Master Thesis:
New Algorithm for Combinatorial Hypergraph Partitioning

Milana Gatarić

University of Novi Sad, Department of Mathematics and Informatics

2010
Motivation

Maintaining of large software system

- the system needs to be split into smaller, more manageable modules
- the communication among them is minimized
Maintaining of large software system

- the system needs to be split into smaller, more manageable modules
- the communication among them is minimized
Motivation

Maintaining of large software system

- the system needs to be split into smaller, more manageable modules
- the communication among them is minimized
Motivation

Maintaining of large software system

- the system needs to be split into smaller, more manageable modules
- the communication among them is minimized
Software system can be represented by a directed graph:

- vertices represent programs, classes, or similar program units of software system
- arcs represent communication among them: an arc \((v, u)\), i.e. \(v \rightarrow u\), means that program \(v\) calls program \(u\)
- vertex \(v\) may have a weight \(\omega(v)\), such as the weight of maintaining the corresponding program unit
Software system can be represented by a directed graph:

- Vertices represent programs, classes, or similar program units of software system.
- Arcs represent communication among them: an arc \((v, u)\), i.e. \(v \rightarrow u\), means that program \(v\) calls program \(u\).
- Vertex \(v\) may have a weight \(\omega(v)\), such as the weight of maintaining the corresponding program unit.
Software system can be represented by a directed graph:

- Vertices represent programs, classes, or similar program units of software system.
- Arcs represent communication among them: an arc \((v, u)\), i.e. \(v \rightarrow u\), means that program \(v\) calls program \(u\).
- Vertex \(v\) may have a weight \(\omega(v)\), such as the weight of maintaining the corresponding program unit.
Software system can be represented by a directed graph:

- vertices represent programs, classes, or similar program units of software system
- arcs represent communication among them: an arc \((v, u)\), i.e. \(v \rightarrow u\), means that program \(v\) calls program \(u\)
- vertex \(v\) may have a weight \(\omega(v)\), such as the weight of maintaining the corresponding program unit
Software system can be represented by a directed graph:

- vertices represent programs, classes, or similar program units of software system
- arcs represent communication among them: an arc \((v, u)\), i.e. \(v \rightarrow u\), means that program \(v\) calls program \(u\)
- vertex \(v\) may have a weight \(\omega(v)\), such as the weight of maintaining the corresponding program unit
Software system can be represented by a directed graph:

- vertices represent programs, classes, or similar program units of software system
- arcs represent communication among them: an arc \((v, u)\), i.e. \(v \rightarrow u\), means that program \(v\) calls program \(u\)
- vertex \(v\) may have a weight \(\omega(v)\), such as the weight of maintaining the corresponding program unit
Introduction

Call graph partitioning problem

Directed graph (*call graph*) partitioning

- **Given:** $D(V, E)$, weight function $\omega: V \rightarrow \mathbb{N}$, number of modules $L \in \mathbb{N}$ and the capacity of each $K \in \mathbb{N}$

- **Find:** a partition $\{V_1, \ldots, V_L\}$ of $V$, so that $\sum_{v \in V_l} \omega(v) \leq K$ for all $l$ and so that the number of interfaces is minimized.
Directed graph (*call graph*) partitioning

- **Given:** \( D(V, E) \), weight function \( \omega : V \to \mathbb{N} \), number of modules \( L \in \mathbb{N} \) and the capacity of each \( K \in \mathbb{N} \)

- **Find:** a partition \( \{ V_1, ..., V_L \} \) of \( V \), so that \( \sum_{v \in V_i} \omega(v) \leq K \) for all \( i \) and so that the number of interfaces is minimized.
Directed graph (*call graph*) partitioning

**Given:** $D(V, E)$, weight function $\omega : V \rightarrow \mathbb{N}$, number of modules $L \in \mathbb{N}$ and the capacity of each $K \in \mathbb{N}$

