Package: mwcsr (via r-universe)

September 10, 2024

Title Solvers for Maximum Weight Connected Subgraph Problem and Its Variants

Version 0.1.9

Description Algorithms for solving various Maximum Weight Connected Subgraph Problems, including variants with budget constraints, cardinality constraints, weighted edges and signals. The package represents an R interface to high-efficient solvers based on relax-and-cut approach (Álvarez-Miranda E., Sinnl M. (2017) <doi:10.1016/j.cor.2017.05.015>) mixed-integer programming (Loboda A., Artyomov M., and Sergushichev A. (2016) <doi:10.1007/978-3-319-43681-4_17>) and simulated annealing.

Depends R (>= 3.5)

Imports methods, igraph, Rcpp

Suggests knitr, rmarkdown, mathjaxr, testthat, BioNet, roxygen2, DLBCL

SystemRequirements Java (>=8)

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.2.3

Roxygen list(markdown = TRUE)

VignetteBuilder knitr

URL `https://github.com/ctlab/mwcsr`

BugReports `https://github.com/ctlab/mwcsr/issues`

LinkingTo Rcpp

Repository https://ctlab.r-universe.dev

RemoteUrl https://github.com/ctlab/mwcsr

RemoteRef HEAD

RemoteSha 6d1a851f7580c5e58c9a1a404e12e06b201e5608

Contents

`annealing_solver` *Construct an annealing solver*

Description

Simulated annealing is a heuristic method of solving optimization problems. Typically, it allows to find some good solution in a short time. This implementation doesn't compute any upper bound on solution, so there is no guarantee of optimality of solution provided.

Usage

```
annealing_solver(
  schedule = c("fast", "boltzmann"),
  initial_temperature = 1,
  final_temperature = 1e-06,
  verbose = FALSE
)
```

Arguments

`schedule` boltzmann annealing or fast annealing

`initial_temperature`
: initial value for the temperature

`final_temperature`
: final value for the temperature

`verbose` whether be verbose or not

Details

Algorithm maintains connected subgraph staring with empty subgraph. Each iteration one random action is considered. It may be a removal of a vertex or an edge which does not separate graph or addition of an extra vertex or an edge neighboring existing graph. If the subgraph is empty all vertices are considered as candidates to form a subgaph. After candidate is chosen two subgraph scores are considered: for a new subgraph and for an old one. Simulated annealing operates with a notion of temperature. The candidate action is accepted with probability: p(S'|S) = exp(-E / T), where E is weight difference between subgraphs and T is current temperature.

Temperature is calculated in each iteration. in `mwcsr` there are two temperature schedules supported. So called Boltmann annealing uses the formula: T(k) = T0 / (ln(1 + k)), while in case of fast annealing this one is used: T(k) = T0 / k, where k is iteration number.

To tune the algorithm it is useful to realize how typical changes in the goal function for single actions are distributed. Calculating the acceptance probabilities at initial temperature and final temperatures may help to choose schedule and temperatures.

Value

An object of class `mwcs_solver`

See Also

rnc_solver will probably be a better choice with minimal tuning necessary

---

`bionet_example` *Example MWCS instance obtained from BioNet package tutorial*

Description

Example MWCS instance obtained from BioNet package tutorial

Usage

```
bionet_example
```

Format

An object of class `igraph` of length 2559.

gam_example *GAM instance for MWCS problem*

Description

A dataset containing some real-world instances appeared in network enrichment analysis tool Shiny GAM (doi:10.1093/nar/gkw266).

Usage

```
gam_example
```

Format

A vector of named vertex-weighted igraph instances

Source

http://dimacs11.zib.de/instances/MWCS-GAM.zip

gatom_example *Example of graph from which an SGMWCS instance can be obtained*

Description

The graph is based on `gatom` package

Usage

```
gatom_example
```

Format

An object of class `igraph` of length 194.

get_instance_type *Check the type and the validity of an MWCS instance*

Description

Check the type and the validity of an MWCS instance

Usage

```
get_instance_type(instance)
```

Arguments

`instance`	`igraph` object, containing an instance to be checked

Value

A list with members `type` containing the type of the instance, `valid` – boolean flag indicating whether the instance is valid or not, `errors` – a character vector containing the error messages

A list with two fields: the type of the instance with which it will be treated by solve_mwcsp function and boolean showing validness of the instance.

