

The ROAR-NET API: Constructive Search

ROAR-NET Problem Modelling Code Fest

Andreia P. Guerreiro
INESC-ID, Portugal

Constructive Search: Greedy Algorithm



- Greedy construction:

Constructive Search: Greedy Algorithm



- Greedy construction:
 - Start with an empty solution \mathbf{s}

Constructive Search: Greedy Algorithm



- Greedy construction:
 - Start with an empty solution s
 - While s is not complete

Constructive Search: Greedy Algorithm



- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising

Constructive Search: Greedy Algorithm



- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising
 - Apply the move to (update) solution \mathbf{s}

Constructive Search: Greedy Algorithm

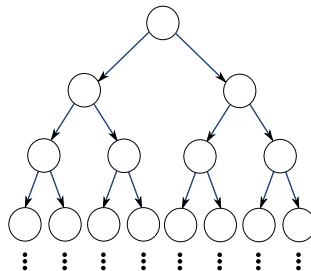


- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising
 - Apply the move to (update) solution \mathbf{s}
 - Return \mathbf{s}

Constructive Search: Greedy Algorithm

ROAR
NET

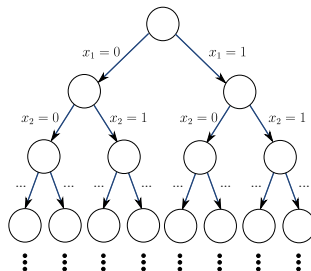
- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising
 - Apply the move to (update) solution \mathbf{s}
 - Return \mathbf{s}



Constructive Search: Greedy Algorithm

ROAR
NET

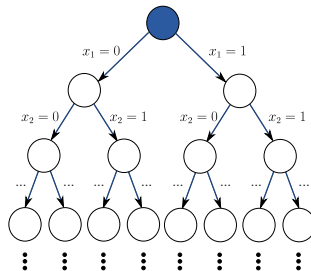
- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising
 - Apply the move to (update) solution \mathbf{s}
 - Return \mathbf{s}



Constructive Search: Greedy Algorithm

ROAR
NET

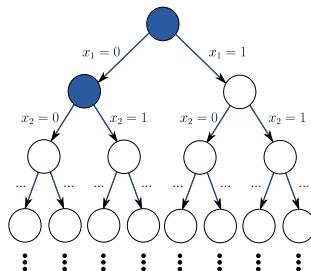
- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising
 - Apply the move to (update) solution \mathbf{s}
 - Return \mathbf{s}



Constructive Search: Greedy Algorithm

ROAR
NET

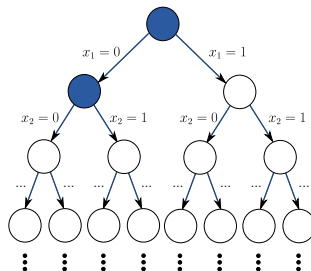
- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising
 - Apply the move to (update) solution \mathbf{s}
 - Return \mathbf{s}



Constructive Search: Greedy Algorithm

ROAR
NET

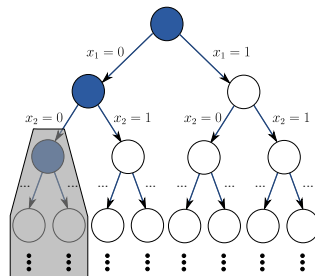
- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising
 - Apply the move to (update) solution \mathbf{s}
 - Return \mathbf{s}
- Lower/Upper bound information can be used to evaluate how promising a move is



Constructive Search: Greedy Algorithm

ROAR
NET

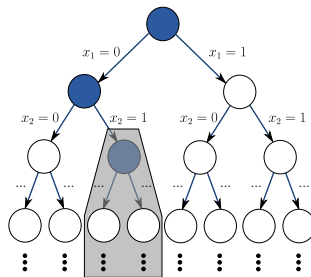
- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising
 - Apply the move to (update) solution \mathbf{s}
 - Return \mathbf{s}
- Lower/Upper bound information can be used to evaluate how promising a move is



