Simple Undirected Two-Commodity Integral Flow with a Unitary Demand

Alexsander A. Melo[1] Celina M. H. Figueiredo[1] Uéverton S. Souza[2]

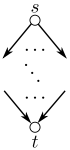
[1]Federal University of Rio de Janeiro, Brazil
[2]Federal Fluminense University, Brazil

LAGOS 2017

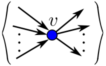
September 15, 2017

Introduction

- The classical single-commodity flow problem is well-known as MAXIMUM FLOW

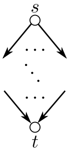


  - The goal is to send the maximum possible flow from the *source* s into the *sink* t;
  - The edge capacity and flow conservation constraints must be satisfied.

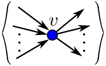


Introduction

- The classical single-commodity flow problem is well-known as MAXIMUM FLOW

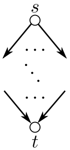


  - The goal is to send the maximum possible flow from the *source* s into the *sink* t;
  - The edge capacity and flow conservation constraints must be satisfied.

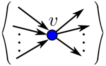


- A natural generalisation of MAXIMUM FLOW is the MULTICOMMODITY FLOW problem

- The classical single-commodity flow problem is well-known as MAXIMUM FLOW

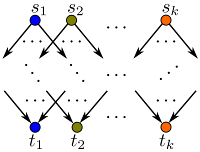


  - The goal is to send the maximum possible flow from the *source* s into the *sink* t;
  - The edge capacity and flow conservation constraints must be satisfied.



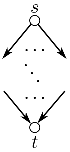
- A natural generalisation of MAXIMUM FLOW is the MULTICOMMODITY FLOW problem

  - Multiple commodities: $\{s_i, t_i\}$;

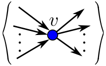


Introduction

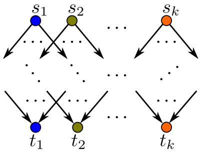
- The classical single-commodity flow problem is well-known as MAXIMUM FLOW



  - The goal is to send the maximum possible flow from the *source* s into the *sink* t;
  - The edge capacity and flow conservation constraints must be satisfied.

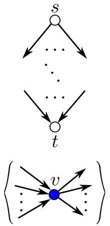


- A natural generalisation of MAXIMUM FLOW is the MULTICOMMODITY FLOW problem
  - Multiple commodities: $\{s_i, t_i\}$;
  - Each commodity has a different flow demand;



Introduction

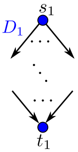
- The classical single-commodity flow problem is well-known as MAXIMUM FLOW



  - The goal is to send the maximum possible flow from the *source* s into the *sink* t;
  - The edge capacity and flow conservation constraints must be satisfied.

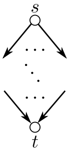
- A natural generalisation of MAXIMUM FLOW is the MULTICOMMODITY FLOW problem

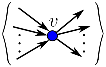
  - Multiple commodities: $\{s_i, t_i\}$;
  - Each commodity has a different flow demand;



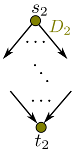
Introduction

- The classical single-commodity flow problem is well-known as MAXIMUM FLOW
  - The goal is to send the maximum possible flow from the *source* s into the *sink* t;
  - The edge capacity and flow conservation constraints must be satisfied.



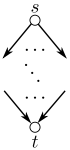


- A natural generalisation of MAXIMUM FLOW is the MULTICOMMODITY FLOW problem
  - Multiple commodities: $\{s_i, t_i\}$;
  - Each commodity has a different flow demand;

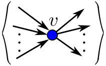


Introduction

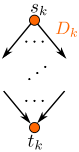
- The classical single-commodity flow problem is well-known as MAXIMUM FLOW



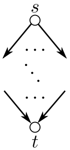
  - The goal is to send the maximum possible flow from the *source* s into the *sink* t;
  - The edge capacity and flow conservation constraints must be satisfied.



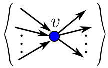
- A natural generalisation of MAXIMUM FLOW is the MULTICOMMODITY FLOW problem
  - Multiple commodities: $\{s_i, t_i\}$;
  - Each commodity has a different flow demand;



- The classical single-commodity flow problem is well-known as MAXIMUM FLOW

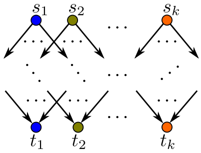


  - The goal is to send the maximum possible flow from the *source* s into the *sink* t;
  - The edge capacity and flow conservation constraints must be satisfied.



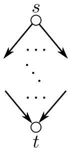
- A natural generalisation of MAXIMUM FLOW is the MULTICOMMODITY FLOW problem

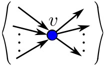
  - Multiple commodities: $\{s_i, t_i\}$;
  - Each commodity has a different flow demand;
  - The capacities of the edges are shared between the flows of each commodity.



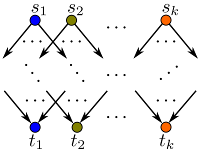
Introduction

- The classical single-commodity flow problem is well-known as MAXIMUM FLOW
  - The goal is to send the maximum possible flow from the *source* s into the *sink* t;
  - The edge capacity and flow conservation constraints must be satisfied.





