cesar Asensio (2021-2023)

`improve_cover_flip` *Improving a cover with local search*

Description

Local search to improve a cover by using "neighboring" vertex subsets differing in just one element from the initial subset.

Usage

```
improve_cover_flip(G, X)
```

Arguments

`G`	A graph
`X`	A cover list with components $set, $size as returned by routines build_cover_greedy or build_cover_approx. X represents the cover to be improved

Details

Given some cover specified by a vertex subset X in a graph, this routine scans the neighboring subsets obtained from X by removing a vertex from X looking for a smaller cover. If such a cover is found, it replaces the starting cover and the search starts again. This iterative procedure continues until no smaller cover can be found. Of course, the resulting cover is only a local minimum.

Value

A list with two components: $set contains the subset of V(g) representing the cover and $size contains the number of vertices of the cover.

Author(s)

Cesar Asensio

See Also

is_cover checks if a vertex subset is a vertex cover, build_cover_greedy builds a cover using a greedy heuristic, build_cover_approx builds a cover using a 2-approximation algorithm, search_cover_random looks for a random cover of fixed size, search_cover_ants looks for a random cover using a version of the ant-colony optimization heuristic, find_cover_BB finds covers using a branch-and-bound technique, plot_cover plots a cover.

Examples

```
set.seed(1)
n <- 25
g <- sample_gnp(n, p=0.25)  # Random graph

X1 <- build_cover_greedy(g)
X1$size    # 17
plot_cover(X1, g)

X2 <- build_cover_approx(g)
X2$size    # 20
plot_cover(X2, g)

X3 <- improve_cover_flip(g, X1)
X3$size    # 17 : Not improved
plot_cover(X3,g)

X4 <- improve_cover_flip(g, X2)
X4$size    # 19 : It is improved by a single vertex
plot_cover(X4,g)

# Vertex subsets or n-1 elements are always vertex covers:
for (i in 1:25) {
   X3 <- improve_cover_flip(g, list(set = setdiff(1:25,i), size = 24))
   print(X3$size)
} # 19 18 18 18 18 18 17 20 19 17 17 18 18 18 17 19 20 19 19 17 19 19 19 19 19
```

`improve_cut_flip` *Improving a cut with local search*

Description

Local search to improve a cut by using "neighboring" vertex subsets differing in just one element from the initial subset.

Usage

```
improve_cut_flip(G, K, w = NA, return.cut = TRUE)
```

Arguments

`G`	A graph
`K`	A cut list with components $set, $size, $weight and $cut as returned by routines build_cut_greedy, build_cut_random or compute_cut_weight. Only the $set and $weight components are used. K represents the cut to be improved
`w`	Weight matrix (defaults to NA). It should be zero for those edges not in G
`return.cut`	Boolean. Should the routine return the cut? It is passed on to compute_cut_weight on return. It defaults to TRUE

Details

Given some cut specified by a vertex subset S in a graph, this routine scans the neighboring subsets obtained from S by adding/removing a vertex from S looking for a larger cut. If such a cut is found, it replaces the starting cut and the search starts again. This iterative procedure continues until no larger cut can be found. Of course, the resulting cut is only a local maximum.

Value

A list with four components: $set contains the subset of V(g) representing the cut, $size contains the number of edges of the cut, $weight contains the weight of the cut (which coincides with $size if w is NA) and $cut contains the edges of the cut, joining vertices inside $set with vertices outside $set. When return.cut is FALSE, components $set and $cut are omitted.

Author(s)

Cesar Asensio

See Also

build_cut_random builds a random cut, build_cut_greedy builds a cut using a greedy algorithm, compute_cut_weight computes cut size, weight and edges, plot_cut plots a cut.

Examples

```
set.seed(1)
n <- 25
g <- sample_gnp(n, p=0.25)  # Random graph

c1 <- build_cut_random(g)
c1$size    # 44
plot_cut(c1, g)

c2 <- build_cut_greedy(g)
c2$size    # 59
plot_cut(c2, g)

c3 <- improve_cut_flip(g, c1)
c3$size    # 65
plot_cut(c3,g)

c4 <- improve_cut_flip(g, c2)
c4$size    # 60
plot_cut(c4,g)
```

`improve_tour_2opt` *Tour improving for a TSP using the 2-opt heuristic*

Description

2-opt heuristic tour-improving algorithm for the Traveling Salesperson Problem

Usage

```
improve_tour_2opt(d, n, C)
```

Arguments

`d`	Distance matrix of the TSP.
`n`	Number of vertices of the TSP complete graph.
`C`	Starting tour to be improved.

Details

It applies the 2-opt algorithm to a starting tour of a TSP instance until no further improvement can be found. The tour thus improved is a 2-opt local minimum.

The 2-opt algorithm consists of applying all possible 2-interchanges on the starting tour. Informally, a 2-interchange is the operation of cutting the tour in two pieces (by removing two nonincident edges) and gluing the pieces together to form a new tour by interchanging the endpoints.

Value

A list with two components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, $distance contains the value of the distance covered by the tour.