Examples

```
data(mwcs_example)
get_instance_type(mwcs_example)
```

get_weight *Calculate weight of the solution. MWCS, GMWCS and SGMWCS instances are supported*

Description

Calculate weight of the solution. MWCS, GMWCS and SGMWCS instances are supported

Usage

```
get_weight(solution)
```

Arguments

`solution`	Either `mwcsp_solution` or ‘igraph“ object representing the solution

Value

Weight of the subgraph

gmwcs_example *Example GMWCS instance*

Description

Instance is based on `gatom` package.

Usage

```
gmwcs_example
```

Format

An object of class `igraph` of length 194.

gmwcs_small_instance *Small example of GMWCS instance for demonstration purposes.*

Description

Small example of GMWCS instance for demonstration purposes.

Usage

```
gmwcs_small_instance
```

Format

An object of class `igraph` of length 5.

mwcs_example *Example MWCS instance*

Description

Instance is based on `gatom` package.

Usage

```
mwcs_example
```

Format

An object of class `igraph` of length 194.

mwcs_small_instance — *Small example of MWCS instance for demonstration purposes.*

Description

Small example of MWCS instance for demonstration purposes.

Usage

```
mwcs_small_instance
```

Format

An object of class `igraph` of length 5.

normalize_sgmwcs_instance — *Helper function to convert an `igraph` object into a proper SGMWCS instance*

Description

This function generates new `igraph` object with additional `signals` field added. The way the instance is constructed is defined by the function parameters. Nodes and edges are grouped separately, grouping columns are defined by `nodes.group.by` and `edges.group.by` arguments. `group.only.positive` param specifies whether to group only positive-weighted (specified by `nodes/edges.weight.column`) nodes and edges.

Usage

```
normalize_sgmwcs_instance(
  g,
  nodes.weight.column = "weight",
  edges.weight.column = "weight",
  nodes.group.by = "signal",
  edges.group.by = "signal",
  group.only.positive = TRUE
)
```

Arguments

`g` Graph to convert

`nodes.weight.column`
Nodes column name (e.g. weight, score, w) for scoring

`edges.weight.column`
: Edges column name for scoring

`nodes.group.by` Nodes grouping column (e.g. signal, group, class)

`edges.group.by` Edges grouping column

`group.only.positive`
: Whether to group only positive-scored nodes/edges#'

Value

An `igraph` object with proper attributes set.

Examples

```
data("gatom_example")
normalize_sgmwcs_instance(gatom_example)
```

`parameters` *The method returns all parameters supported by specific solver*

Description

The method returns all parameters supported by specific solver

Usage

```
parameters(solver)
```

Arguments

`solver` a solver object

Value

A table containing parameter names and possible values for each parameter.

rmwcs_solver *Generate a rmwcs solver*

Description

The method is based on relax-and-cut approach and allows to solve Maximum Weight Subgraph Probleam and its budget and cardinality variants. By constructing lagrangian relaxation of MWCS problem necessary graph connectivity constraints are introduced in the objective function giving upper bound on the weight of the optimal solution. On the other side, primal heuristic uses individul contribution of the variables to lagrangian relaxation to find possible solution of the initial problem. The relaxation is then optimized by using iterative subgradient method.

Usage

```
rmwcs_solver(
  timelimit = 1800L,
  max_iterations = 1000L,
  beta_iterations = 5L,
  separation = "strong",
  start_constraints = TRUE,
  pegging = TRUE,
  max_age = 10,
  sep_iterations = 10L,
  sep_iter_freeze = 50L,
  heur_iterations = 10L,
  subgradient = "classic",
  beta = 2,
  verbose = FALSE
)
```

Arguments

`timelimit`	Timelimit in seconds
`max_iterations`	Maximum number of iterations
`beta_iterations`	Number of nonimproving iterations until beta is halved
`separation`	Separation: "strong" or "fast"
`start_constraints`	Whether to add flow-conservation/degree constraints at start
`pegging`	variable fixing
`max_age`	number of iterations in aging procedure for non-violated cuts
`sep_iterations`	Frequency of separating cuts (in iterations)
`sep_iter_freeze`	Number of iterations when a newly separated cut is anaffected by subgradient algorithm.

heur_iterations
: Frequency of calling heuristic method (in iterations)

subgradient
: Subgradient: "classic", "average", "cft"

beta
: Initial step size of subgradient algorithm

verbose
: Should the solving progress and stats be printed?