**Find:** a partition $\{V_1, \ldots, V_L\}$ of $V$, so that $\sum_{v \in V_l} \omega(v) \leq K$ for all $l$ and so that the number of interfaces is minimized.
Directed graph *(call graph)* partitioning

- **Given:** $D(V, E)$, weight function $\omega : V \rightarrow \mathbb{N}$, number of modules $L \in \mathbb{N}$ and the capacity of each $K \in \mathbb{N}$
- **Find:** a partition $\{V_1, \ldots, V_L\}$ of $V$, so that $\sum_{v \in V_i} \omega(v) \leq K$ for all $i$ and so that the number of interfaces is minimized.
Directed graph (call graph) partitioning

- **Given:** $D(V, E)$, weight function $\omega : V \rightarrow \mathbb{N}$, number of modules $L \in \mathbb{N}$ and the capacity of each $K \in \mathbb{N}$

- **Find:** a partition $\{V_1, ..., V_L\}$ of $V$, so that $\sum_{v \in V_l} \omega(v) \leq K$ for all $l$ and so that the number of interfaces is minimized.
Directed graph (*call graph*) partitioning

- **Given:** \( D(V, E) \), weight function \( \omega : V \rightarrow \mathbb{N} \), number of modules \( L \in \mathbb{N} \) and the capacity of each \( K \in \mathbb{N} \)
- **Find:** a partition \( \{ V_1, ..., V_L \} \) of \( V \), so that \( \sum_{v \in V_l} \omega(v) \leq K \) for all \( l \) and so that the number of interfaces is minimized.
Directed graph (call graph) partitioning

- **Given:** $D(V, E)$, weight function $\omega : V \rightarrow \mathbb{N}$, number of modules $L \in \mathbb{N}$ and the capacity of each $K \in \mathbb{N}$

- **Find:** a partition $\{V_1, ..., V_L\}$ of $V$, so that $\sum_{v \in V_l} \omega(v) \leq K$ for all $l$ and so that the number of interfaces is minimized.
Directed graph (*call graph*) partitioning

- **Given:** $D(V, E)$, weight function $\omega : V \to \mathbb{N}$, number of modules $L \in \mathbb{N}$ and the capacity of each $K \in \mathbb{N}$

- **Find:** a partition $\{V_1, ..., V_L\}$ of $V$, so that $\sum_{v \in V_i} \omega(v) \leq K$ for all $i$ and so that the number of interfaces is minimized.
Call graph partitioning problem
Call graph partitioning problem
Call graph partitioning problem
partitioning problem of a call graph with minimization of interfaces

⇔

hypergraph partitioning problem with minimization of broken hyperedges
partitioning problem of a call graph with minimization of interfaces

⇔

hypergraph partitioning problem with minimization of broken hyperedges
Definition of a hypergraph:

A hypergraph $H(V, \mathcal{N})$ consists of a nonempty finite set of vertices $V$ and a set $\mathcal{N} \subseteq \mathcal{P}(V)$ of hyperedges, or nets, which are subsets of $V$ of arbitrary cardinality.

Hypergraph partitioning problem:

The aim: partition the set $V$ into pairwise disjoint nonempty subsets $\{V_1, \ldots, V_L\}$, so that weight of each subset is limited and the number of broken nets is minimized.
Definition of a hypergraph:

A hypergraph $H(V, \mathcal{N})$ consists of a nonempty finite set of vertices $V$ and a set $\mathcal{N} \subseteq \mathcal{P}(V)$ of hyperedges, or nets, which are subsets of $V$ of arbitrary cardinality.

Hypergraph partitioning problem:

The aim: partition the set $V$ into pairwise disjoint nonempty subsets $\{V_1, \ldots, V_L\}$, so that weight of each subset is limited and the number of broken nets is minimized.
Definition of a hypergraph:

A hypergraph $H(V, \mathcal{N})$ consists of a nonempty finite set of vertices $V$ and a set $\mathcal{N} \subseteq \mathcal{P}(V)$ of hyperedges, or nets, which are subsets of $V$ of arbitrary cardinality.