Author(s)

Cesar Asensio

See Also

improve_tour_3opt improves a tour using the 3-opt algorithm, build_tour_nn_best nearest neighbor heuristic, build_tour_2tree double-tree heuristic, compute_tour_distance computes tour distances, compute_distance_matrix computes a distance matrix, plot_tour plots a tour.

Examples

```
## Regular example with obvious solution (minimum distance 48)
m <- 10   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
b <- build_tour_2tree(d, n)
```

```
b$distance    # Distance 57.868
bi <- improve_tour_2opt(d, n, b$tour)
bi$distance   # Distance 48 (optimum)
plot_tour(z,b)
plot_tour(z,bi)

## Random points
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
b <- build_tour_2tree(d, n)
b$distance    # Distance 48.639
bi <- improve_tour_2opt(d, n, b$tour)
bi$distance   # Distance 37.351
plot_tour(z,b)
plot_tour(z,bi)
```

`improve_tour_3opt` *Tour improving for a TSP using the 3-opt heuristic*

Description

3-opt heuristic tour-improving algorithm for the Traveling Salesperson Problem

Usage

```
improve_tour_3opt(d, n, C)
```

Arguments

`d`	Distance matrix of the TSP.
`n`	Number of vertices of the TSP complete graph.
`C`	Starting tour to be improved.

Details

It applies the 3-opt algorithm to a starting tour of a TSP instance until no further improvement can be found. The tour thus improved is a 3-opt local minimum.

The 3-opt algorithm consists of applying all possible 3-interchanges on the starting tour. A 3-interchange removes three non-indicent edges from the tour, leaving three pieces, and combine them to form a new tour by interchanging the endpoints in all possible ways and gluing them together by adding the missing edges.

Value

A list with two components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, $distance contains the value of the distance covered by the tour.

Author(s)

Cesar Asensio

See Also

improve_tour_2opt improves a tour using the 2-opt algorithm, build_tour_nn_best nearest neighbor heuristic, build_tour_2tree double-tree heuristic, compute_tour_distance computes tour distances, compute_distance_matrix computes a distance matrix, plot_tour plots a tour.

Examples

```
## Regular example with obvious solution (minimum distance 32)
m <- 6   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
b <- build_tour_2tree(d, n)
b$distance    # Distance 38.43328
bi <- improve_tour_3opt(d, n, b$tour)
bi$distance   # Distance 32 (optimum)
plot_tour(z,b)
plot_tour(z,bi)

## Random points
set.seed(1)
n <- 15
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
b <- build_tour_2tree(d, n)
b$distance    # Distance 45.788
bi <- improve_tour_3opt(d, n, b$tour)
bi$distance   # Distance 32.48669
plot_tour(z,b)
plot_tour(z,bi)
```

---

improve_tour_LinKer *Tour improving for a TSP using a poor version of the Lin-Kernighan heuristic*

Description

Lin-Kernighan heuristic tour-improving algorithm for the Traveling Salesperson Problem using fixed 2-opt instead of variable k-opt exchanges.

Usage

```
improve_tour_LinKer(d, n, C, try = 5)
```

Arguments

d	Distance matrix of the TSP.
n	Number of vertices of the TSP complete graph.
C	Starting tour to be improved.
try	Number of tries before quitting.

Details

It applies a version of the core Lin-Kernighan algorithm to a starting tour of a TSP instance until no further improvement can be found. The tour thus improved is a local minimum.

The Lin-Kernighan algorithm implemented here is based on the core routine described in the reference below. It is provided here as an example of a local search routine which can be embedded in larger search strategies. However, instead of using variable k-opt moves to improve the tour, it uses 2-exhanges only, which is far easier to program. Tours improved with this technique are of course 2-opt.

The TSP library provides an interface to the Lin-Kernighan algorithm with all its available improvements in the external program Concorde, which should be installed separately.

Value

A list with two components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, $distance contains the value of the distance covered by the tour.

Author(s)

Cesar Asensio

References

Hromkovic *Algorithmics for Hard Problems* (2004)

See Also

improve_tour_2opt improves a tour using the 2-opt algorithm, improve_tour_3opt improves a tour using the 3-opt algorithm, build_tour_nn_best nearest neighbor heuristic, build_tour_2tree double-tree heuristic, compute_tour_distance computes tour distances, compute_distance_matrix computes a distance matrix, plot_tour plots a tour.

Examples

```
## Regular example with obvious solution (minimum distance 48)
m <- 10   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
b <- build_tour_2tree(d, n)
b$distance    # Distance 57.868
bi <- improve_tour_LinKer(d, n, b$tour)
```

```
bi$distance   # Distance 48 (optimum)
plot_tour(z,b)
plot_tour(z,bi)

## Random points
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
b <- build_tour_2tree(d, n)
b$distance    # Distance 48.639
bi <- improve_tour_LinKer(d, n, b$tour)
bi$distance   # Distance 37.351 (2-opt)
plot_tour(z,b)
plot_tour(z,bi)
```

`is_cover` *Check vertex cover*

Description

Check if some vertex subset of a graph covers all its edges.

