

# The ROAR-NET API: Local Search

ROAR-NET Problem Modelling Code Fest

Andreia P. Guerreiro  
INESC-ID, Portugal

# Local Search: Best-improvement algorithm



- Best improvement local search:

# Local Search: Best-improvement algorithm



- Best improvement local search:
  - Start with a random solution  $\mathbf{s}$

# Local Search: Best-improvement algorithm



- Best improvement local search:
  - Start with a random solution  $\mathbf{s}$
  - While there is an improving local move

# Local Search: Best-improvement algorithm



- Best improvement local search:
  - Start with a random solution  $\mathbf{s}$
  - While there is an improving local move
    - Select the local move that improves the objective value the most

# Local Search: Best-improvement algorithm



- Best improvement local search:
  - Start with a random solution  $\mathbf{s}$
  - While there is an improving local move
    - Select the local move that improves the objective value the most
    - Apply the move to (update) solution  $\mathbf{s}$

# Local Search: Best-improvement algorithm

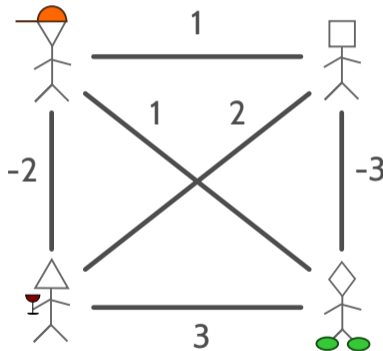


- Best improvement local search:
  - Start with a random solution  $\mathbf{s}$
  - While there is an improving local move
    - Select the local move that improves the objective value the most
    - Apply the move to (update) solution  $\mathbf{s}$
  - Return  $\mathbf{s}$

# Community Detection Problem

ROAR  
NET

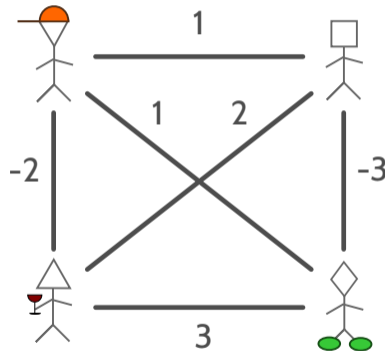
- A fully-connected undirected graph



# Community Detection Problem

ROAR  
NET

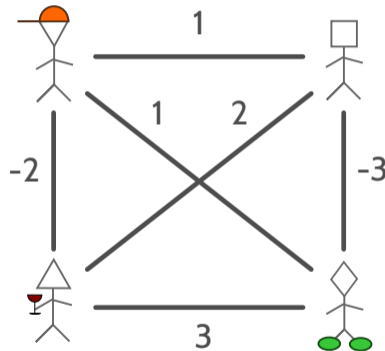
- A fully-connected undirected graph
  - Vertices represent users



# Community Detection Problem

ROAR  
NET

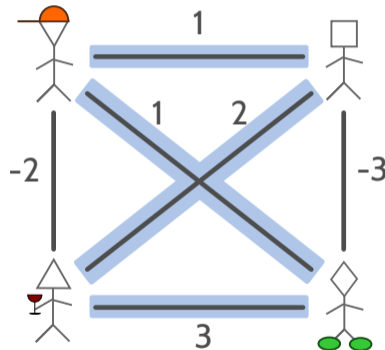
- A fully-connected undirected graph
  - Vertices represent users
  - Weighted edges represent the intensity of some attribute of their interaction



# Community Detection Problem

ROAR  
NET

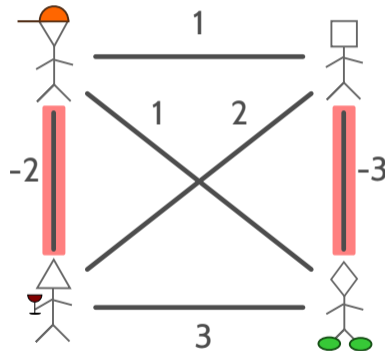
- A fully-connected undirected graph
  - Vertices represent users
  - Weighted edges represent the intensity of some attribute of their interaction
- Positive weight show affinity between users