Details

One iteration of algorithm includes solving lagrangian relaxation and updating lagrange multipliers. It may also contain cuts (or connectivity constraints) separation process, run of heuristic method, variable fixing routine. The initial step size for subgradient method can be passed as `beta` argument. If there is no improvement in upper bound in consequtive `beta_iterations` iterations the step size is halved. There are three possible strategies for updating multipliers. See the references section for the article where differences are discussed.

Some initial cuts are added at the start of the algorithm if `start_constraints` is set to `TRUE`. Other constraints are separated on the fly and are unaffected in the next `sep_iter_freeze` iterations of the subgradient mehod. Then the corresponding lagrange mutipliers are updated from iteration to iteration. Aging procedure for cuts is incorporated in the algorithm meaning constraint multipliers are updated for non-violated cuts for up to `max_age` iterations from the point where a cut was violated last time. There are two separation methods implemented: fast and strong, where tha latter is supposed to minimize number of variables used in generated constraint while in the former case there is no need to explore whole graph to construct a constraint.

A variant of MST approximation of PCSTP is used as Primal Heuristic. See references for more details.

The frequences of running separation process and heuristic are specified in `sep_iterations` and `heur_iterations`.

Value

An object of class `mwcs_solver`.

References

Álvarez-Miranda E., Sinnl M. (2017) "A Relax-and-Cut framework for large-scale maximum weight connected subgraph problems" doi:10.1016/j.cor.2017.05.015

`rnc_solver` *Construct relax-and-cut SGMWCS solver*

Description

The solver is based on the same approach as rmwcs solver. Modifications to the original scheme are introduced to tackle problems arising with introduction of edge weights and signals. It is recommended to use rmwcs solver to solve MWCS problems, while due to differences in primal heuristic it may be a good pratice to run both solvers on the same problem.

Usage

```
rnc_solver(
  max_iterations = 1000L,
  beta_iterations = 50L,
  heur_iterations = 10L,
  sep_iterations = 10L,
  verbose = FALSE
)
```

Arguments

`max_iterations`	Maximum number of iterations
`beta_iterations`	Number of nonimproving iterations until beta is halved
`heur_iterations`	Frequency of calling heuristic method (in iterations)
`sep_iterations`	Frequency of separating cuts (in iterations)
`verbose`	Should the solving progress and stats be printed?

Value

An object of class `mwcs_solver`

See Also

rmwcs_solver

`scipjack_solver` *Construct a SCIP-jack solver*

Description

This solver requires STP extension of SCIP-jack solver. To use this class you first need to download and build `SCIP-jack` and `SCIPSTP` application.

Usage

```
scipjack_solver(scipstp_bin, config_file = NULL)
```

Arguments

`scipstp_bin`	path to `scipstp binary`.
`config_file`	scipstp-formatted file. Parameters list is accessible at Official SCIP website.

Details

You can access solver directly using `run_scip` function. See example.

References

Rehfeldt D., Koch T. (2019) "Combining NP-Hard Reduction Techniques and Strong Heuristics in an Exact Algorithm for the Maximum-Weight Connected Subgraph Problem." doi:10.1137/17M1145963

Examples

```
## Not run:
data("bionet_example")
scip <- scipjack_solver(scipstp_bin='/path/to/scipoptsuite/build/bin/applications/scipstp')
sol <- solve_mwcsp(scip, bionet_example)

## End(Not run)
```

`set_parameters` *Sets values of specific parameters*

Description

Sets values of specific parameters

Usage

```
set_parameters(solver, ...)
```

Arguments

`solver`	a solver
`...`	listed parameter names and values assigned to them

Value

The solver with parameters changed.

`sgmwcs_example` *Example SGMWCS instance*

Description

Instance is based on `gatom` package.

Usage

```
sgmwcs_example
```

Format

An object of class `igraph` of length 194.

sgmwcs_small_instance *Small example of SGMWCS instance for demonstration purposes.*

Description

Small example of SGMWCS instance for demonstration purposes.

Usage

```
sgmwcs_small_instance
```

Format

An object of class `igraph` of length 6.

solve_mwcsp *Solves a MWCS instance.*

Description

Generic function for solving MWCS instances using solvers collected in the package.