- A natural generalisation of MAXIMUM FLOW is the MULTICOMMODITY FLOW problem
  - Multiple commodities: $\{s_i, t_i\}$;
  - Each commodity has a different flow demand;
  - The capacities of the edges are shared between the flows of each commodity.
  - Again, the edge capacity and flow conservation constraints must be satisfied.



Introduction: complexity

- The MAXIMUM FLOW problem is a classical polynomial-time solvable problem
  - Even if the flow must be integral, *i.e.* an integer-valued function

Introduction: complexity

- The MAXIMUM FLOW problem is a classical polynomial-time solvable problem
  - Even if the flow must be integral, *i.e.* an integer-valued function
- By using linear programming, the MULTICOMMODITY FLOW problem can be solved in polynomial-time if the flows are **real-valued** functions

Introduction: complexity

- The MAXIMUM FLOW problem is a classical polynomial-time solvable problem
  - Even if the flow must be integral, *i.e.* an integer-valued function
- By using linear programming, the MULTICOMMODITY FLOW problem can be solved in polynomial-time if the flows are **real-valued** functions
- On the other hand, Karp (1975) proved that MULTICOMMODITY FLOW is a NP-complete problem if the flows must be **integral**.

Introduction: simple two-commodity integral instances

- An instance of a network flow problem is called ***simple*** if the capacity of all edges of the input graph are **unitary**.

Introduction: simple two-commodity integral instances

- An instance of a network flow problem is called ***simple*** if the capacity of all edges of the input graph are **unitary**.
- A particular case of MULTICOMMODITY INTEGRAL FLOW is the SIMPLE TWO-COMMODITY INTEGRAL FLOW problem
  - Simple instances
  - Only two-commodities
  - Integral flows.

Introduction: SIMPLE TWO-COMMODITY INTEGRAL FLOW complexity

- In 1976, Even, Itai and Shamir proved the NP-completeness of SIMPLE TWO-COMMODITY INTEGRAL FLOW
  - Both in the directed and undirected cases.

Introduction: SIMPLE TWO-COMMODITY INTEGRAL FLOW complexity

- In 1976, Even, Itai and Shamir proved the NP-completeness of SIMPLE TWO-COMMODITY INTEGRAL FLOW
  - Both in the directed and undirected cases.
- For the **directed case**, they proved that the problem is still NP-complete if the demand of one commodity is **unitary**.

Introduction: SIMPLE TWO-COMMODITY INTEGRAL FLOW complexity

- In 1976, Even, Itai and Shamir proved the NP-completeness of SIMPLE TWO-COMMODITY INTEGRAL FLOW
  - Both in the directed and undirected cases.
- For the **directed case**, they proved that the problem is still NP-complete if the demand of one commodity is **unitary**.
- Nevertheless, for the **undirected case**, the hard instance constructed by them does not satisfy the condition of a demand to be unitary or even bounded by a constant.

Introduction: SIMPLE TWO-COMMODITY INTEGRAL FLOW complexity

- In 1976, Even, Itai and Shamir proved the NP-completeness of SIMPLE TWO-COMMODITY INTEGRAL FLOW
  - Both in the directed and undirected cases.
- For the **directed case**, they proved that the problem is still NP-complete if the demand of one commodity is **unitary**.
- Nevertheless, for the **undirected case**, the hard instance constructed by them does not satisfy the condition of a demand to be unitary or even bounded by a constant.

- The main goal of our work is to **close this forty-year complexity gap**.

Simple undirected two-commodity integral flow (Simple U2CIF)

Input: An undirected graph G, two *commodities* $\{s_1, t_1\}$ and $\{s_2, t_2\}$, where s_1, t_1, s_2 and t_2 are vertices of G, and two *demands* $D_1, D_2 \in \mathbb{Z}^+$.

Question: Are there two flow functions $f_1, f_2 \colon \{\overrightarrow{uv}, \overrightarrow{vu} \mid uv \in E(G)\} \to \mathbb{Z}_0^+$ such that

1. for each $uv \in E(G)$ and $i \in \{1, 2\}$,
$$f_i(\overrightarrow{uv}) = 0 \text{ or } f_i(\overrightarrow{vu}) = 0;$$
2. for each $uv \in E(G)$, the total flow through uv does not exceed its *unitary capacity*, *i.e.*
$$\max\{f_1(\overrightarrow{uv}), f_1(\overrightarrow{vu})\} + \max\{f_2(\overrightarrow{uv}), f_2(\overrightarrow{vu})\} \leq 1;$$
3. for each $i \in \{1, 2\}$ and $v \in V \setminus \{s_i, t_i\}$, the flow function f_i is *conserved* at v, *i.e.*
$$\sum_{x \in N_G(v)} f_i(\overrightarrow{xv}) = \sum_{y \in N_G(v)} f_i(\overrightarrow{vy}); \text{ and}$$
4. for each $i \in \{1, 2\}$,
$$F_i = \sum_{v \in N_G(t_i)} f_i(\overrightarrow{vt_i}) \geq D_i?$$

NP-completeness of SIMPLE U2CIF with a unitary demand

- We prove that SIMPLE U2CIF remains **NP-complete** when the capacity of one commodity is unitary

NP-completeness of SIMPLE U2CIF with a unitary demand

- We prove that SIMPLE U2CIF remains **NP-complete** when the capacity of one commodity is unitary
  - By a polynomial-time reduction from **3-SAT**, where each clause has exactly three distinct literals.