# Community Detection Problem

ROAR  
NET

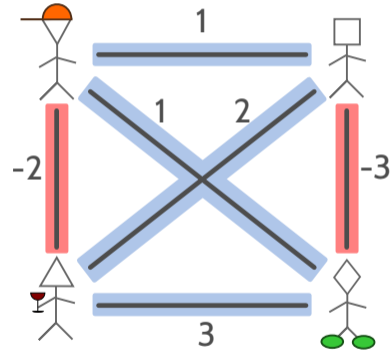
- A fully-connected undirected graph
  - Vertices represent users
  - Weighted edges represent the intensity of some attribute of their interaction
- Positive weight show affinity between users
- Negative edge weights indicate lack of affinity



# Community Detection Problem

FOAR  
NET

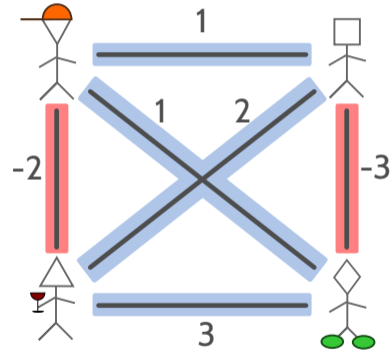
- A fully-connected undirected graph
  - Vertices represent users
  - Weighted edges represent the intensity of some attribute of their interaction
- Positive weight show affinity between users
- Negative edge weights indicate lack of affinity
- Groups of users connected mostly by positively weighted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



# Community Detection Problem

ROAR  
NET

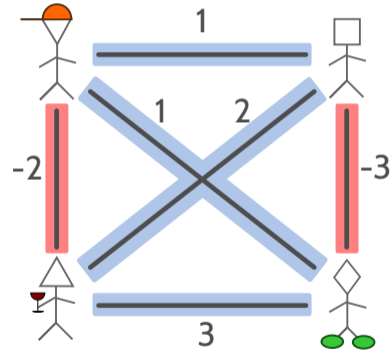
- A fully-connected undirected graph
  - Vertices represent users
  - Weighted edges represent the intensity of some attribute of their interaction
- Positive weight show affinity between users
- Negative edge weights indicate lack of affinity
- Groups of users connected mostly by positively weighted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



# 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



## Problem

```
random_solution(Problem) : Solution  
local_neighbourhood(Problem) : Neighbourhood
```

## Solution

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

## Neighbourhood

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

## Move

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

# Problem modelling: Local neighbourhood



- What is a neighbour?

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$
  - **nodes**: A dictionary;  $nodes[c]$  is the set of vertices in clique  $c$

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$
  - **nodes**: A dictionary;  $nodes[c]$  is the set of vertices in clique  $c$
  - **unused**: List of clique indices not used

# Problem modelling: Local neighbourhood



- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$
  - **nodes**: A dictionary;  $nodes[c]$  is the set of vertices in clique  $c$
  - **unused**: List of clique indices not used
  - **k**: Number of cliques

# Problem modelling: Local neighbourhood

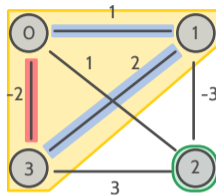


- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$
  - **nodes**: A dictionary;  $nodes[c]$  is the set of vertices in clique  $c$
  - **unused**: List of clique indices not used
  - **k**: Number of cliques
  - **ov**: Objective value

# Problem modelling: Local neighbourhood

ROAR  
NET

- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$
  - **nodes**: A dictionary;  $nodes[c]$  is the set of vertices in clique  $c$
  - **unused**: List of clique indices not used
  - **k**: Number of cliques
  - **ov**: Objective value



clique: [0,0,1,0]

nodes:

0: {0,1,3}

1: {2}

unused: [3,2]