Constructive Search: Greedy Algorithm

ROAR
NET

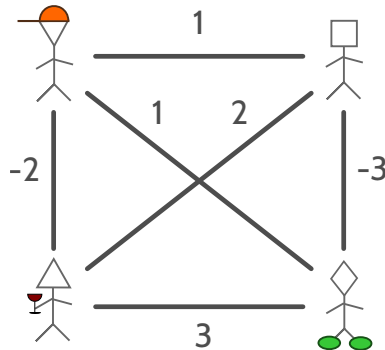
- Greedy construction:
 - Start with an empty solution \mathbf{s}
 - While \mathbf{s} is not complete
 - Select the construction move that seems to be the most promising
 - Apply the move to (update) solution \mathbf{s}
 - Return \mathbf{s}
- Lower/Upper bound information can be used to evaluate how promising a move is



Community Detection Problem

ROAR
NET

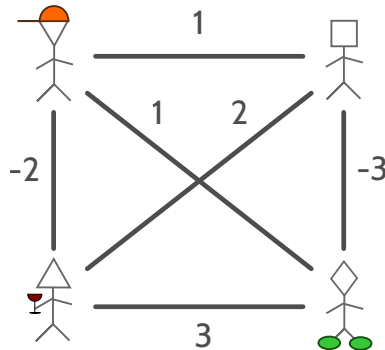
- A fully-connected undirected graph



Community Detection Problem

ROAR
NET

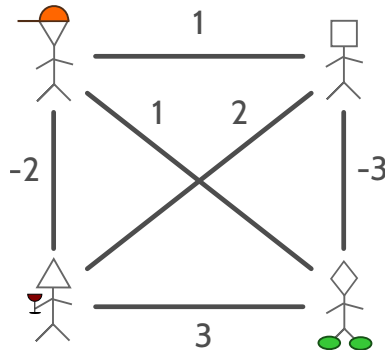
- A fully-connected undirected graph
 - Vertices represent users



Community Detection Problem

ROAR
NET

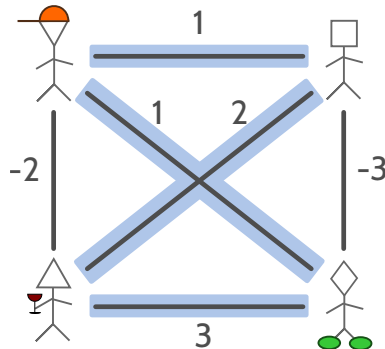
- A fully-connected undirected graph
 - Vertices represent users
 - Weighted edges represent the intensity of some attribute of their interaction



Community Detection Problem

ROAR
NET

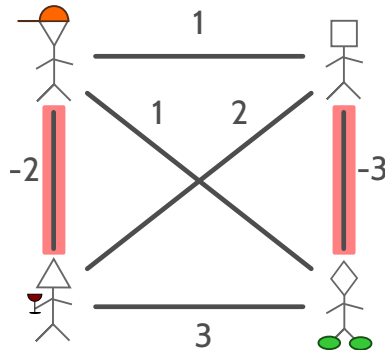
- A fully-connected undirected graph
 - Vertices represent users
 - Weighted edges represent the intensity of some attribute of their interaction
- Positive weight show affinity between users



Community Detection Problem

ROAR
NET

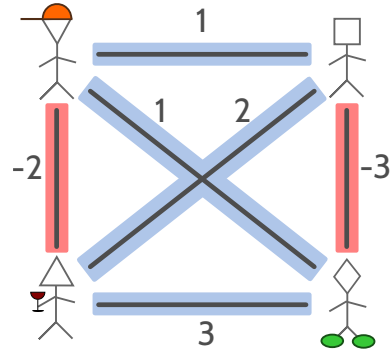
- A fully-connected undirected graph
 - Vertices represent users
 - Weighted edges represent the intensity of some attribute of their interaction
- Positive weight show affinity between users
- Negative edge weights indicate lack of affinity