NP-completeness of SIMPLE U2CIF with a unitary demand

- We prove that SIMPLE U2CIF remains **NP-complete** when the capacity of one commodity is unitary
  - By a polynomial-time reduction from **3-SAT**, where each clause has exactly three distinct literals.
- So, let $I = (X, \mathcal{C})$ be a such instance of 3-SAT, where X is the variable set and $\mathcal{C}$ is the clause set of I.

NP-completeness of SIMPLE U2CIF with a unitary demand

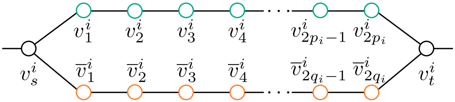
- We prove that SIMPLE U2CIF remains **NP-complete** when the capacity of one commodity is unitary
  - By a polynomial-time reduction from **3-SAT**, where each clause has exactly three distinct literals.
- So, let $I = (X, \mathcal{C})$ be a such instance of 3-SAT, where X is the variable set and $\mathcal{C}$ is the clause set of I.
- We construct from I an instance $g(I) = (G, \{s_1, t_1\}, \{s_2, t_2\}, D_1, D_2)$ of SIMPLE U2CIF as follows.

SIMPLE U2CIF: construction of the instance $g(I)$

- We define $D_1 = 1$ and $D_2 = 5m$, where $m = |\mathcal{C}|$.

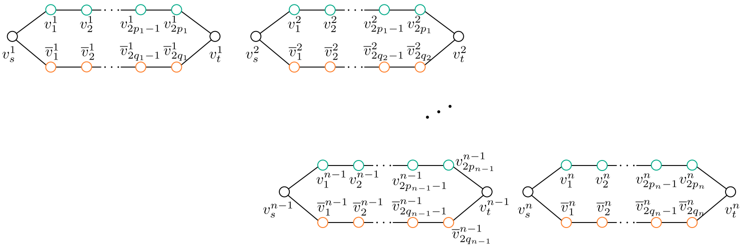
SIMPLE U2CIF: construction of the instance $g(I)$

- We define $D_1 = 1$ and $D_2 = 5m$, where $m = |\mathcal{C}|$.
- For each variable $x_i \in X$, we create the gadget G_{x_i}:

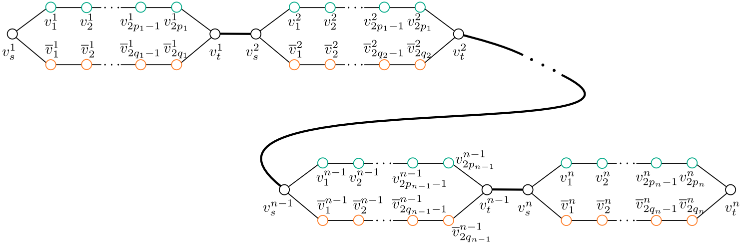


  - p_i: the number of occurrences of the **positive literal** x_i;
  - q_i: the number of occurrences of the **negative literal** $\overline{x}_i$.

- We connect the gadgets G_{x_i} to one another in series:

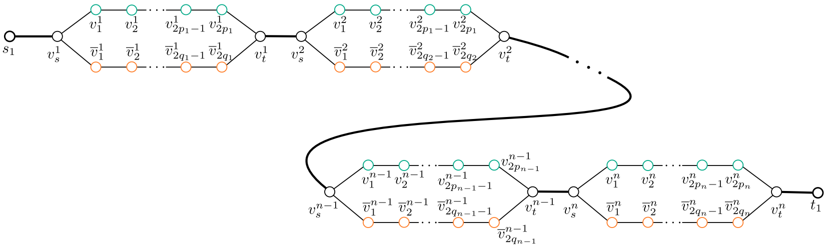


- We connect the gadgets G_{x_i} to one another in series:



SIMPLE U2CIF: construction of the instance $g(I)$

- We connect the gadgets G_{x_i} to one another in series:



SIMPLE U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

SIMPLE U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

- we create the *clause vertices* u_{c_ι} and w_{c_ι}

SIMPLE U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

- we create the *clause vertices* u_{c_ι} and w_{c_ι}

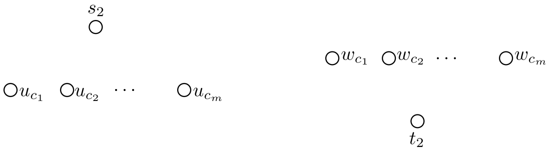
Simple U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;

For each clause $C_\iota \in \mathcal{C}$,

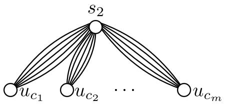
- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;

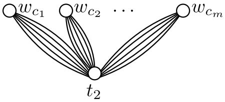


SIMPLE U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;

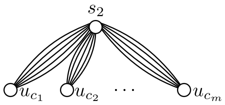


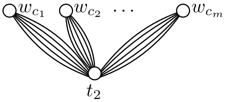


Simple U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;
- we create the *literal vertices* $u^\iota_{l_1}, u^\iota_{l_2}, u^\iota_{l_3}$ and $w^\iota_{l_1}, w^\iota_{l_2}, w^\iota_{l_3}$, and we add the edges $u^\iota_{l_i} u_{c_\iota}$ and $w^\iota_{l_i} w_{c_\iota}$, for $1 \leq i \leq 3$;