Usage

```
solve_mwcsp(solver, instance, ...)

## S3 method for class 'virgo_solver'
solve_mwcsp(solver, instance, ...)

## S3 method for class 'rmwcs_solver'
solve_mwcsp(solver, instance, max_cardinality = NULL, budget = NULL, ...)

## S3 method for class 'rnc_solver'
solve_mwcsp(solver, instance, ...)

## S3 method for class 'simulated_annealing_solver'
solve_mwcsp(solver, instance, warm_start, ...)

## S3 method for class 'scipjack_solver'
solve_mwcsp(solver, instance, ...)
```

Arguments

solver	a solver object returned by rmwcs_solver, annealing_solver, rnc_solver or virgo_solver.
instance	an MWCS instance, an igraph object with problem-related vertex, edge and graph attributes. See details.
...	other arguments to be passed.
max_cardinality	integer maximum number of vertices in solution.
budget	numeric maximum budget of solution.
warm_start	warm start solution, an object of the class mwcsp_solution.

Details

MWCS instance here is represented as an undirected graph, an `igraph` object. The package supports four types of instances: Simple MWCS, Generalized MWCS, Budget MWCS, signal MWCS problems. All the necessary weights and costs are passed by setting vertex and edge attributes. See get_instance_type to check if the `igraph` object is a correct MWCS instance. For Simple MWCS problem numeric vertex attribute `weight` must be set. For generalized version `weights` can be provided for edges. For budget version of the problem in addition to vertex weights it is required that `igraph` object would have `budget_cost` vertex attribute with positive numeric values.

Signal MWCS instance is quite different. There is no `weight` attribute for neither vertices nor edges. Instead, vertex and edge attribute `signal` should be provided with signal names. A numeric vector containing weights for the signals should be assigned to graph attribute `signals`.

See vignette for description of the supported problems. See `igraph` package documentation for more details about getting/setting necesasry attributes.

Value

An object of class `mwcsp_solution` consisting of resulting subgraph, its weight and other information about solution provided.

Examples

```
library(igraph)

# for a MWCS instance

data(mwcs_example)
head(V(mwcs_example)$weight)

# for a GMWCS instance
data(gmwcs_example)
head(E(gmwcs_example)$weight)

# for a SGMWCS instance
data(sgmwcs_example)
head(V(sgmwcs_example)$signal)
head(E(sgmwcs_example)$signal)
```

```
head(sgmwcs_example$signals)
```

`timelimit<-` *Sets time limitation for a solver*

Description

Sets time limitation for a solver

Usage

```
timelimit(x) <- value
```

Arguments

x	a variable name.
value	a value to be assigned to x.

Value

The solver with new timelimit set.

`virgo_solver` *Construct a virgo solver*

Description

This solver uses reformulation of MWCS problem in terms of mixed integer programming. The later problem can be efficiently solved with commercial optimization software. Exact version of solver uses CPLEX and requires it to be installed. CPLEX 12.7.1 or higher is required.

Usage

```
virgo_solver(
  cplex_dir,
  threads = parallel::detectCores(),
  timelimit = NULL,
  penalty = 0,
  memory = "2G",
  log = 0,
  cplex_bin = NULL,
  cplex_jar = NULL,
  mst = FALSE,
  dryrun = FALSE,
  jvmargs = NULL
)
```

Arguments

`cplex_dir`	a path to dir containing cplex_bin and cplex_jar, setting this to NULL sets `mst`` param to TRUE`‘
`threads`	number of threads for simultaneous computation
`timelimit`	maximum number of seconds to solve the problem
`penalty`	additional edge penalty for graph edges
`memory`	maximum amount of memory(-Xmx flag)
`log`	verbosity level
`cplex_bin`	a path to cplex binary dir
`cplex_jar`	a path to cplex jar file
`mst`	whether to use approximate MST solver, no CPLEX files required with this parameter is set to `TRUE`
`dryrun`	if set to `TRUE` only prints the solver command, without actually running it
`jvmargs`	character vector with additional arguments for Java Virtual Machine

Details

The solver currently does not support repeated negative signals, i.e. every negative signal should be present only once among all edges and vertices.

You can access solver directly using `run_main` function. See example.

Value

An object of class `mwcs_solver`.

References

Loboda A., Artyomov M., and Sergushichev A. (2016) "Solving generalized maximum-weight connected subgraph problem for network enrichment analysis" doi:10.1007/9783319436814_17

Examples

```
data("sgmwcs_small_instance")
approx_vs <- virgo_solver(mst=TRUE, threads = 1)
approx_vs$run_main("-h")
sol <- solve_mwcsp(approx_vs, sgmwcs_small_instance)
## Not run:
vs <- virgo_solver(cplex_dir='/path/to/cplex')
sol <- solve_mwcsp(approx_vs, sgmwcs_example)

## End(Not run)
```

Index