Usage

```
is_cover(X, eG)
```

Arguments

`X`	Vertex subset to check.
`eG`	Edgelist of the graph as returned by as_edgelist

Details

The routine simply goes through the edge list of the graph to see if both ends of each edge are inside the vertex subset to be checked. When an edge with both ends ouside X is encountered, the routine returns FALSE; otherwise, it returns TRUE.

Value

Boolean: TRUE if X is a vertex cover of the graph represented by eG, FALSE otherwise.

Author(s)

Cesar Asensio

See Also

build_cover_greedy builds a cover using a greedy heuristic, build_cover_approx builds a cover using a 2-approximation algorithm, improve_cover_flip improves a cover using local search, search_cover_random looks for a random cover of fixed size, search_cover_ants looks for a random cover using a version of the ant-colony optimization heuristic, find_cover_BB finds covers using a branch-and-bound technique, plot_cover plots a cover.

Examples

```
set.seed(1)
n <- 25
g <- sample_gnp(n, p=0.25)  # Random graph
eg <- as_edgelist(g)

X1 <- build_cover_greedy(g)
is_cover(X1$set, eg)   # TRUE
is_cover(c(1:10),eg)   # FALSE
plot_cover(list(set = 1:10, size = 10), g)  # See uncovered edges
```

`mutate_binary_sequence` *Binary sequence mutation*

Description

Mutation of binary sequences for use in the genetic algorithm

Usage

```
mutate_binary_sequence(s, p = 0.1)
```

Arguments

s	Sequence consisting of 0 and 1
p	Mutation probability. Defaults to 0.1

Details

This routine takes a binary sequence and it flips ("mutates") each bit with a fixed probability. In the genetic algorithm context, this operation randomly explores regions of configuration space which are far away from the starting point, thus trying to avoid local optima. The fitting function values of mutated individuals are generically very poor, and this behavior is to be expected. Thus, mutation is not an optimization procedure per se.

Value

A mutated binary sequence

Author(s)

Cesar Asensio

See Also

search_cut_genetic genetic cut-searching algorithm, crossover_sequences crossover operation

Examples

```
set.seed(1)
s <- sample(0:1, 10, replace = TRUE)  # 0 0 1 1 0 1 1 1 1 0
mutate_binary_sequence(s, p = 0.5)    # 1 1 1 0 0 0 1 1 0 0
mutate_binary_sequence(s, p = 1)      # 1 1 0 0 1 0 0 0 0 1
```

---

neigh_index *Previous, current, and next positions of a given index in a cycle.*

Description

Previous, current, and next positions of a given index in a cycle.

Usage

```
neigh_index(i, n)
```

Arguments

i	Position in a cycle
n	Lenght of the cycle

Details

Given some position i in a n-lenght cycle, this function returns the triple c(i-1,i,i+1) taking into account that the next position of i=n is 1 and the previous position of i=1 is n. It is used to perform a 4-exchange in a cycle.

Value

A three component vector c(previous, current, next)

Author(s)

Cesar Asensio

Examples

```
neigh_index(6, 9)  # 5 6 7
neigh_index(9, 9)  # 8 9 1
neigh_index(1, 9)  # 9 1 2
```

`next_index` *Next position to i in a cycle*

Description

Next position to i in a cycle.

Usage

```
next_index(i, n)
```

Arguments

`i`	Position in cycle
`n`	Lenght of cycle

Details

In a cycle, the next slot to the i-th position is i+1 unless i=n. In this case, the next is 1.

Value

The next position in cycle

Author(s)

Cesar Asensio

Examples

```
next_index(5, 7)   # 6
next_index(7, 7)   # 1
```

perturb_tour_4exc *Random 4-exchange transformation*

Description

It performs a random 4-exchange transformation to a cycle.

Usage

```
perturb_tour_4exc(C, V, n)
```

Arguments

C	Cycle to be 4-exchanged
V	1:n list, positions to draw from
n	Number of vertices of the cycle

Details

The transformation is carried out by randomly selecting four non-mutually incident edges from the cycle. Upon eliminating these four edges, we obtain four pieces ci of the original cycle. The 4-exchanged cycle is c1, c4, c3, c2. This is a typical 4-exchange which cannot be constructed using 2-exchanges and therefore it is used by local search routines as an escape from 2-opt local minima.

Value

The 4-exchanged cycle.

Author(s)

Cesar Asensio

Examples

```
set.seed(1)
perturb_tour_4exc(1:9, 1:9, 9)   # 2 3 9 1 7 8 4 5 6
```

`plot_cover` *Vertex cover plotting*

Description

Plot of a vertex cover in a graph.

Usage

```
plot_cover(X, G)
```

Arguments

`X`	Cover to be plotted; an output list returned by some cover-building function, see below.
`G`	Graph on which to superimpose the cover.

Details

It plots a graph, then superimposes a vertex cover in a different color. It also draws the covered edges, to help in detecting non-covers by inspection.

Value

This function is called for its side effect of plotting.