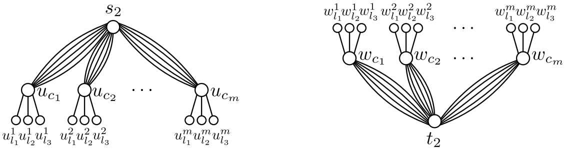




Simple U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

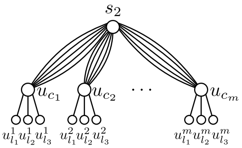
- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;
- we create the *literal vertices* $u^\iota_{l_1}, u^\iota_{l_2}, u^\iota_{l_3}$ and $w^\iota_{l_1}, w^\iota_{l_2}, w^\iota_{l_3}$, and we add the edges $u^\iota_{l_i} u_{c_\iota}$ and $w^\iota_{l_i} w_{c_\iota}$, for $1 \leq i \leq 3$;

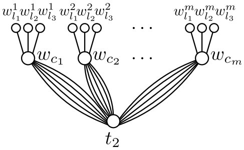


SIMPLE U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;
- we create the *literal vertices* $u^\iota_{l_1}, u^\iota_{l_2}, u^\iota_{l_3}$ and $w^\iota_{l_1}, w^\iota_{l_2}, w^\iota_{l_3}$, and we add the edges $u^\iota_{l_i} u_{c_\iota}$ and $w^\iota_{l_i} w_{c_\iota}$, for $1 \le i \le 3$;
- we create the vertices y^ι_1, y^ι_2 and z^ι_1, z^ι_2

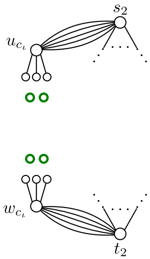




SIMPLE U2CIF: construction of the instance $g(I)$

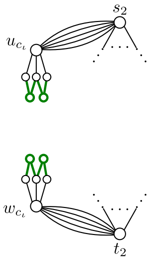
For each clause $C_\iota \in \mathcal{C}$,

- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;
- we create the *literal vertices* $u^\iota_{l_1}, u^\iota_{l_2}, u^\iota_{l_3}$ and $w^\iota_{l_1}, w^\iota_{l_2}, w^\iota_{l_3}$, and we add the edges $u^\iota_{l_i} u_{c_\iota}$ and $w^\iota_{l_i} w_{c_\iota}$, for $1 \le i \le 3$;
- we create the vertices y^ι_1, y^ι_2 and z^ι_1, z^ι_2



For each clause $C_\iota \in \mathcal{C}$,

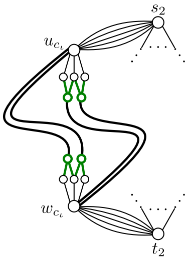
- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;
- we create the *literal vertices* $u^\iota_{l_1}, u^\iota_{l_2}, u^\iota_{l_3}$ and $w^\iota_{l_1}, w^\iota_{l_2}, w^\iota_{l_3}$, and we add the edges $u^\iota_{l_i} u_{c_\iota}$ and $w^\iota_{l_i} w_{c_\iota}$, for $1 \leq i \leq 3$;
- we create the vertices y^ι_1, y^ι_2 and z^ι_1, z^ι_2, and we add the following edges;



SIMPLE U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;
- we create the *literal vertices* $u^\iota_{l_1}, u^\iota_{l_2}, u^\iota_{l_3}$ and $w^\iota_{l_1}, w^\iota_{l_2}, w^\iota_{l_3}$, and we add the edges $u^\iota_{l_i}u_{c_\iota}$ and $w^\iota_{l_i}w_{c_\iota}$, for $1 \leq i \leq 3$;
- we create the vertices y^ι_1, y^ι_2 and z^ι_1, z^ι_2, and we add the following edges;



SIMPLE U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

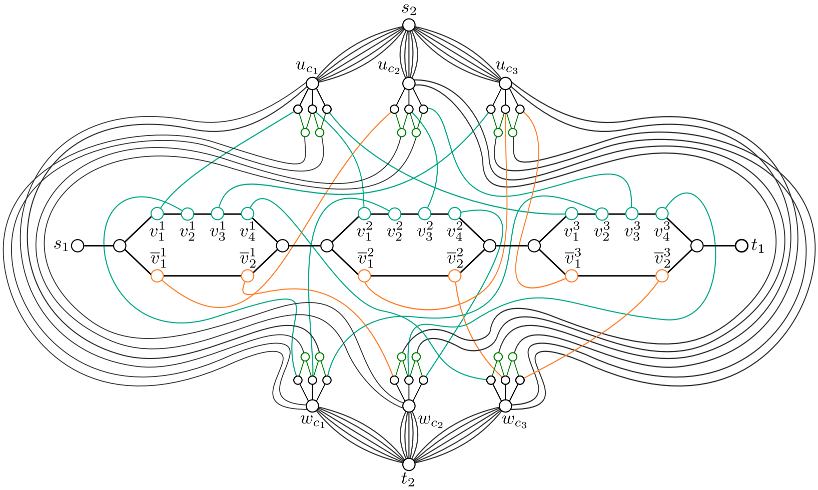
- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;
- we create the *literal vertices* $u^\iota_{l_1}, u^\iota_{l_2}, u^\iota_{l_3}$ and $w^\iota_{l_1}, w^\iota_{l_2}, w^\iota_{l_3}$, and we add the edges $u^\iota_{l_i} u_{c_\iota}$ and $w^\iota_{l_i} w_{c_\iota}$, for $1 \leq i \leq 3$;
- we create the vertices y^ι_1, y^ι_2 and z^ι_1, z^ι_2, and we add the following edges;
- Finally, we add the edges $u^\iota_{l_\kappa} v^i_{2j-1}$ and $v^i_{2j} w^\iota_{l_\kappa}$ if the κ-th literal in C_ι corresponds to the j-th occurrence of the **positive** literal $\boldsymbol{x_i}$;