Hypergraph partitioning problem:

The aim: partition the set $V$ into pairwise disjoint nonempty subsets $\{V_1, \ldots, V_L\}$, so that weight of each subset is limited and the number of broken nets is minimized.
Definition of a hypergraph:

A hypergraph $H(V, \mathcal{N})$ consists of a nonempty finite set of vertices $V$ and a set $\mathcal{N} \subseteq \mathcal{P}(V)$ of hyperedges, or nets, which are subsets of $V$ of arbitrary cardinality.

Hypergraph partitioning problem:

The aim: partition the set $V$ into pairwise disjoint nonempty subsets $\{V_1, \ldots, V_L\}$, so that weight of each subset is limited and the number of broken nets is minimized.
Definition of a hypergraph:

A hypergraph $H(V, \mathcal{N})$ consists of a nonempty finite set of vertices $V$ and a set $\mathcal{N} \subseteq \mathcal{P}(V)$ of hyperedges, or nets, which are subsets of $V$ of arbitrary cardinality.

Hypergraph partitioning problem:

The aim: partition the set $V$ into pairwise disjoint nonempty subsets $\{V_1, \ldots, V_L\}$, so that weight of each subset is limited and the number of broken nets is minimized.
Definition of a hypergraph:

A hypergraph $H(V, \mathcal{N})$ consists of a nonempty finite set of vertices $V$ and a set $\mathcal{N} \subseteq \mathcal{P}(V)$ of hyperedges, or nets, which are subsets of $V$ of arbitrary cardinality.

Hypergraph partitioning problem:

The aim: partition the set $V$ into pairwise disjoint nonempty subsets $\{V_1, \ldots, V_L\}$, so that weight of each subset is limited and the number of broken nets is minimized.
Definition of a hypergraph:

A hypergraph $H(V, \mathcal{N})$ consists of a nonempty finite set of vertices $V$ and a set $\mathcal{N} \subseteq \mathcal{P}(V)$ of hyperedges, or nets, which are subsets of $V$ of arbitrary cardinality.

Hypergraph partitioning problem:

The aim: partition the set $V$ into pairwise disjoint nonempty subsets $\{V_1, \ldots, V_L\}$, so that weight of each subset is limited and the number of broken nets is minimized.
Hypergraph partitioning problem
Hypergraph partitioning problem
Introduction

New algorithm

Hypergraph partitioning problem
partitioning problem of a call graph with minimization of interfaces

⇔

hypergraph partitioning problem with minimization of broken hyperedges

NP-hard!
partitioning problem of a call graph with minimization of interfaces

⇔

hypergraph partitioning problem with minimization of broken hyperedges

NP-hard!
partitioning problem of a call graph with minimization of interfaces

⇔

hypergraph partitioning problem with minimization of broken hyperedges

NP-hard!
New algorithm

Input:
- Directed graph $D(V, E)$, $\omega: v \mapsto \omega(v)$
- Module capacity:

$$K = \left\lfloor \frac{\nu \omega(D)}{L} \right\rfloor,$$

where:
$L$ is a given number of modules,
$\omega(D) = \sum_{v \in V} \omega(v)$ total graph weight,
$\nu$ is a tolerance parameter of exceeding module capacity (1.1-1.2)

We solve the problem of partitioning the call graph $D$ with minimizing the number of interfaces!
New algorithm

Input:

- Directed graph $D(V, E)$, $\omega : v \mapsto \omega(v)$
- Module capacity:

\[ K = \left\lfloor \frac{\nu \omega(D)}{L} \right\rfloor, \]

where:
- $L$ is a given number of modules,
- $\omega(D) = \sum_{v \in V} \omega(v)$ total graph weight,
- $\nu$ is a tolerance parameter of exceeding module capacity (1.1-1.2)

We solve the problem of partitioning the call graph $D$ with minimizing the number of interfaces!
New algorithm