Community Detection Problem

ROAR
NET

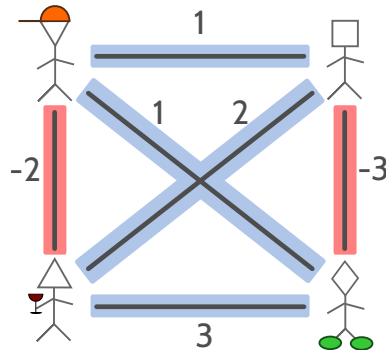
- A fully-connected undirected graph
 - Vertices represent users
 - Weighted edges represent the intensity of some attribute of their interaction
- Positive weight show affinity between users
- Negative edge weights indicate lack of affinity
- Groups of users connected mostly by positively weighted edges suggest the existence of a community involving those individuals



Community Detection Problem

FOAR
NET

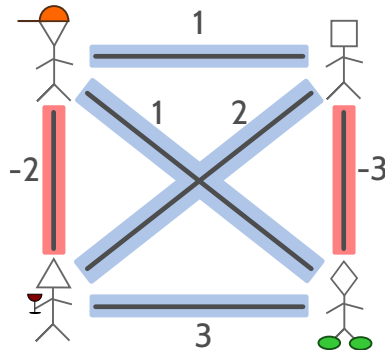
- A fully-connected undirected graph
 - Vertices represent users
 - Weighted edges represent the intensity of some attribute of their interaction
- Positive weight show affinity between users
- Negative edge weights indicate lack of affinity
- Groups of users connected mostly by positively weighted edges suggest the existence of a community involving those individuals
- Goal: Partition the vertices into subsets while maximising the total weight of the edges within the groups (cliques)



Community Detection Problem

ROAR
NET

- A fully-connected undirected graph
 - Vertices represent users
 - Weighted edges represent the intensity of some attribute of their interaction
- Positive weight show affinity between users
- Negative edge weights indicate lack of affinity
- Groups of users connected mostly by positively weighted edges suggest the existence of a community involving those individuals
- Goal: Partition the vertices into subsets while maximising the total weight of the edges within the groups (cliques)
 - Clique-partitioning problem



API functions to be implemented

ROAR
NET

Problem

```
empty_solution(Problem) : Solution  
construction_neighbourhood(Problem) : Neighbourhood
```

Solution

```
objective_value(Solution) : double[0..1]  
lower_bound(Solution) : double[0..1]
```

Neighbourhood

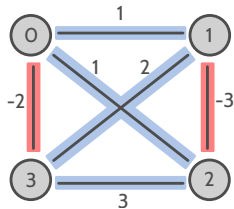
```
moves(Neighbourhood, Solution) : Move[0..*]
```

Move

```
apply_move(Move, Solution) : Solution  
lower_bound_increment(Move, Solution) : double[0..1]
```

Problem modelling: Load problem instances

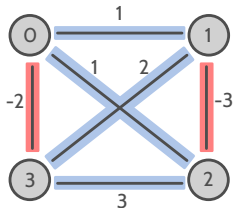
- Implement
 - Constructor of the **Problem** class
 - Method to load a problem instance



$$\begin{bmatrix} 0 & 1 & 1 & -2 \\ 1 & 0 & -3 & 2 \\ 1 & -3 & 0 & 3 \\ -2 & 2 & 3 & 0 \end{bmatrix}$$

Problem modelling: Load problem instances

- Implement
 - Constructor of the **Problem** class
 - Method to load a problem instance
- Input instance format
 - Number of vertices
 - Upper triangle part of the cost matrix



$$\begin{bmatrix} \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{-2} \\ 1 & \mathbf{0} & \mathbf{-3} & \mathbf{2} \\ 1 & -3 & \mathbf{0} & \mathbf{3} \\ -2 & 2 & 3 & \mathbf{0} \end{bmatrix}$$

```
4
0 1 1 -2
0 -3 2
0 3
0
```