SIMPLE U2CIF: construction of the instance $g(I)$

For each clause $C_\iota \in \mathcal{C}$,

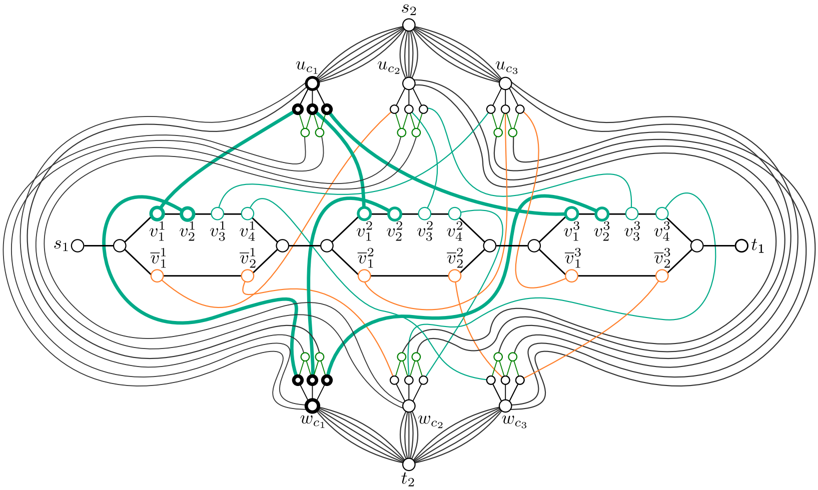
- we create the *clause vertices* u_{c_ι} and w_{c_ι}, and we add five parallel edges between s_2 and u_{c_ι} and five parallel edges between w_{c_ι} and t_2;
- we create the *literal vertices* $u^\iota_{l_1}, u^\iota_{l_2}, u^\iota_{l_3}$ and $w^\iota_{l_1}, w^\iota_{l_2}, w^\iota_{l_3}$, and we add the edges $u^\iota_{l_i} u_{c_\iota}$ and $w^\iota_{l_i} w_{c_\iota}$, for $1 \leq i \leq 3$;
- we create the vertices y^ι_1, y^ι_2 and z^ι_1, z^ι_2, and we add the following edges;
- Finally, we add the edges $u^\iota_{l_\kappa} v^i_{2j-1}$ and $v^i_{2j} w^\iota_{l_\kappa}$ if the κ-th literal in C_ι corresponds to the *j*-th occurrence of the **positive** literal $\boldsymbol{x_i}$;
- And, we add the edges $u^\iota_{l_\kappa} \overline{v}^i_{2j-1}$ and $\overline{v}^i_{2j} w^\iota_{l_\kappa}$ if the κ-th literal in C_ι corresponds to the *j*-th occurrence of the **negative** literal $\boldsymbol{\overline{x}_i}$.

$$I = (x_1 \lor x_2 \lor x_3) \land (\overline{x}_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor \overline{x}_3)$$



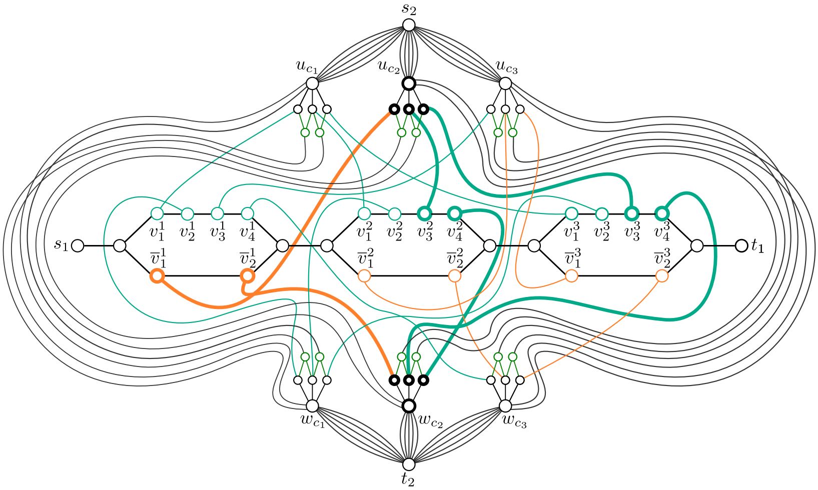
An example

$$I = (\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3}) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