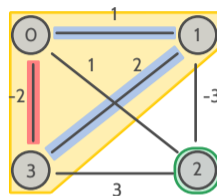
k: 2

ov: 1

# Problem modelling: Local neighbourhood

FOAR  
NET

- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$
  - **nodes**: A dictionary;  $nodes[c]$  is the set of vertices in clique  $c$
  - **unused**: List of clique indices not used
  - **k**: Number of cliques
  - **ov**: Objective value



clique: [0,0,1,0]

nodes:

0: {0,1,3}

1: {2}

unused: [3,2]

k: 2

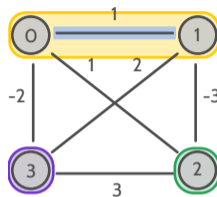
ov: 1

Apply move (3,2)

# Problem modelling: Local neighbourhood

FOAR  
NET

- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$
  - **nodes**: A dictionary;  $nodes[c]$  is the set of vertices in clique  $c$
  - **unused**: List of clique indices not used
  - **k**: Number of cliques
  - **ov**: Objective value



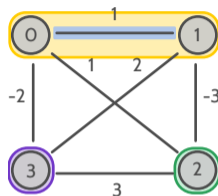
```
clique: [0,0,1,2]
nodes:
  0: {0,1}
  1: {2}
  2: {3}
unused: [3]
k: 3
ov: 1
```

Applied move (3,2)

# Problem modelling: Local neighbourhood

FOAR  
NET

- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$
  - **nodes**: A dictionary;  $nodes[c]$  is the set of vertices in clique  $c$
  - **unused**: List of clique indices not used
  - **k**: Number of cliques
  - **ov**: Objective value

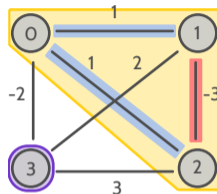


```
clique: [0,0,1,2]
nodes:
  0: {0,1}
  1: {2}
  2: {3}
unused: [3]
k: 3
ov: 1
```

Apply move (2,0)

# Problem modelling: Local neighbourhood

- What is a neighbour?
  - A solution that differs in the clique index of a single vertex
- How to generate a move?
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries
  - Neighbourhood size:  $O(k)$
- Move:  $(v, c)$ 
  - $v$ : vertex index
  - $c$ : clique index
- Objective value
  - Current total clique value
- Solution representation
  - **clique**: A list;  $clique[i]$  is the clique index of vertex  $i$
  - **nodes**: A dictionary;  $nodes[c]$  is the set of vertices in clique  $c$
  - **unused**: List of clique indices not used
  - **k**: Number of cliques
  - **ov**: Objective value



```
clique: [0,0,0,2]
nodes:
  0: {0,1,2}

  2: {3}
unused: [3,1]
k: 2
ov: -1
```

Applied move (2,0)

# Problem modelling: Solution class



## Problem

```
random_solution(Problem) : Solution  
local_neighbourhood(Problem) : Neighbourhood
```

## Solution

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

## Neighbourhood

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

## Move

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

- Update data structures in **Solution** class
- Implement and/or update API functions to create and evaluate solutions

# Problem modelling: Neighbourhood class



## Problem

```
random_solution(Problem) : Solution  
local_neighbourhood(Problem) : Neighbourhood
```

## Solution

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

## Neighbourhood

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

## Move

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

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

# Problem modelling: Neighbourhood class



- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one

# Problem modelling: Neighbourhood class

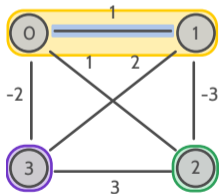


- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries if the move:

# Problem modelling: Neighbourhood class

- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries if the move:
    - consists in merging two cliques of size 1

Solution  $s$



nodes:

0: {0,1}

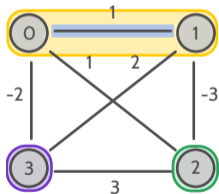
1: {2}

2: {3}

# Problem modelling: Neighbourhood class

- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries if the move:
    - consists in merging two cliques of size 1