Problem modelling: Construction neighbourhood



- Construction rule

Problem modelling: Construction neighbourhood



- Construction rule
 - Select (in increasing index order) the next unassigned vertex

Problem modelling: Construction neighbourhood



- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one

Problem modelling: Construction neighbourhood



- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$

Problem modelling: Construction neighbourhood



- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)

Problem modelling: Construction neighbourhood



- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index

Problem modelling: Construction neighbourhood



- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound

Problem modelling: Construction neighbourhood



- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices

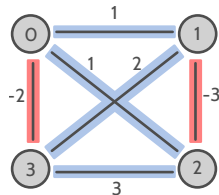
Problem modelling: Construction neighbourhood



- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound

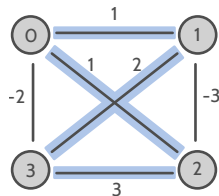
Problem modelling: Construction neighbourhood

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; *clique*[i] is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



Problem modelling: Construction neighbourhood

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



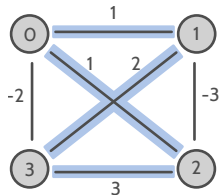
clique: []

k: 0

UB: 7

Problem modelling: Construction neighbourhood

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



clique: []

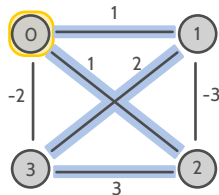
k: 0

UB: 7

Add move (0,0)

Problem modelling: Construction neighbourhood

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



clique: [0]

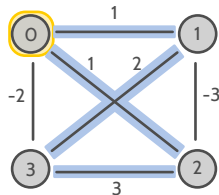
k: 1

UB: 7

Added move (0,0)

Problem modelling: Construction neighbourhood

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



clique: [0]

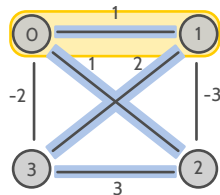
k: 1

UB: 7

Add move (1,0)

Problem modelling: Construction neighbourhood

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



clique: [0,0]

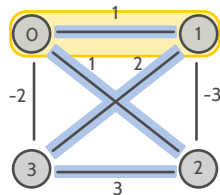
k: 1

UB: 7

Added move (1,0)

Problem modelling: Construction neighbourhood

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



clique: [0,0]

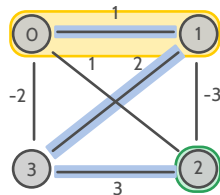
k: 1

UB: 7

Add move (2,1)

Problem modelling: Construction neighbourhood

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



clique: [0,0,1]

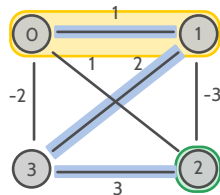
k: 2

UB: 6

Added move (2,1)

Problem modelling: Construction neighbourhood

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



clique: [0,0,1]

k: 2

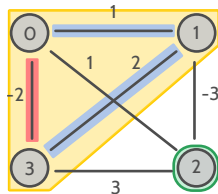
UB: 6

Add move (3,0)

Problem modelling: Construction neighbourhood

ROAR
NET

- Construction rule
 - Select (in increasing index order) the next unassigned vertex
 - Either assign to one of the k existing cliques or to a newly created one
 - Neighbourhood size: $k + 1$
- Move: (v, c)
 - v : vertex index
 - c : clique index
- Upper bound
 - Current total clique value plus the sum of all positive weights of edges incident on at least one unassigned vertices
- Solution representation
 - **clique**: A list; $clique[i]$ is the clique index of vertex i
 - **k**: Number of cliques
 - **ub**: Upper bound



clique: [0,0,1,0]

k: 2

UB: 1

Added move (3,0)