Author(s)

Cesar Asensio

See Also

is_cover checks if a vertex subset is a vertex cover, build_cover_greedy builds a cover using a greedy heuristic, build_cover_approx builds a cover using a 2-approximation algorithm, improve_cover_flip improves a cover using local search, search_cover_random looks for a random cover of fixed size, search_cover_ants looks for a random cover using a version of the ant-colony optimization heuristic, find_cover_BB finds covers using a branch-and-bound technique.

Examples

```
set.seed(1)
g <- sample_gnp(25, p=0.25)   # Random graph
X1 <- build_cover_greedy(g)
plot_cover(X1, g)

st <- 1:5   # Not a vertex cover
plot_cover(list(set = st, size = length(st)), g)  # See covered edges
```

plot_cut *Cut plotting*

Description

Plot of a cut in a graph.

Usage

```
plot_cut(K, G)
```

Arguments

K	Cut to be plotted; an output list returned by some cut-building function, see below.
G	Graph on which to superimpose the cut.

Details

It plots a graph, then superimposes a cut, drawing the associated vertex set in a different color.

Value

This function is called for its side effect of plotting.

Author(s)

Cesar Asensio

See Also

build_cut_random builds a random cut, build_cut_greedy builds a cut using a greedy algorithm, improve_cut_flip uses local search to improve a cut obtained by other methods, compute_cut_weight computes cut size, weight and edges.

Examples

```
K10 <- make_full_graph(10)   # Max cut of size 25
c0 <- build_cut_random(K10)
plot_cut(c0, K10)
```

plot_tour *TSP tour simple plotting*

Description

Plotting tours constructed by tour-building routines for TSP

Usage

```
plot_tour(z, h, ...)
```

Arguments

z	Set of points of a TSP
h	List with $tour and $distance components returned from a TSP tour building algorithm
...	Parameters to be passed to plot

Details

It plots the two-dimensional cities of a TSP and a tour among them for visualization purposes. No aesthetically appealing effort has been invested in this function.

Value

This function is called by its side effect.

Author(s)

Cesar Asensio

See Also

build_tour_nn nearest neighbor heuristic with a single starting point, build_tour_nn_best repeats the previous algorithm with all possible starting points, compute_distance_matrix computes the distance matrix of a set of two-dimensional points.

Examples

```
set.seed(1)
n <- 25
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
b <- build_tour_nn_best(d, n)
plot_tour(z,b)
```

search_cover_ants *Ant colony optimization algorithm for Vertex-Cover*

Description

Ant colony optimization (ACO) heuristic algorithm to search for a vertex cover of small size in a graph. ACO is a random algorithm; as such, it yields better results when longer searches are run. To guess the adequate parameter values resulting in better performance in particular instances requires some experimentation, since no universal values of the parameters seem to be appropriate to all examples.

Usage

```
search_cover_ants(g, K, N, alpha = 2, beta = 2, dt = 1, rho = 0.1, verb = TRUE)
```

Arguments

g	Graph.
K	Number of ants per iteration.
N	Number of iterations.
alpha	Exponent of the pheronome index, see details.
beta	Exponent of the vertex degree, see details.
dt	Pheromone increment.
rho	Pheromone evaporation rate.
verb	Boolean; if TRUE (default) it echoes to the console the routine progress .

Details

ACO is an optimization paradigm that tries to replicate the behavior of a colony of ants when looking for food. Ants leave after them a soft pheromone trail to help others follow the path just in case some food has been found. Pheromones evaporate, but following again the trail reinforces it, making it easier to find and follow. Thus, a team of ants search a vertex cover in a graph, leaving a pheromone trail on the chosen vertices. At each step, each ant decides the next vertex to add based on the pheromone level and on the degree of the remaining vertices, according to the formula $P(v)\ phi(v)^alpha * exp(beta * d(v))$, where $phi(v)$ is the pheromone level, $d(v)$ is the degree of the vertex v, and $alpha$, $beta$ are two exponents to broad or sharpen the probability distribution. After each vertex has been added to the subset, its incident adges are removed, following a randomized version of the greedy heuristic. In a single iteration, each ant builds a vertex cover, and the best of them is recorded. Then the pheromone level of the vertices of the best cover are enhanced, and the remaining pheromones begin to evaporate.

Default parameter values have been chosen in order to find the optimum in the examples considered below. However, it cannot be guarateed that this is the best choice for all cases. Keep in mind that no polynomial time exact algorithm can exist for the VCP, and thus harder instances will require to fine-tune the parameters. In any case, no guarantee of optimality of covers found by this method can be given, so they might be improved further by other methods.

Value

A list with three components: $set contains the subset of V(g) representing the cover and $size contains the number of vertices of the cover; $found is the number of vertex covers found in subsequent iterations (often they are repeated, that is, different explorations may find the same vertex cover).

Author(s)

Cesar Asensio

See Also

is_cover checks if a vertex subset is a vertex cover, build_cover_greedy builds a cover using a greedy heuristic, build_cover_approx builds a cover using a 2-approximation algorithm, improve_cover_flip improves a cover using local search, search_cover_random looks for a random cover of fixed size, find_cover_BB finds covers using a branch-and-bound technique, plot_cover plots covers.