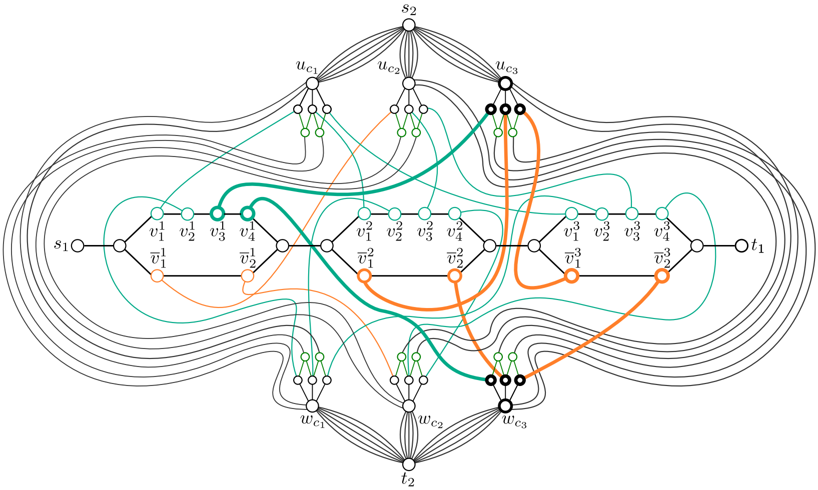
An example

$$I = (x_1 \vee x_2 \vee x_3) \wedge (\boldsymbol{\overline{x}_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



An example

$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\boldsymbol{x_1} \vee \overline{\boldsymbol{x_2}} \vee \overline{\boldsymbol{x}}_{\boldsymbol{3}})$$



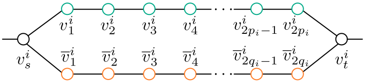
SIMPLE U2CIF: relation between the $g(I)$ and I instances

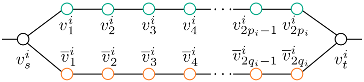
Lemma

If $g(I)$ is a YES *instance of* SIMPLE U2CIF*, then the first commodity flow uses only edges whose endpoints belong to* $\{s_1, t_1\} \cup V(G_{x_1}) \cup \cdots \cup V(G_{x_n})$.

Lemma

If $g(I)$ is a YES *instance of* SIMPLE U2CIF*, then the first commodity flow uses only edges whose endpoints belong to* $\{s_1, t_1\} \cup V(G_{x_1}) \cup \cdots \cup V(G_{x_n})$.

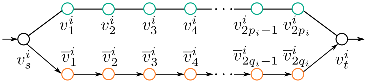


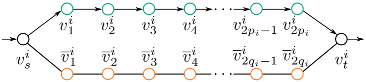


SIMPLE U2CIF: relation between the $g(I)$ and I instances

Lemma

If $g(I)$ is a YES *instance of* SIMPLE U2CIF*, then the first commodity flow uses only edges whose endpoints belong to* $\{s_1, t_1\} \cup V(G_{x_1}) \cup \cdots \cup V(G_{x_n})$.

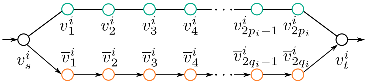


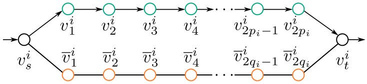


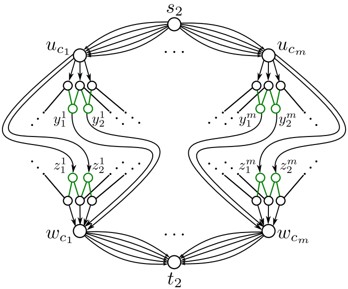
SIMPLE U2CIF: relation between the $g(I)$ and I instances

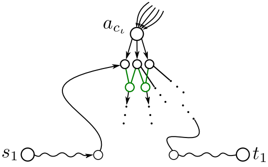
Lemma

If $g(I)$ is a YES *instance of* SIMPLE U2CIF, *then the first commodity flow uses only edges whose endpoints belong to* $\{s_1, t_1\} \cup V(G_{x_1}) \cup \cdots \cup V(G_{x_n})$.







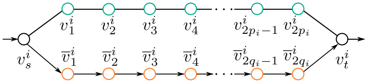


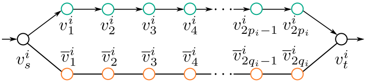
$a_{c_\iota} \in \{u_{c_\iota}, w_{c_\iota}\}$

SIMPLE U2CIF: relation between the $g(I)$ and I instances

Lemma

If $g(I)$ is a YES *instance of* SIMPLE U2CIF*, then the first commodity flow uses only edges whose endpoints belong to* $\{s_1, t_1\} \cup V(G_{x_1}) \cup \cdots \cup V(G_{x_n})$.





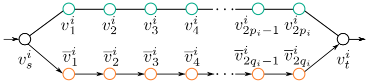
Lemma

$g(I)$ is Yes instance of SIMPLE U2CIF *if and only if I is a* YES *instance of 3-SAT.*

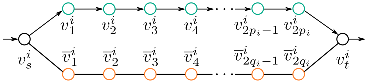
SIMPLE U2CIF: relation between the $g(I)$ and I instances

Lemma

If $g(I)$ is a YES *instance of* SIMPLE U2CIF*, then the first commodity flow uses only edges whose endpoints belong to* $\{s_1, t_1\} \cup V(G_{x_1}) \cup \cdots \cup V(G_{x_n})$.



