cspy is an open source Python package that gathers some algorithms to solve the (resource) Constrained Shortest Path problem.

By setting different options when calling the algorithms, one can have up to five different algorithms.

- Monodirectional forward labeling algorithm;
- Monodirectional backward labeling algorithm;
- Bidirectional labeling algorithm with static halfway point;
- Bidirectional labeling algorithm with dynamic halfway point (Tilk et al 2017);
- Heuristic Tabu search;
- Particle Swarm Optimization with combined Local and Global Expanding Neighborhood Topology (PSOL-GENT) (Marinakis et al 2017).

Features implemented include: generic resource extension functions (Inrich 2005) (not restricted to additive resources), generic resource consumptions (not restricted to non-negative values), and, increased efficiency (when compared to other implementations of monodirectional algorithms).

cspy is installable via pip (see Getting Started) or the source code is made available here.
1.1 Installation

You can install the latest release of cspy from PyPi by:

```bash
pip install cspy
```

Alternatively, you can clone the latest development version of cspy from the repository:

```bash
git clone https://github.com/torressa/cspy
```

1.1.1 Requirements

The requirements for running cspy are:

- **NetworkX**: Graph manipulation and creation.
- **NumPY**: Array manipulation.
Here is the guide of how to use the cspy package.

## 2.1 Initialisations

In order to use cspy package and the algorithms within, first, one has to create a directed graph on which to apply the algorithms.

To do so, we make use of the well-known networkx package. To be able to apply resource constraints, we have the following input graph requirements,

- Graph must be a networkx.DiGraph;
- Graph must have an attribute n_res (set when initialising the graph) which determines the number of resources we are considering for the particular problem;
- Graph must have a single Source and Sink nodes with no incoming or outgoing edges respectively;
- Edges must have res_cost (of type numpy.array) and weight attributes.

For example,

```python
>>> from networkx import DiGraph
>>> from numpy import array
>>> G = DiGraph(directed=True, n_res=2)
>>> G.add_edge('Source', 'A', res_cost=array([1]), weight=1.0)
>>> G.add_edge('A', 'B', res_cost=array([1]), weight=1.0)
>>> G.add_edge('B', 'Sink', res_cost=array([1]), weight=1.0)
```

## 2.2 Algorithms

Have a look and choose which algorithm you’d like to use. In order to run the algorithms create a appropriate algorithm instance (with the appropriate inputs) and call run().
• BiDirectional: Bidirectional and monodirectional algorithms
• Tabu Heuristic Tabu Search
• GreedyElim Greedy Elimination Procedure
• GRASP GRASP
• PSOLGENT Particle Swarm Optimization with combined Local and Global Expanding Neighborhood Topology (PSOLGENT)

Please see individual algorithm documentation for examples.

2.3 Prerequisites

For the BiDirectional algorithm, there is a number of assumptions required (Tilk et al 2017).

1. The first resource must be a monotone resource;
2. The resource extension are invertible.

For assumption 1, resource can be either artificial, such as the number of edges in the graph, or real like for example time. Clearly, these are problem-dependent and if your problem doesn’t seem to have a monotone resource, it is easier to use an artificial one.

This allows for the monotone resource to comparable for the forward and backward directions. In practice, this means, that $n_{res} = \text{len}(\text{max}_{res}) = \text{len}(\text{min}_{res}) \geq 2$, and that the first element in both edge attributes and input limits is the monotone resource.

For assumption 2, if resource extension functions are additive, these are easily invertible.

2.4 REFs

Additive resource extension functions (REFs), are implemented by default in all the algorithms. However, you can use your own custom REFs. For theoretical information on what REFs are we refer you to the paper by Inrich 2005.

Practically, a custom REF will need two inputs: $res$, a cumulative resource array, and $edge$, an edge to consider for the extension of the current partial path. This function will be called every time the algorithms wish to consider and edge as part of the shortest path.

As an example, the following function would be valid:

```python
from numpy import array

def REF_CUSTOM(cumulative_res, edge):
    new_res = array(cumulative_res)
    # your filtering criteria that changes the elements of new_res
    # For example:
    head_node, tail_node = edge[0:2]
    if head_node != tail_node:
        new_res[0] = 0
    else:
        new_res[0] = new_res[0] + 1
    return new_res
```

Your custom REF can then be passed with this format, into the algorithm of choice using the REF argument (see individual algorithms for details). Note that for the BiDirectional algorithm, due to the properties of the algorithm, if you want to use this feature, you have to pass two custom REFs: one for the forward search and one for the backward
search. Where the backward REF has to be the inverse of the forward REF, otherwise the algorithm will not return a meaningful path (Tilk et al 2017).
Implementation of the bidirectional labeling algorithm with dynamic half-way point (Tilk 2017). Depending on the range of values for U, L, we get four different algorithms. See self.name_algorithm and Notes.