Problem modelling: Solution class



Problem

```
empty_solution(Problem) : Solution  
construction_neighbourhood(Problem) : Neighbourhood
```

Solution

```
objective_value(Solution) : double[0..1]  
lower_bound(Solution) : double[0..1]
```

Neighbourhood

```
moves(Neighbourhood, Solution) : Move[0..*]
```

Move

```
apply_move(Move, Solution) : Solution  
lower_bound_increment(Move, Solution) : double[0..1]
```

- Implement **Solution** constructor
- Implement API functions to create and evaluate solutions

Problem modelling: Neighbourhood class



Problem

```
empty_solution(Problem) : Solution  
construction_neighbourhood(Problem) : Neighbourhood
```

Solution

```
objective_value(Solution) : double[0..1]  
lower_bound(Solution) : double[0..1]
```

Neighbourhood

```
moves(Neighbourhood, Solution) : Move[0..*]
```

Move

```
apply_move(Move, Solution) : Solution  
lower_bound_increment(Move, Solution) : double[0..1]
```

- Implement **Neighbourhood** constructor
- Implement **Move** constructor
- Implement API functions related to neighbourhoods

Problem modelling: Move class



Problem

```
empty_solution(Problem) : Solution  
construction_neighbourhood(Problem) : Neighbourhood
```

Solution

```
objective_value(Solution) : double[0..1]  
lower_bound(Solution) : double[0..1]
```

Neighbourhood

```
moves(Neighbourhood, Solution) : Move[0..*]
```

Move

```
apply_move(Move, Solution) : Solution  
lower_bound_increment(Move, Solution) : double[0..1]
```

- Implement API functions related to (applying) moves

Problem modelling: lower (upper) bound increment



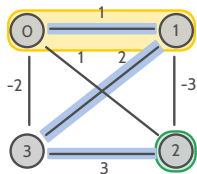
- `ub_inc`: Upper bound increment of move (v, c)

Problem modelling: lower (upper) bound increment

ROAR
NET

- **ub_inc**: Upper bound increment of move (v, c)

- Example: Move $(3, 0)$



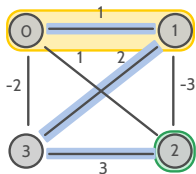
ub: 6

Problem modelling: lower (upper) bound increment

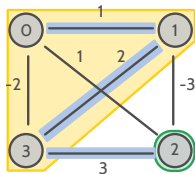
ROAR
NET

- **ub_inc**: Upper bound increment of move (v, c)
- Initialise **ub_inc** to 0

- Example: Move $(3, 0)$



ub: 6

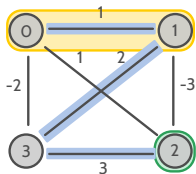


ub_inc: 0

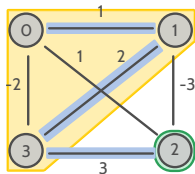
Problem modelling: lower (upper) bound increment

- **ub_inc**: Upper bound increment of move (v, c)
- Initialise **ub_inc** to 0
- For every assigned vertex v' (in **clique** list)

- Example: Move $(3, 0)$



ub: 6

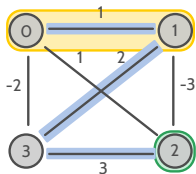


ub_inc: 0

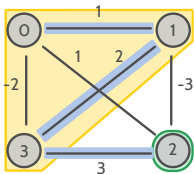
Problem modelling: lower (upper) bound increment

- **ub_inc**: Upper bound increment of move (v, c)
- Initialise **ub_inc** to 0
- For every assigned vertex v' (in **clique** list)
 - If v' is in clique c and the edge (v', v) has negative weight w

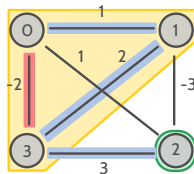
- Example: Move $(3, 0)$



ub: 6



ub_inc: 0

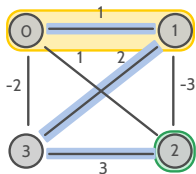


ub_inc: -2

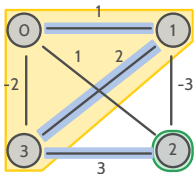
Problem modelling: lower (upper) bound increment