$\alpha(x_i) =$ *true*.



$\alpha(x_i) =$ *false*.

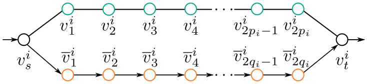
Lemma

$g(I)$ is Yes instance of SIMPLE U2CIF *if and only if I is a* YES *instance of 3-SAT.*

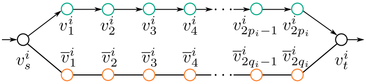
SIMPLE U2CIF: relation between the $g(I)$ and I instances

Lemma

If $g(I)$ is a YES *instance of* SIMPLE U2CIF*, then the first commodity flow uses only edges whose endpoints belong to $\{s_1, t_1\} \cup V(G_{x_1}) \cup \cdots \cup V(G_{x_n})$.*



$\alpha(x_i) =$ *true*.



$\alpha(x_i) =$ *false*.

Lemma

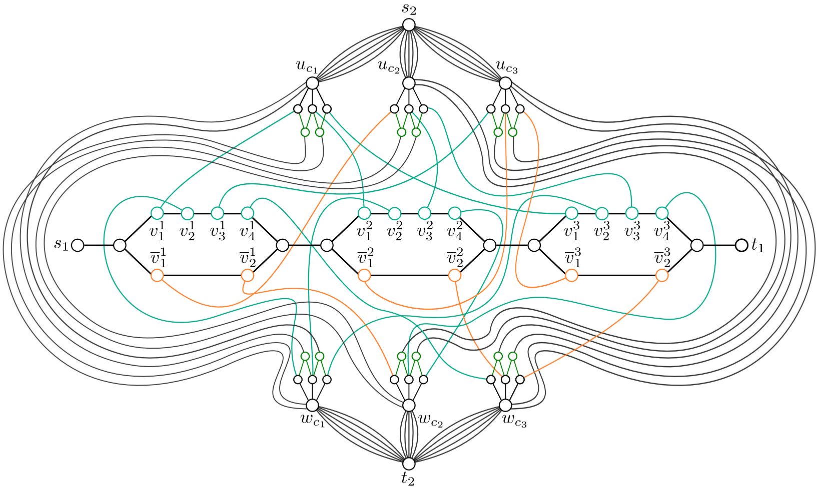
$g(I)$ is Yes instance of SIMPLE U2CIF *if and only if I is a* YES *instance of 3-SAT.*

Theorem

The SIMPLE U2CIF *problem is NP-complete even if the* **demand** *of one commodity is* **unitary***.*

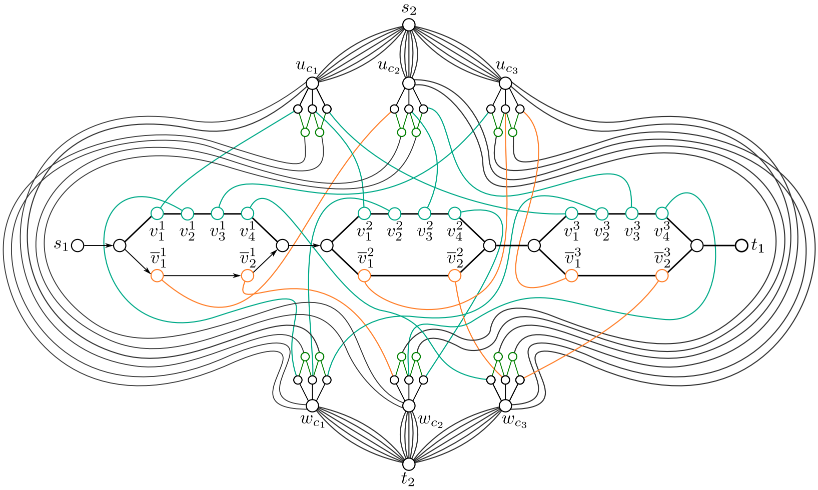
An example

$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



$\alpha(x_1) =$ *true* $\quad \alpha(x_2) =$ *true* $\quad \alpha(x_3) =$ *false*.

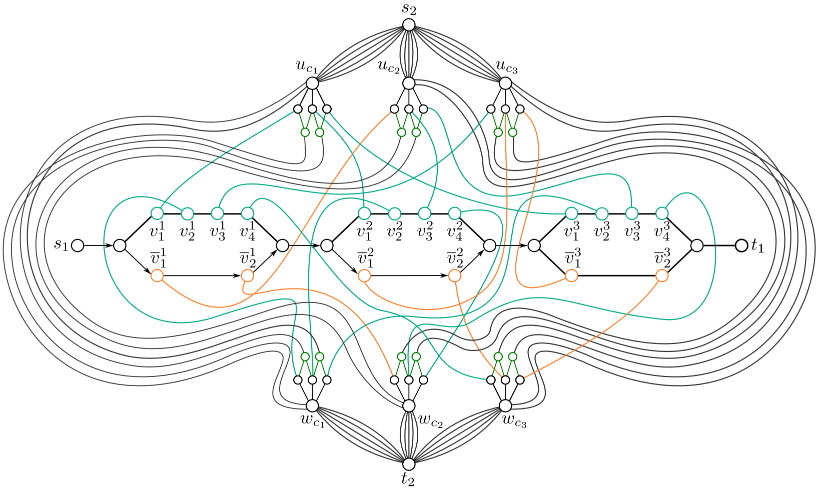
$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



$\alpha(x_1) =$ *true* $\quad \alpha(x_2) =$ *true* $\quad \alpha(x_3) =$ *false*.