3.1 Parameters

G [object instance nx.Digraph()] must have n_res graph attribute and all edges must have res_cost attribute.

max_res [list of floats] \([H_F, M_1, M_2, ..., M_{n\_res}]\) upper bounds for resource usage (including initial forward stopping point). We must have \(\text{len}(\text{max_res}) \geq 2\).

min_res [list of floats] \([H_B, L_1, L_2, ..., L_{n\_res}]\) lower bounds for resource usage (including initial backward stopping point). We must have \(\text{len}(\text{min_res}) = \text{len}(\text{max_res}) \geq 2\)


3.2 Returns

list nodes in shortest path obtained.

3.3 Notes

The input graph must have a n_res attribute which must be \(\geq 2\). The edges in the graph must all have a res_cost attribute.
According to the inputs, four different algorithms can be implemented, if you'd like to check which algorithm your running given your resource limits, run .name_algorithm(U, L) for a log with the classification. If direction is not given the absolute resource limits have to be given:

```
BiDirectional.name_algorithm(U, L)()
```

**U** [float, optional] Upper bound for monotone resource (for algorithm classification purposes see Notes).

**L** [float, optional] Lower bound for monotone resource (for algorithm classification purposes see Notes).

According to these, we have,

- \( H_F = H_B > U \) or **direction** = ‘forward’: Monodirectional forward labeling algorithm
- \( L < H_F = H_B < U \): Bidirectional labeling algorithm with static halfway point
- \( H_F = H_B < L \) or **direction** = ‘backward’: Monodirectional backward labeling algorithm
- \( U = H_F > H_B = L \): Bidirectional labeling algorithm with dynamic halfway point.

### 3.4 Example

To run the algorithm, create a BiDirectional instance and call run.

```python
>>> import cspy
>>> from networkx import DiGraph
>>> from numpy import array

>>> G = DiGraph(directed=True, n_res=2)
>>> G.add_edge('Source', 'A', res_cost=array([1, 2]), weight=0)
>>> G.add_edge('A', 'B', res_cost=array([1, 0.3]), weight=0)
>>> G.add_edge('A', 'C', res_cost=array([1, 0.1]), weight=0)
>>> G.add_edge('B', 'C', res_cost=array([1, 3]), weight=-10)
>>> G.add_edge('B', 'Sink', res_cost=array([1, 2]), weight=10)
>>> G.add_edge('C', 'Sink', res_cost=array([1, 10]), weight=0)

>>> max_res, min_res = [4, 20], [1, 0]

>>> path = BiDirectional(G, max_res, min_res, direction='both').run()

>>> print(path)
['Source', 'A', 'B', 'C', 'Sink']
```
Simple Tabu-esque algorithm for the (resource) constrained shortest path problem.

### 4.1 Parameters

- **G** [object instance `nx.DiGraph()`] must have `n_res` graph attribute and all edges must have `res_cost` attribute.
- **max_res** [list of floats] \([M_1, M_2, ..., M_{n_{res}}]\) upper bounds for resource usage (including initial forward stopping point).
- **min_res** [list of floats] \([L_1, L_2, ..., L_{n_{res}}]\) lower bounds for resource usage (including initial backward stopping point). We must have \(\text{len}(\text{min_res}) = \text{len}(\text{max_res})\).

### 4.2 Returns

**path** [list] nodes in shortest path obtained.

### 4.3 Notes

The input graph must have a `n_res` attribute. The edges in the graph must all have a `res_cost` attribute. See `Using cspy`.

### 4.4 Example

To run the algorithm, create a `Tabu` instance and call `run`.
```python
>>> from cspy import Tabu
>>> from networkx import DiGraph
>>> from numpy import array

>>> G = DiGraph(directed=True, n_res=2)
>>> G.add_edge('Source', 'A', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('Source', 'B', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('A', 'C', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('B', 'C', res_cost=array([2, 1]), weight=-1)
>>> G.add_edge('C', 'D', res_cost=array([1, 1]), weight=-1)
>>> G.add_edge('D', 'E', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('D', 'F', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('F', 'Sink', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('E', 'Sink', res_cost=array([1, 1]), weight=1)

>>> max_res, min_res = [5, 5], [0, 0]
>>> path = Tabu(G, max_res, min_res).run()

>>> print(path)
['Source', 'A', 'C', 'D', 'E', 'Sink']
```
Simple Greedy elimination algorithm for the (resource) constrained shortest path problem. The algorithm solves a standard shortest path problem and eliminates resource infeasible edges iteratively until a resource feasible path is found.