Examples

```
set.seed(1)
g <- sample_gnp(25, p=0.25)                  # Random graph

X6 <- search_cover_ants(g, K = 20, N = 10)
plot_cover(X6, g)
X6$found
```

`search_cover_random` *Random vertex covers*

Description

Random algorithm for vertex-cover.

Usage

```
search_cover_random(G, N, k, alpha = 1)
```

Arguments

G	Graph.
N	Number of random vertex set to try.
k	Cardinality of the random vertex sets generated by the algorithm.
alpha	Exponent of the probability distribution from which vertices are drawn: P(v) ~ d(v)^alpha.

Details

This routine performs N iterations of the simple procedure of selecting a random sample of size k on the vertex set of a graph and check if it is a vertex cover, counting successes and failures. The last cover found is returned, or an empty set if none is found.

Value

A list with four components: $set contains the subset of V(g) representing the cover and $size contains the number of vertices of the cover (it coincides with k). $found is the number of vertex covers found and $failed is the number of generated subset that were not vertex covers.

Author(s)

Cesar Asensio

See Also

is_cover checks if a vertex subset is a vertex cover, build_cover_greedy builds a cover using a greedy heuristic, build_cover_approx builds a cover using a 2-approximation algorithm, improve_cover_flip improves a cover using local search, search_cover_ants looks for a random cover using a version of the ant-colony optimization heuristic, find_cover_BB finds covers using a branch-and-bound technique, plot_cover plots a cover.

Examples

```
set.seed(1)
n <- 25
g <- sample_gnp(n, p=0.25)  # Random graph
X7 <- search_cover_random(g, 10000, 17, alpha = 3)
plot_cover(X7, g)
X7$found    # 21 (of 10000) covers of size 17

## Looking for a cover of size 16...
X8 <- search_cover_random(g, 10000, 16, alpha = 3)  # ...we don't find any!
plot_cover(X8, g)   # All edges uncovered
X8$found    # 0
```

---

search_cut_genetic *Genetic Algorithm for Max-Cut*

Description

Genetic algorithm for Max-Cut. In addition to crossover and mutation, which are described below, the algorithm performs also local search on offsprings and mutants.

Usage

```
search_cut_genetic(
  G,
  w = NA,
  Npop = 5,
  Ngen = 20,
  pmut = 0.1,
  beta = 1,
  elite = 2,
  Pini = NA,
  verb = TRUE,
  log = FALSE
)
```

Arguments

G	Graph.
w	Weight matrix.
Npop	Population size.
Ngen	Number of generations (iterations of the algorithm).
pmut	Mutation probability. It defaults to 0.1.
beta	Control parameter of the crossing and selection probabilities. It defaults to 1.
elite	Number of better fitted individuals to pass on to the next generation. It defaults to 2.
Pini	Initial population. If it is NA, a random initial population of Npop individuals is generated. Otherwise, it should be a matrix; each row should be an individual (a permutation of the 1:n sequence) and then Npop is set to the number of rows of Pini. This option allows to chain several runs of the genetic algorithms, which could be needed in the hardest cases.
verb	Boolean to activate console echo. It defaults to TRUE.
log	Boolean to activate the recording of the weights of all cuts found by the algorithm. It defaults to FALSE.

Details

This algorithm manipulates cuts by means of its associated binary sequences defined as follows. Each cut K is defined by its associated vertex subset S of V(G): K contains all edges joining vertices inside S with vertices outside S. If |V(G)|=n, we can construct a n-bit binary sequence b = (b1,b2,...,bn) with bi = 1 if vertex vi belongs to S, and 0 otherwise.

The genetic algorithm consists of starting with a cut population, where each cut is represented by its corresponding binary sequence defined above, and thus the population is simply a binary matrix. This initial cut population can be provided by the user or can be random. The initial population can be the output of a previous run of the genetic algorithm, thus allowing a chained execution. Then the routine sequentially perform over the cuts of the population the **crossover**, **mutation**, **local search** and **selection** operations.

The **crossover** operation takes two cuts as "parents" and forms two "offsprings" by cutting and interchanging the binary sequences of the parents; see crossover_sequences for more information.

The **mutation** operation performs a "small" perturbation of each cut trying to escape from local optima. It uses a random flip on each bit of the binary sequence associated with the cut, see mutate_binary_sequence for more information.

The **local search** operation takes the cuts found by the crossover and mutation operations and improves them using some local search heuristic, in this case improve_cut_flip. This allows this algorithm to approach local maxima faster.

The **selection** operation is used when selecting pairs of parents for crossover and when selecting individuals to form the population for the next generation. In both cases, it uses a probability exponential in the weight with rate parameter "beta", favouring the better fitted to be selected. Lower values of beta favours the inclusion of cuts with worse fitting function values. When selecting the next population, the selection uses *elitism*, which is to save the best fitted individuals to the next generation; this is controlled with parameter "elite".

The usefulness of the crossover and mutation operations stems from its abitily to escape from the local maxima. Of course, more iterations (Ngen) and larger populations (Npop) might improve the result, but recall that no random algorithm can guarantee to find the optimum of a given Max-Cut instance.