Input:
- Directed graph $D(V, E)$, $\omega : v \mapsto \omega(v)$
- Module capacity:

\[ K = \lfloor \frac{\nu \omega(D)}{L} \rfloor, \]

where:
- $L$ is a given number of modules,
- $\omega(D) = \sum_{v \in V} \omega(v)$ total graph weight,
- $\nu$ is a tolerance parameter of exceeding module capacity (1.1-1.2)

We solve the problem of partitioning the call graph $D$ with minimizing the number of interfaces!
Input:
- Directed graph $D(V, E)$, $\omega : v \mapsto \omega(v)$
- Module capacity:

$$K = \left\lfloor \frac{\nu \omega(D)}{L} \right\rfloor,$$

where:
- $L$ is a given number of modules,
- $\omega(D) = \sum_{v \in V} \omega(v)$ total graph weight,
- $\nu$ is a tolerance parameter of exceeding module capacity (1.1-1.2)

We solve the problem of partitioning the call graph $D$ with minimizing the number of interfaces!
New algorithm

Input:
- Directed graph $D(V, E)$, $\omega : v \mapsto \omega(v)$
- Module capacity:

$$K = \left\lfloor \frac{\nu \omega(D)}{L} \right\rfloor,$$

where:
- $L$ is a given number of modules,
- $\omega(D) = \sum_{v \in V} \omega(v)$ total graph weight,
- $\nu$ is a tolerance parameter of exceeding module capacity (1.1-1.2)

We solve the problem of partitioning the call graph $D$ with minimizing the number of interfaces!
New algorithm

Input:
- Directed graph $D(V, E)$, $\omega : v \mapsto \omega(v)$
- Module capacity:

$$K = \left\lfloor \frac{\nu \omega(D)}{L} \right\rfloor,$$

where:
- $L$ is a given number of modules,
- $\omega(D) = \sum_{v \in V} \omega(v)$ total graph weight,
- $\nu$ is a tolerance parameter of exceeding module capacity (1.1-1.2)

We solve the problem of partitioning the call graph $D$ with minimizing the number of interfaces!
New algorithm

Input:
- Directed graph $D(V, E)$, $\omega : v \mapsto \omega(v)$
- Module capacity:

$$K = \left\lfloor \frac{\nu \omega(D)}{L} \right\rfloor,$$

where:
- $L$ is a given number of modules,
- $\omega(D) = \sum_{v \in V} \omega(v)$ total graph weight,
- $\nu$ is a tolerance parameter of exceeding module capacity (1.1-1.2)

We solve the problem of partitioning the call graph $D$ with minimizing the number of interfaces!
New algorithm

Input:

- Directed graph $D(V, E), \omega : v \mapsto \omega(v)$
- Module capacity:

$$K = \left\lfloor \frac{\nu \omega(D)}{L} \right\rfloor,$$

where:
- $L$ is a given number of modules,
- $\omega(D) = \sum_{v \in V} \omega(v)$ total graph weight,
- $\nu$ is a tolerance parameter of exceeding module capacity (1.1-1.2)

We solve the problem of partitioning the call graph $D$ with minimizing the number of interfaces!
New algorithm

Input:
- Directed graph $D(V, E)$, $\omega : v \mapsto \omega(v)$
- Module capacity:

$$K = \left\lfloor \frac{\nu \omega(D)}{L} \right\rfloor,$$

where:
- $L$ is a given number of modules,
- $\omega(D) = \sum_{v \in V} \omega(v)$ total graph weight,
- $\nu$ is a tolerance parameter of exceeding module capacity (1.1-1.2)

We solve the problem of partitioning the call graph $D$ with minimizing the number of interfaces!
Basic idea:

- During search for interfaces, make graph smaller and smaller: do it by clustering and elimination of vertices which don’t have to be interfaces

- First, make a graph smaller
Basic idea:

- During search for interfaces, make graph smaller and smaller: do it by clustering and elimination of vertices which don’t have to be interfaces

- First, make a graph smaller
Basic idea:

- During search for interfaces, make graph smaller and smaller: do it by clustering and elimination of vertices which don’t have to be interfaces

- First, make a graph smaller
for each $v \in V$, if

$$\omega(v) + \sum_{u: (u,v) \in E} \omega(u) > K$$

make $v$ interface and delete all its incoming edges
Inevitable interfaces

- for each $v \in V$, if

$$\omega(v) + \sum_{u : (u,v) \in E} \omega(u) > K$$

make $v$ interface and delete all its incoming edges
Inevitable interfaces

- for each $v \in V$, if

$$\omega(v) + \sum_{u:(u,v)\in E} \omega(u) > K$$

make $v$ interface and delete all its incoming edges
for each $v \in V$, if

$$\omega(v) + \sum_{u : (u, v) \in E} \omega(u) > K$$

make $v$ interface and delete all its incoming edges
Remove "small" components

for each component of connectivity $D'$, if

$$\omega(D') \leq K - \left\lfloor \frac{\omega(D \setminus D')}{L} \right\rfloor$$

leave $D'$ aside and avoid interfaces in it
Remove "small" components

for each component of connectivity $D'$, if

$$
\omega(D') \leq K - \left\lfloor \frac{\omega(D \setminus D')}{L} \right\rfloor
$$

leave $D'$ aside and avoid interfaces in it
Remove "small" components

for each component of connectivity $D'$, if

$$\omega(D') \leq K - \left\lfloor \frac{\omega(D \setminus D')}{L} \right\rfloor$$

leave $D'$ aside and avoid interfaces in it
New algorithm

Making graph smaller

Remove "small" components

- for each component of connectivity $D'$, if

$$\omega(D') \leq K - \left\lfloor \frac{\omega(D \setminus D')}{L} \right\rfloor$$

leave $D'$ aside and avoid interfaces in it
Clustering of vertices

- Imagine that $v$ is an interface and temporarily delete all its incoming edges $u \rightarrow v$.

- Find the component $D_v$, $v \in D_v$, if
  \[
  \omega(D_v) \leq K - \left\lfloor \frac{\omega(D \setminus D_v)}{L} \right\rfloor,
  \]
  cluster all vertices from $D_v$ in $v$, $\omega(v) := \omega(D_v)$.

- For each $u \in V$, where $u \rightarrow v$ is temporarily deleted, if $D_u \neq D_v$ and
  \[
  \omega(D_u) \leq K - \left\lfloor \frac{\omega(D \setminus D_u)}{L} \right\rfloor,
  \]
  cluster all vertices from $D_u$ in $u$, $\omega(u) := \omega(D_u)$.
Clustering of vertices

- imagine that $v$ is an interface and temporarily delete all its incoming edges $u \rightarrow v$
- find the component $D_v$, $v \in D_v$, if
  \[ \omega(D_v) \leq K - \left\lfloor \frac{\omega(D \setminus D_v)}{L} \right\rfloor, \]
  cluster all vertices from $D_v$ in $v$, $\omega(v) := \omega(D_v)$
- for each $u \in V$, where $u \rightarrow v$ is temporarily deleted, if $D_u \neq D_v$ and
  \[ \omega(D_u) \leq K - \left\lfloor \frac{\omega(D \setminus D_u)}{L} \right\rfloor, \]
  cluster all vertices from $D_u$ in $u$, $\omega(u) := \omega(D_u)$
Clustering of vertices