5.1 Parameters

- **G** [object instance `nx.DiGraph()`] must have `n_res` graph attribute and all edges must have `res_cost` attribute.
- **max_res** [list of floats] \([M_1, M_2, ..., M_{n_res}]\) upper bounds for resource usage (including initial forward stopping point).
- **min_res** [list of floats] \([L_1, L_2, ..., L_{n_res}]\) lower bounds for resource usage (including initial backward stopping point). We must have \(\text{len(min_res)} = \text{len(max_res)}\).
- **return_G** [bool, optional] whether or not you’d like the resulting graph returned

5.2 Returns

- **path** [list] nodes in shortest path obtained.

5.3 Raises

- **Exception** if no resource feasible path is found
5.4 Notes

The input graph must have a \texttt{n\_res} attribute. The edges in the graph must all have a \texttt{res\_cost} attribute. See Using \texttt{cspy}

5.5 Example

To run the algorithm, create a \texttt{GreedyElim} instance and call \texttt{run}.

```python
>>> from cspy import GreedyElim
>>> G = nx.DiGraph(directed=True, n_res=2)
>>> G.add_edge('Source', 'A', res_cost=[1, 1], weight=1)
>>> G.add_edge('Source', 'B', res_cost=[1, 1], weight=1)
>>> G.add_edge('A', 'C', res_cost=[1, 1], weight=1)
>>> G.add_edge('B', 'C', res_cost=[2, 1], weight=-1)
>>> G.add_edge('C', 'D', res_cost=[1, 1], weight=-1)
>>> G.add_edge('D', 'E', res_cost=[1, 1], weight=1)
>>> G.add_edge('D', 'F', res_cost=[1, 1], weight=1)
>>> G.add_edge('F', 'Sink', res_cost=[1, 1], weight=1)
>>> G.add_edge('E', 'Sink', res_cost=[1, 1], weight=1)
>>> max_res, min_res = [5, 5], [0, 0]
>>> path = GreedyElim(G, max_res, min_res).run()
>>> print(path)
['Source', 'A', 'C', 'D', 'E', 'Sink']
```

### 6.1 Parameters

- **G** [object instance `nx.DiGraph()`] must have `n_res` graph attribute and all edges must have `res_cost` attribute. Also, the number of nodes must be $\geq 5$.
- **max_res** [list of floats] $[M_1, M_2, ..., M_{n_res}]$ upper bounds for resource usage.
- **min_res** [list of floats] $[L_1, L_2, ..., L_{n_res}]$ lower bounds for resource usage.
- **REF** [function, optional] Custom resource extension function. See **REFs** for more details. Default : additive.
- **max_iter** [int, optional] Maximum number of iterations for algorithm. Default : 100.
- **alpha** [float, optional] Greediness factor 0 (random) $\rightarrow$ 1 (greedy). Default : 0.2.

### 6.2 Returns

- **path** [list] nodes in resource feasible shortest path obtained.

### 6.3 Raises

**Exception** if no resource feasible path is found
6.4 Notes

The input graph must have a \texttt{n\_res} attribute. The edges in the graph must all have a \texttt{res\_cost} attribute. Also, we must have $\text{len}(\text{min\_res}) = \text{len}(\text{max\_res})$. See Using \texttt{cspy}

6.5 Example

To run the algorithm, create a \texttt{GRASP} instance and call \texttt{run}.

```python
>>> from cspy import GRASP
>>> from networkx import DiGraph
>>> from numpy import array

>>> G = DiGraph(directed=True, n_res=2)
>>> G.add_edge('Source', 'A', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('Source', 'B', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('Source', 'C', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('A', 'C', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('A', 'E', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('A', 'F', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('B', 'C', res_cost=array([2, 1]), weight=-1)
>>> G.add_edge('B', 'F', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('B', 'E', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('C', 'D', res_cost=array([1, 1]), weight=-1)
>>> G.add_edge('D', 'E', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('D', 'F', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('D', 'Sink', res_cost=array([10, 10]), weight=10)
>>> G.add_edge('F', 'Sink', res_cost=array([10, 1]), weight=1)
>>> G.add_edge('E', 'Sink', res_cost=array([1, 1]), weight=1)

>>> path = GRASP(G, [5, 5], [0, 0], max_iter=50, max_localiter=10).run()

>>> print(path)
['Source', 'A', 'C', 'D', 'E', 'Sink']
```
Particle Swarm Optimization with combined Local and Global Expanding Neighborhood Topology (PSOLGENT) algorithm for the (resource) constrained shortest path problem (Marinakis et al 2017).

Given the nature of our problem we have set the default parameters of the algorithm as suggested in the paper mentioned.

Code adapted from Solid.