This algorithm calls many times the routines compute_cut_weight, crossover_sequences, mutate_binary_sequence and improve_cut_flip; therefore, it is not especially efficient when called on large problems or with high populations or many generations. Please consider chaining the algorithm: perform short runs, using the output of a run as the input of the next.

Value

A list with several components: $set contains the subset of V(g) representing the cut, $size contains the number of edges of the cut, $weight contains the weight of the cut (which coincides with $size if w is NA) and $cut contains the edges of the cut, joining vertices inside $set with vertices outside $set; $generation contains the generation when the maximum was found and $population contains the final cut population. When log=TRUE, the output includes several lists of weights of cuts found by the algorithm, separated by initial cuts, offsprings, mutants, local maxima and selected cuts.

Author(s)

Cesar Asensio

References

Hromkovic *Algorithms for hard problems* (2004), Hartmann, Weigt, *Phase transitions in combinatorial optimization problems* (2005).

See Also

crossover_sequences performs crossover operation, mutate_binary_sequence performs mutation operation, build_cut_random builds a random cut, build_cut_greedy builds a cut using a greedy algorithm, improve_cut_flip improves a cut by local search, compute_cut_weight computes cut size, weight and edges, plot_cut plots a cut.

Examples

```
set.seed(1)
n <- 10
g <- sample_gnp(n, p=0.5)  # Random graph
c5 <- search_cut_genetic(g)
plot_cut(c5, g)
improve_cut_flip(g, c5)     # It does not improve
for (i in 1:5) {            # Weights of final population
   s5 <- which(c5$population[i,] == 1)
   cs5 <- compute_cut_weight(s5, gorder(g), as_edgelist(g))
   print(cs5$weight)
}


## Longer examples
c5 <- search_cut_genetic(g, Npop=10, Ngen=50, log = TRUE)
boxplot(c5$Wini, c5$Woff, c5$Wmut, c5$Wvec, c5$Wsel,
        names=c("Ini", "Off", "Mut", "Neigh", "Sel"))

set.seed(1)
n <- 20
g <- sample_gnp(n, p=0.25)
Wg <- matrix(sample(1:3, n^2, replace=TRUE), nrow=n)
Wg <- Wg + t(Wg)
A <- as_adjacency_matrix(g)
Wg <- Wg * A
c6 <- search_cut_genetic(g, Wg, Ngen = 9)   # Size 38, weigth 147
plot_cut(c6, g)
```

`search_tour_ants` *Ant colony optimization algorithm for the TSP*

Description

Ant colony optimization (ACO) heuristic algorithm to search for a low-distance tour of a TSP instance. ACO is a random algorithm; as such, it yields better results when longer searches are run. To guess the adequate parameter values resulting in better performance in particular instances requires some experimentation, since no universal values of the parameters seem to be appropriate to all examples.

Usage

```
search_tour_ants(
  d,
  n,
  K = 200,
  N = 50,
```

```
  beta = 3,
  alpha = 5,
  dt = 1,
  rho = 0.05,
  log = FALSE
)
```

Arguments

d	Distance matrix of the TSP instance.
n	Number of vertices of the complete TSP graph. It can be less than the number of rows of the distance matrix d.
K	Number of tour-searching ants. Defaults to 200.
N	Number of iterations. Defaults to 50.
beta	Inverse temperature which determines the thermal probability in selecting the next vertex in tour. High beta (low temperature) rewards lower distances (and thus it gets stuck sooner in local minima), while low beta (high temperature) rewards longer tours, thus escaping from local minima. Defaults to 3.
alpha	Exponent enhancing the pheromone trail. High alpha means a clearer trail, low alpha means more options. It defaults to 5.
dt	Basic pheromone enhancement at each iteration. It defaults to 1.
rho	Parameter in the (0,1) interval controlling pheromone evaporation rate. Pheromones of the chosen tour increase in dt*rho, while excluded pheromones diminish in 1-rho. A rho value near 1 means select just one tour, while lower values of rho spread the probability and more tours can be explored. It defaults to 0.05.
log	Boolean. When TRUE, it also outputs two vectos recording the performance of the algorithm. It defaults to FALSE.

Details

ACO is an optimization paradigm that tries to replicate the behavior of a colony of ants when looking for food. Ants leave after them a soft pheromone trail to help others follow the path just in case some food has been found. Pheromones evaporate, but following again the trail reinforces it, making it easier to find and follow. Thus, a team of ants search a tour in a TSP instance, leaving a pheromone trail on the edges of the tour. At each step, each ant decides the next step based on the pheromone level and on the distance of each neighboring edge. In a single iteration, each ant completes a tour, and the best tour is recorded. Then the pheromone level of the edges of the best tour are enhanced, and the remaining pheromones evaporate.

Default parameter values have been chosen in order to find the optimum in the examples considered below. However, it cannot be guarateed that this is the best choice for all cases. Keep in mind that no polynomial time exact algorithm can exist for the TSP, and thus harder instances will require to fine-tune the parameters. In any case, no guarantee of optimality of tours found by this method can be given, so they might be improved further by other methods.