- **ub_inc**: Upper bound increment of move (v, c)
- Initialise **ub_inc** to 0
- For every assigned vertex v' (in **clique** list)
 - If v' is in clique c and the edge (v', v) has negative weight w
 - Then, add w to **ub_inc**

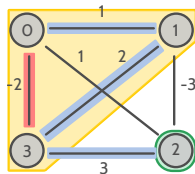
- Example: Move $(3, 0)$



ub: 6



ub_inc: 0

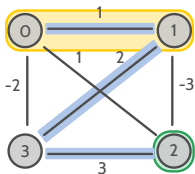


ub_inc: -2

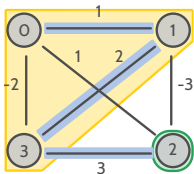
Problem modelling: lower (upper) bound increment

ROAR
NET

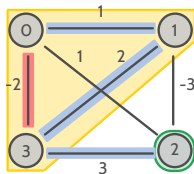
- **ub_inc**: Upper bound increment of move (v, c)
- Initialise **ub_inc** to 0
- For every assigned vertex v' (in **clique** list)
 - If v' is in clique c and the edge (v', v) has negative weight w
 - Then, add w to **ub_inc**
 - If v' is not in clique c and the edge (v', v) has positive weight w
- Example: Move $(3, 0)$



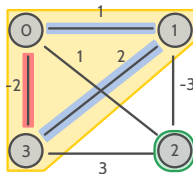
ub: 6



ub_inc: 0



ub_inc: -2

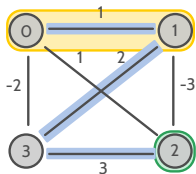


ub_inc: -5

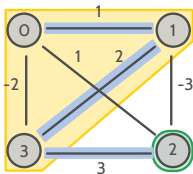
Problem modelling: lower (upper) bound increment

ROAR
NET

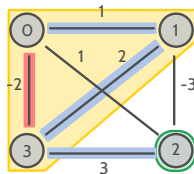
- **ub_inc**: Upper bound increment of move (v, c)
- Initialise **ub_inc** to 0
- For every assigned vertex v' (in **clique** list)
 - If v' is in clique c and the edge (v', v) has negative weight w
 - Then, add w to **ub_inc**
 - If v' is not in clique c and the edge (v', v) has positive weight w
 - Then, subtract w from **ub_inc**
- Example: Move $(3, 0)$



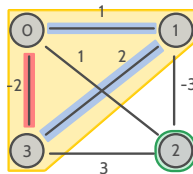
ub: 6



ub_inc: 0



ub_inc: -2



ub_inc: -5

Problem modelling: Move class



Problem

```
empty_solution(Problem) : Solution  
construction_neighbourhood(Problem) : Neighbourhood
```

Solution

```
objective_value(Solution) : double[0..1]  
lower_bound(Solution) : double[0..1]
```

Neighbourhood

```
moves(Neighbourhood, Solution) : Move[0..*]
```

Move

```
apply_move(Move, Solution) : Solution  
lower_bound_increment(Move, Solution) : double[0..1]
```

- Implement API functions related to (applying) moves

Concluding Remarks



- Community Detection Problem (better known as the Clique Partitioning Problem)
- API functions required by a simple greedy algorithm
- Simple example of how to model this problem

Concluding Remarks



- Community Detection Problem (better known as the Clique Partitioning Problem)
- API functions required by a simple greedy algorithm
- Simple example of how to model this problem

Thank you!

This presentation is based upon work from COST Action Randomised Optimisation Algorithms Research Network (ROAR-NET), CA22137, supported by COST (European Cooperation in Science and Technology).

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project: 2022.08367.CEECIND/CP1717/CT0001