- imagine that $v$ is an interface and temporarily delete all its incoming edges $u \rightarrow v$
- find the component $D_v$, $v \in D_v$, if
  \[
  \omega(D_v) \leq K - \left\lfloor \frac{\omega(D \setminus D_v)}{L} \right\rfloor,
  \]
  cluster all vertices from $D_v$ in $v$, $\omega(v) := \omega(D_v)$
- for each $u \in V$, where $u \rightarrow v$ is temporarily deleted, if $D_u \neq D_v$ and
  \[
  \omega(D_u) \leq K - \left\lfloor \frac{\omega(D \setminus D_u)}{L} \right\rfloor,
  \]
  cluster all vertices from $D_u$ in $u$, $\omega(u) := \omega(D_u)$
Clustering of vertices

- imagine that $v$ is an interface and temporarily delete all its incoming edges $u \to v$
- find the component $D_v$, $v \in D_v$, if

$$\omega(D_v) \leq K - \left\lfloor \frac{\omega(D \setminus D_v)}{L} \right\rfloor,$$

cluster all vertices from $D_v$ in $v$, $\omega(v) := \omega(D_v)$
- for each $u \in V$, where $u \to v$ is temporarily deleted, if $D_u \neq D_v$ and

$$\omega(D_u) \leq K - \left\lfloor \frac{\omega(D \setminus D_u)}{L} \right\rfloor,$$

cluster all vertices from $D_u$ in $u$, $\omega(u) := \omega(D_u)$
Clustering of vertices

- imagine that $v$ is an interface and temporarily delete all its incoming edges $u \rightarrow v$
- find the component $D_v$, $v \in D_v$, if
  \[ \omega(D_v) \leq K - \left\lfloor \frac{\omega(D \setminus D_v)}{L} \right\rfloor, \]
  cluster all vertices from $D_v$ in $v$, $\omega(v) := \omega(D_v)$
- for each $u \in V$, where $u \rightarrow v$ is temporarily deleted, if $D_u \neq D_v$ and
  \[ \omega(D_u) \leq K - \left\lfloor \frac{\omega(D \setminus D_u)}{L} \right\rfloor, \]
  cluster all vertices from $D_u$ in $u$, $\omega(u) := \omega(D_u)$
Clustering of vertices

- imagine that \( v \) is an interface and temporarily delete all its incoming edges \( u \rightarrow v \)
- find the component \( D_v, \ v \in D_v \), if

\[
\omega(D_v) \leq K - \left\lfloor \frac{\omega(D \setminus D_v)}{L} \right\rfloor,
\]

cluster all vertices from \( D_v \) in \( v \), \( \omega(v) := \omega(D_v) \)

- for each \( u \in V \), where \( u \rightarrow v \) is temporarily deleted, if \( D_u \neq D_v \) and

\[
\omega(D_u) \leq K - \left\lfloor \frac{\omega(D \setminus D_u)}{L} \right\rfloor,
\]

cluster all vertices from \( D_u \) in \( u \), \( \omega(u) := \omega(D_u) \)
Clustering of vertices

- imagine that \( v \) is an interface and temporarily delete all its incoming edges \( u \rightarrow v \)
- find the component \( D_v, v \in D_v \), if

\[
\omega(D_v) \leq K - \left\lfloor \frac{\omega(D \setminus D_v)}{L} \right\rfloor,
\]

cluster all vertices from \( D_v \) in \( v \), \( \omega(v) := \omega(D_v) \)
- for each \( u \in V \), where \( u \rightarrow v \) is temporarily deleted , if \( D_u \neq D_v \) and

\[
\omega(D_u) \leq K - \left\lfloor \frac{\omega(D \setminus D_u)}{L} \right\rfloor,
\]

cluster all vertices from \( D_u \) in \( u \), \( \omega(u) := \omega(D_u) \)
Clustering of vertices