Value

A list with two components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, $distance contains the value of the distance covered by the tour. When log=TRUE, the output list contains also the component $Lant, best tour distance found in the current iteration, and component $Lopt, best tour distance found before and including the current iteration.

Author(s)

Cesar Asensio

See Also

compute_distance_matrix computes matrix distances using 2d points, improve_tour_2opt improves a tour using 2-exchanges, plot_tour draws a tour

Examples

```
## Regular example with obvious solution (minimum distance 32)
m <- 6   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
set.seed(2)
b <- search_tour_ants(d, n, K = 70, N = 20)
b$distance    # Distance 32 (optimum)
plot_tour(z,b)

## Random points
set.seed(1)
n <- 15
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
b <- search_tour_ants(d, n, K = 50, N = 20)
b$distance    # Distance 32.48669
plot_tour(z,b)
```

---

search_tour_chain2opt *Chained 2-opt search with multiple, random starting tours*

Description

Random heuristic algorithm for TSP which performs chained 2-opt local search with multiple, random starting tours.

Usage

```
search_tour_chain2opt(d, n, Nit, Nper, log = FALSE)
```

Arguments

d	Distance matrix of the TSP instance.
n	Number of vertices of the TSP complete graph.
Nit	Number of iterations of the algorithm, see details.
Nper	Number of chained perturbations of 2-opt minima, see details.
log	Boolean: Whether the algorithm should record the distances of the tours it finds during execution. It defaults to FALSE.

Details

Chained local search consists of starting with a random tour, improving it using 2-opt, and then perturb it using a random 4-exchange. The result is 2-optimized again, and then 4-exchanged... This sequence of chained 2-optimizations/perturbations is repeated Nper times for each random starting tour. The entire process is repeated Nit times, drawing a fresh random tour each iteration.

The purpose of supplement the deterministic 2-opt algorithm with random additions (random starting point and random 4-exchange) is escaping from the 2-opt local minima. Of course, more iterations and more perturbations might lower the result, but recall that no random algorithm can guarantee to find the optimum in a reasonable amount of time.

This technique is most often applied in conjunction with the Lin-Kernighan local search heuristic.

It should be warned that this algorithm calls Nper*Nit times the routine improve_tour_2opt, and thus it is not especially efficient.

Value

A list with two components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, $distance contains the value of the distance covered by the tour.

Author(s)

Cesar Asensio

References

Cook et al. *Combinatorial Optimization* (1998)

See Also

perturb_tour_4exc transforms a tour using a random 4-exchange, improve_tour_2opt improves a tour using the 2-opt algorithm, improve_tour_3opt improves a tour using the 3-opt algorithm, build_tour_nn_best nearest neighbor heuristic, build_tour_2tree double-tree heuristic, compute_tour_distance computes tour distances, compute_distance_matrix computes a distance matrix, plot_tour plots a tour.

Examples

```
## Regular example with obvious solution (minimum distance 32)
m <- 6   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
bc <- search_tour_chain2opt(d, n, 5, 3)
bc     # Distance 48
plot_tour(z,bc)

## Random points
set.seed(1)
n <- 15
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
bc <- search_tour_chain2opt(d, n, 5, 3)
bc     # Distance 32.48669
plot_tour(z,bc)
```

---

search_tour_genetic *Genetic Algorithm for the TSP*

Description

Genetic algorithm for TSP. In addition to crossover and mutation, which are described below, the algorithm performs also 2-opt local search on offsprings and mutants. In this way, this algorithm is at least as good as chained 2-opt search.

Usage

```
search_tour_genetic(
  d,
  n,
  Npop = 20,
  Ngen = 50,
  beta = 1,
  elite = 2,
  Pini = NA,
  local = 1,
  verb = TRUE,
  log = FALSE
)
```

Arguments

d	Distance matrix of the TSP instance.
n	Number of vertices of the TSP complete graph.

`Npop`	Population size.
`Ngen`	Number of generations (iterations of the algorithm).
`beta`	Control parameter of the crossing and selection probabilities. It defaults to 1.
`elite`	Number of better fitted individuals to pass on to the next generation. It defaults to 2.
`Pini`	Initial population. If it is NA, a random initial population of Npop individuals is generated. Otherwise, it should be a matrix; each row should be an individual (a permutation of the 1:n sequence) and then Npop is set to the number of rows of Pini. This option allows to chain several runs of the genetic algorithm, which could be needed in the hardest cases.
`local`	Average fraction of parents + offsprings + mutants that will be taken as starting tours by the local search algorithm improve_tour_2opt. It should be a number between 0 and 1. It defauls to 1.
`verb`	Boolean to activate console echo. It defaults to TRUE.
`log`	Boolean to activate the recording of the distances of all tours found by the algorithm. It defaults to FALSE.

Details

The genetic algorithm consists of starting with a tour population, which can be provided by the user or can be random. The initial population can be the output of a previous run of the genetic algorithm, thus allowing a chained execution. Then the routine sequentially perform over the tours of the population the **crossover**, **mutation**, **local search** and **selection** operations.