Solution  $s$



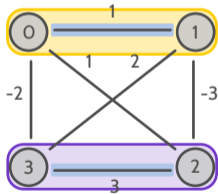
nodes:

0: {0,1}

1: {2}

2: {3}

apply `move(2,2)` to  $s$



nodes:

0: {0,1}

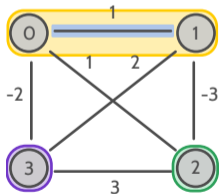
2: {3,2}

# Problem modelling: Neighbourhood class

ROAR  
NET

- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries if the move:
    - consists in merging two cliques of size 1

Solution  $s$



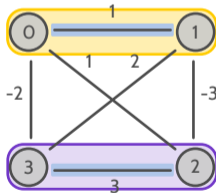
nodes:

0: {0,1}

1: {2}

2: {3}

apply `move(2,2)` to  $s$

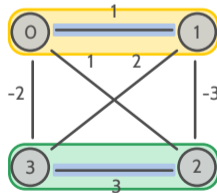


nodes:

0: {0,1}

2: {3,2}

apply `move(3,1)` to  $s$



nodes:

0: {0,1}

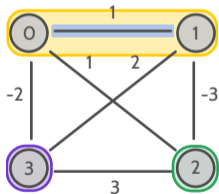
1: {2,3}

# Problem modelling: Neighbourhood class

ROAR  
NET

- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries if the move:
    - consists in merging two cliques of size 1

Solution  $s$



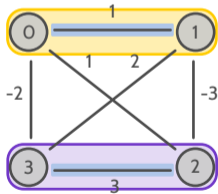
nodes:

0: {0,1}

1: {2}

2: {3}

apply `move(2,2)` to  $s$  ❌

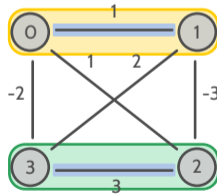


nodes:

0: {0,1}

2: {3,2}

apply `move(3,1)` to  $s$  ✅



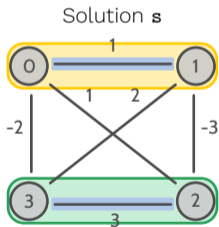
nodes:

0: {0,1}

1: {2,3}

# Problem modelling: Neighbourhood class

- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries if the move:
    - consists in merging two cliques of size 1
    - consists in splitting a clique of size 2



nodes:

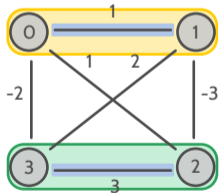
0: {0,1}

1: {2,3}

# Problem modelling: Neighbourhood class

- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries if the move:
    - consists in merging two cliques of size 1
    - consists in splitting a clique of size 2

Solution  $s$

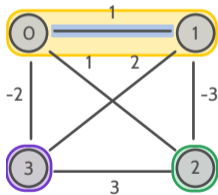


nodes:

0: {0,1}

1: {2,3}

apply `move(3,2)` to  $s$



nodes:

0: {0,1}

1: {2}

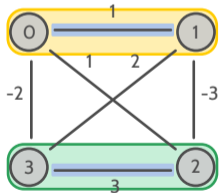
2: {3}

# Problem modelling: Neighbourhood class

ROAR  
NET

- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries if the move:
    - consists in merging two cliques of size 1
    - consists in splitting a clique of size 2

Solution  $s$

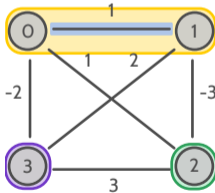


nodes:

0: {0,1}

1: {2,3}

apply `move(3,2)` to  $s$



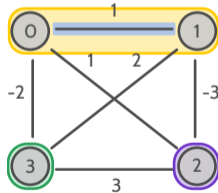
nodes:

0: {0,1}

1: {2}

2: {3}

apply `move(2,2)` to  $s$



nodes:

0: {0,1}

1: {3}

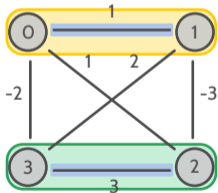
2: {2}

# Problem modelling: Neighbourhood class

ROAR  
NET

- Generate all moves
  - Select a vertex
  - Either assign to one of the other  $k$  existing cliques or to a newly created one
  - Avoid symmetries if the move:
    - consists in merging two cliques of size 1
    - consists in splitting a clique of size 2

Solution  $s$

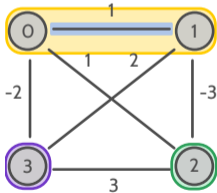


nodes:

0: {0,1}

1: {2,3}

apply `move(3,2)` to  $s$  ✓



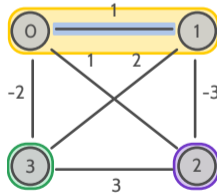
nodes:

0: {0,1}

1: {2}

2: {3}

apply `move(2,2)` to  $s$  ✗



nodes:

0: {0,1}

1: {3}

2: {2}

# Problem modelling: Move class



## Problem

```
random_solution(Problem) : Solution  
local_neighbourhood(Problem) : Neighbourhood
```

## Solution

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

## Neighbourhood

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

## Move

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

- Implement API functions related to (applying) moves

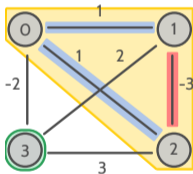
# Problem modelling: objective value increment



- `ov_inc`: Objective value increment of move  $(v, c)$

# Problem modelling: objective value increment

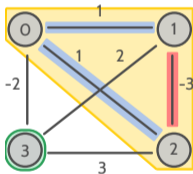
- `ov_inc`: Objective value increment of move  $(v, c)$
- Example: Move  $(2, 1)$



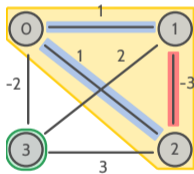
ov: -1

# Problem modelling: objective value increment

- `ov_inc`: Objective value increment of move  $(v, c)$
- Initialise `ov_inc` to 0
- Example: Move  $(2, 1)$



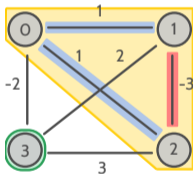
`ov`: -1



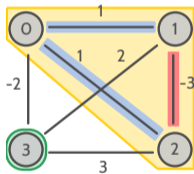
`ov_inc`: 0

# Problem modelling: objective value increment

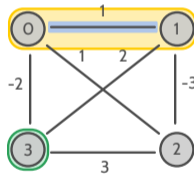
- **ov\_inc**: Objective value increment of move  $(v, c)$
- Initialise **ov\_inc** to 0
- Example: Move  $(2, 1)$



ov: -1



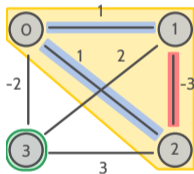
ov\_inc: 0



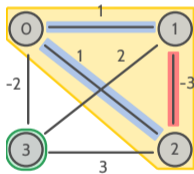
ub\_inc: 2

# Problem modelling: objective value increment

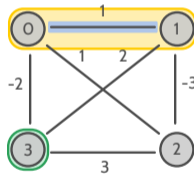
- `ov_inc`: Objective value increment of move  $(v, c)$
- Initialise `ov_inc` to 0
- Example: Move  $(2, 1)$



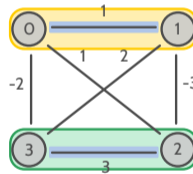
ov: -1



ov\_inc: 0



ub\_inc: 2



ub\_inc: 5

# Problem modelling: Move class



## Problem

```
random_solution(Problem) : Solution  
local_neighbourhood(Problem) : Neighbourhood
```

## Solution

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

## Neighbourhood

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

## Move

```
apply_move(Move, Solution) : Solution  
objective_value_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 local search 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 local search 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