### 7.1 Parameters

- **G** [object] : nx.Digraph() must have n_res graph attribute and all edges must have res_cost attribute.
- **max_res** [list of floats] : $[M_1, M_2, ..., M_{n_{res}}]$ upper bounds for resource usage.
- **min_res** [list of floats] : $[L_1, L_2, ..., L_{n_{res}}]$ lower bounds for resource usage.
- **max_iter** [int, optional] : Maximum number of iterations for algorithm. Default: 100.
- **swarm_size** [int, optional] : number of members in swarm. Default: 50.
- **member_size** [int, optional] : number of components per member vector. Default: len(G.nodes()).
- **upper_bound** [list of floats, optional] : list of upper bounds. Default: numpy.ones(member_size) (all nodes in path).
- **c1** [float, optional] : constant for 1st term in the velocity equation. Default: 1.35.
- **c2** [float, optional] : constant for 2nd term in the velocity equation. Default: 1.35.

seed  [None or int or numpy.random.RandomState instance, optional] seed for PSOLGENT class. Default : None (which gives a single value numpy.random.RandomState).

7.2 Returns

path  [list] nodes in resource feasible shortest path obtained.

7.3 Raises

Exception  if no resource feasible path is found

7.4 Notes

The input graph must have a n_res attribute. The edges in the graph must all have a res_cost attribute. Also, we must have len(min_res) = len(max_res). See Using cspy.

This algorithm requires a consistent sorting of the nodes in the graph. Please see comments and edit the function _sort_nodes accordingly.

7.5 Example

To run the algorithm, create a PSOLGENT instance and call run.

```python
>>> from cspy import PSOLGENT
>>> from networkx import DiGraph
>>> from numpy import zeros, ones, array

>>> G = DiGraph(directed=True, n_res=2)
>>> G.add_edge('Source', 'A', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('Source', 'B', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('Source', 'C', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('A', 'C', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('A', 'E', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('A', 'F', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('B', 'C', res_cost=array([2, 1]), weight=-1)
>>> G.add_edge('B', 'F', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('B', 'E', res_cost=array([10, 1]), weight=10)
>>> G.add_edge('C', 'D', res_cost=array([1, 1]), weight=-1)
>>> G.add_edge('D', 'E', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('D', 'F', res_cost=array([1, 1]), weight=1)
>>> G.add_edge('D', 'Sink', res_cost=array([10, 10]), weight=10)
>>> G.add_edge('F', 'Sink', res_cost=array([10, 1]), weight=1)
>>> G.add_edge('E', 'Sink', res_cost=array([1, 1]), weight=1)

>>> n_nodes = len(self.J.nodes())
>>> path = PSOLGENT(G, [5, 5], [0, 0],
                        max_iter=200,
                        swarm_size=50,
                        member_size=n_nodes,
                        neighbourhood_size=50).run()
```

(continues on next page)
>>> print(path)
['Source', 'A', 'C', 'D', 'E', 'Sink']
cspy.check_and_preprocess(preprocess, G, max_res=None, min_res=None, REF_forward=None, REF_backward=None, direction=None, algorithm=None)

Checks whether inputs and the graph are of the appropriate types and have the required properties. For non-specified REFs, removes nodes that cannot be reached due to resource limits.

**preprocess**  [bool] enables preprocessing routine.

**G**  [object instance nx.DiGraph()] must have n_res graph attribute and all edges must have res_cost attribute.

**max_res**  [list of floats, optional] \([L, M_1, M_2, \ldots, M_{n_res}]\) upper bound for resource usage. We must have \(\text{len}(\text{max}_\text{res}) \geq 2\)

**min_res**  [list of floats, optional] \([U, L_1, L_2, \ldots, L_{n_res}]\) lower bounds for resource usage. We must have \(\text{len}(\text{min}_\text{res}) = \text{len}(\text{max}_\text{res}) \geq 2\)

**REF_forward, REF_backward**  [function, optional] Custom resource extension function. See REFS for more details. Default: additive, subtractive.


**Returns** If preprocess, returns preprocessed graph G if no exceptions are raised, otherwise doesn’t return anything.

**Raises** Raises exceptions if incorrect input is given. If multiple exceptions are raised, and exception with a list of exceptions is raised.
class cspy.Label

Label object that allows comparison and the modelling of dominance relations

- **weight** [float] cumulative edge weight
- **node** [string] name of last node visited
- **res** [list] cumulative edge resource consumption
- **path** [list] all nodes in the path
Python Module Index

C

cspy.BiDirectional, 9
cspy.GRASP, 15
cspy.GreedyElim, 13
cspy.PSOLGENT, 17
cspy.Tabu, 11
Index

C
check_and_preprocess() (in module cspy), 21
cspy.BiDirectional (module), 9
cspy.GRASP (module), 15
cspy.GreedyElim (module), 13
cspy.PSOLGENT (module), 17
cspy.Tabu (module), 11

L
Label (class in cspy), 23