The **crossover** operation takes two tours and forms two offsprings trying to exploit the good structure of the parents; see crossover_tours for more information.

The **mutation** operation performs a "small" perturbation of each tour trying to escape from local optima. It uses a random 4-exchange, see perturb_tour_4exc and search_tour_chain2opt for more information.

The **local search** operation takes the tours found by the crossover and mutation operations and improves them using the 2-opt local search heuristic, see improve_tour_2opt. This makes this algorithm at least as good as chained local search, see search_tour_chain2opt.

The **selection** operation is used when selecting pairs of parents for crossover and when selecting individuals to form the population for the next generation. In both cases, it uses a probability exponential in the distance with rate parameter "beta", favouring the better fitted to be selected. Lower values of beta favours the inclusion of tours with worse fitting function values. When selecting the next population, the selection uses *elitism*, which is to save the best fitted individuals to the next generation; this is controlled with parameter "elite".

The usefulness of the crossover and mutation operations stems from its abitily to escape from the 2-opt local minima in a way akin to the perturbation used in chained local search search_tour_chain2opt. Of course, more iterations (Ngen) and larger populations (Npop) might lower the result, but recall that no random algorithm can guarantee to find the optimum of a given TSP instance.

This algorithm calls many times the routines crossover_tours, improve_tour_2opt and perturb_tour_4exc; therefore, it is not especially efficient when called on large problems or with high populations of many generations. Input parameter "local" can be used to randomly select which tours will start local search, thus diminishing the run time of the algorithm. Please consider chaining the algorithm: perform short runs, using the output of a run as the input of the next.

Value

A list with four components: $tour contains a permutation of the 1:n sequence representing the tour constructed by the algorithm, $distance contains the value of the distance covered by the tour, $generation contains the generation is which the minimum was found and $population contains the final tour population. When log=TRUE, the output includes several lists of distances of tours found by the algorithm, separated by initial tours, offsprings, mutants, local minima and selected tours.

Author(s)

Cesar Asensio

References

Hromkovic *Algorithms for hard problems* (2004), Hartmann, Weigt, *Phase transitions in combinatorial optimization problems* (2005).

See Also

crossover_tours performs the crosover of two tours, gauge_tour transforms a tour into a canonical sequence for comparison, search_tour_chain2opt performs a chained 2-opt search, perturb_tour_4exc transforms a tour using a random 4-exchange, improve_tour_2opt improves a tour using the 2-opt algorithm, improve_tour_3opt improves a tour using the 3-opt algorithm, build_tour_nn_best nearest neighbor heuristic, build_tour_2tree double-tree heuristic, compute_tour_distance computes tour distances, compute_distance_matrix computes a distance matrix, plot_tour plots a tour.

Examples

```
## Regular example with obvious solution (minimum distance 32)
m <- 6   # Generate some points in the plane
z <- cbind(c(rep(0,m), rep(2,m), rep(5,m), rep(7,m)), rep(seq(0,m-1),4))
n <- nrow(z)
d <- compute_distance_matrix(z)
bc <- search_tour_genetic(d, n, Npop = 5, Ngen = 3, local = 0.2)
bc     # Distance 32
plot_tour(z,bc)

## Random points
set.seed(1)
n <- 15
z <- cbind(runif(n,min=1,max=10),runif(n,min=1,max=10))
d <- compute_distance_matrix(z)
bg <- search_tour_genetic(d, n, 5, 3, local = 0.25)
bg     # Distance 32.48669
plot_tour(z,bg)
```

shave_cycle *Shaving a hairy cycle*

Description

Removing dangling vertices of a cycle obtained by adding a single edge to a spanning tree.

Usage

```
shave_cycle(v, eG)
```

Arguments

v	Edge vector of the hairy cycle
eG	Graph given as edgelist, see as_edgelist

Details

When generating a fundamental cycle in a graph, addition of a single edge to a spanning tree gives a "hairy" cycle, that is, a single cycle with some dangling branches of the tree. This routine removes iteratively all leaves from this "hairy" tree until only a 2-regular, connected cycle remains, which is a fundamental cycle of the graph with respect the given spanning tree.

Value

Edge vector of the shaven cycle, to be interpreted with respect to the edgelist eG.

Author(s)

Cesar Asensio

See Also

generate_fundamental_cycles generates the edge vectors of a system of fundamental cycles of a graph, apply_incidence_map applies the incidence map of a graph to an edge vector.

Examples

```
## It is used as a subroutine in [generate_fundamental_cycles].
```

sum_g *Sum of the higher terms of a list*

Description

Sum of the higher terms of a list

Usage

```
sum_g(L, m)
```

Arguments

L	A numeric sequence
m	A scalar

Details

It sorts the list L in decreasing order and returns the sum of the first m components of the ordered list.

Value

The sum of the m higher terms of the list L

Author(s)

Cesar Asensio

Examples

```
sum_g(1:10, 3)  # 8 + 9 + 10 = 27
```

Index