- imagine that \( v \) is an interface and temporarily delete all its incoming edges \( u \rightarrow v \)
- find the component \( D_v, v \in D_v \), if
  \[
  \omega(D_v) \leq K - \left\lfloor \frac{\omega(D \setminus D_v)}{L} \right\rfloor,
  \]
  cluster all vertices from \( D_v \) in \( v \), \( \omega(v) := \omega(D_v) \)
- for each \( u \in V \), where \( u \rightarrow v \) is temporarily deleted, if \( D_u \neq D_v \) and
  \[
  \omega(D_u) \leq K - \left\lfloor \frac{\omega(D \setminus D_u)}{L} \right\rfloor,
  \]
  cluster all vertices from \( D_u \) in \( u \), \( \omega(u) := \omega(D_u) \)
Making graph smaller - numerical results

| example   | |V| | |E| | L | |ν| | reduction | |V| | % |
|-----------|----|----|---|---|---|---|----|---|---|-----------|---|---|
| graph 1   | 15 | 39 | 8 | 1.2 | 47 |
| graph 2   | 449 | 659 | 8 | 1.2 | 66 |
| graph 3   | 947 | 1900 | 8 | 1.2 | 42 |
| graph 4   | 1100 | 2951 | 8 | 1.2 | 24 |
| graph 5   | 1145 | 2686 | 8 | 1.2 | 20 |
| graph 6   | 2142 | 2436 | 8 | 1.2 | 59 |
First solution (not optimal)

Find inevitable interfaces;

\[ T := \text{FindLargestComponentWeight}(); \]

**while** \((T > K)\)

Remove "small" components;
Cluster vertices;
\[ v = \text{FindVertexWithMaxIncoming}(); \]
Make \(v\) interface;
\[ T := \text{FindLargestComponentWeight}(); \]

**end while;**
Find inevitable interfaces;

\[ T := \text{FindLargestComponentWeight}(); \]

while \( T > K \)
  
  Remove "small" components;
  Cluster vertices;
  \( v = \text{FindVertexWithMaxIncoming}(); \)
  Make \( v \) interface;
  \( T := \text{FindLargestComponentWeight}(); \)

end while;
First solution (not optimal)

Find inevitable interfaces;

\[ T := \text{FindLargestComponentWeight();} \]

\text{while } (T > K) \text{ do}

Remove "small" components;
Cluster vertices;
\( v = \text{FindVertexWithMaxIncoming();} \)
Make \( v \) interface;
\( T := \text{FindLargestComponentWeight();} \)

end while;
Find inevitable interfaces;
\[ T := \text{FindLargestComponentWeight}(); \]

while \( T > K \)

- Remove "small" components;
- Cluster vertices;
- \( v = \text{FindVertexWithMaxIncoming}(); \)
- Make \( v \) interface;
- \( T := \text{FindLargestComponentWeight}(); \)

end while;
Optimal solution

- Use previous solution as an initial
- After making graph smaller, sort the rest of vertices by the number of incoming edges
- Search sorted sequence of vertices for the right combination of interfaces which will give us the better solution than the temporary solution is
- Stop when we can not get better solution anymore (we have the optimal one)
Solving the problem

Optimal solution

- Use previous solution as an initial
- After making graph smaller, sort the rest of vertices by the number of incoming edges
- Search sorted sequence of vertices for the right combination of interfaces which will give us the better solution than the temporary solution is
- Stop when we can not get better solution anymore (we have the optimal one)
Optimal solution

- Use previous solution as an initial
- After making graph smaller, sort the rest of vertices by the number of incoming edges
- Search sorted sequence of vertices for the right combination of interfaces which will give us the better solution than the temporary solution is
- Stop when we can not get better solution anymore (we have the optimal one)
Introduction

New algorithm

Solving the problem

Optimal solution

- Use previous solution as an initial
- After making graph smaller, sort the rest of vertices by the number of incoming edges
- Search sorted sequence of vertices for the right combination of interfaces which will give us the better solution than the temporary solution is
- Stop when we can not get better solution anymore (we have the optimal one)
**Optimal solution**

- Use previous solution as an initial
- After making graph smaller, sort the rest of vertices by the number of incoming edges
- Search sorted sequence of vertices for the right combination of interfaces which will give us the better solution than the temporary solution is
- Stop when we can not get better solution anymore (we have the optimal one)
Thank you for your attention.