An example

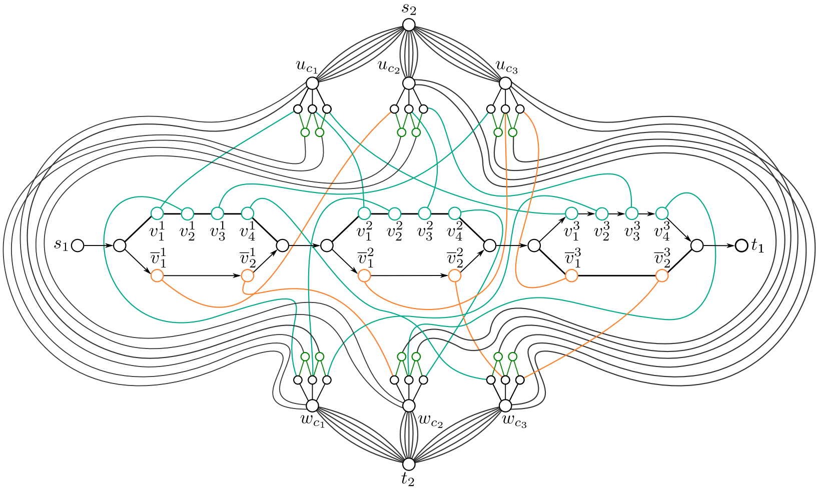
$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



$\alpha(x_1) =$ *true* $\quad \alpha(x_2) =$ *true* $\quad \alpha(x_3) =$ *false*.

An example

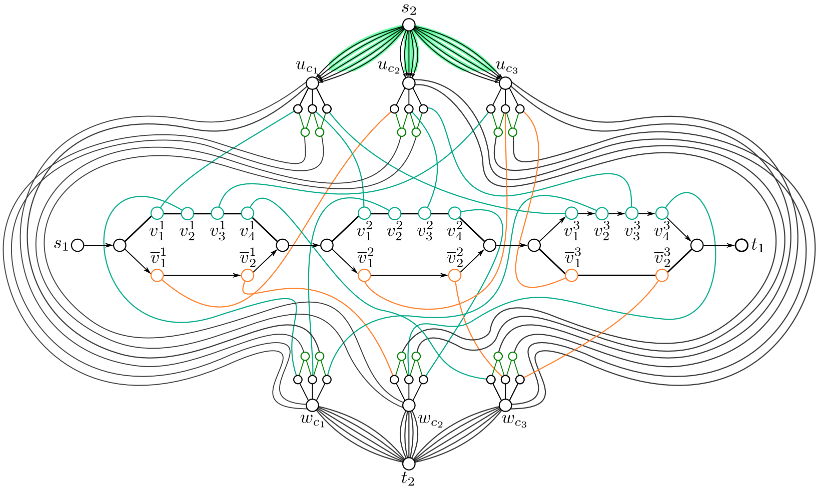
$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



$\alpha(x_1) =$ *true* $\quad \alpha(x_2) =$ *true* $\quad \alpha(x_3) =$ *false*.

An example

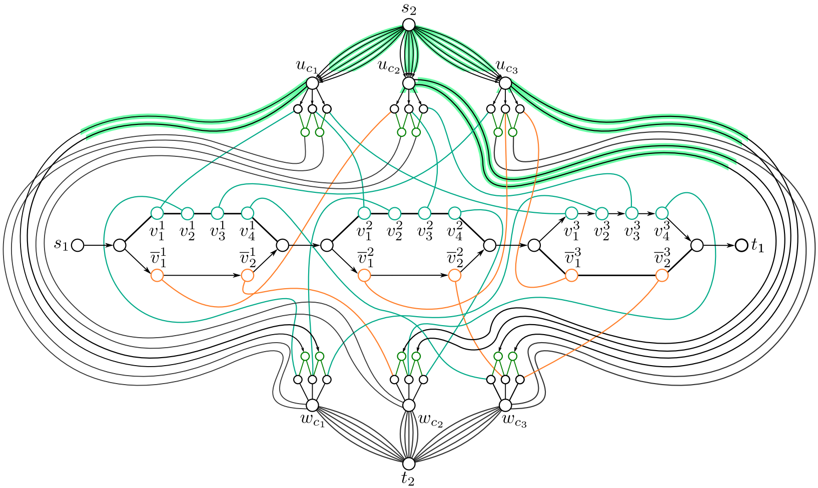
$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



$\alpha(x_1) = \textit{true} \quad \alpha(x_2) = \textit{true} \quad \alpha(x_3) = \textit{false}.$

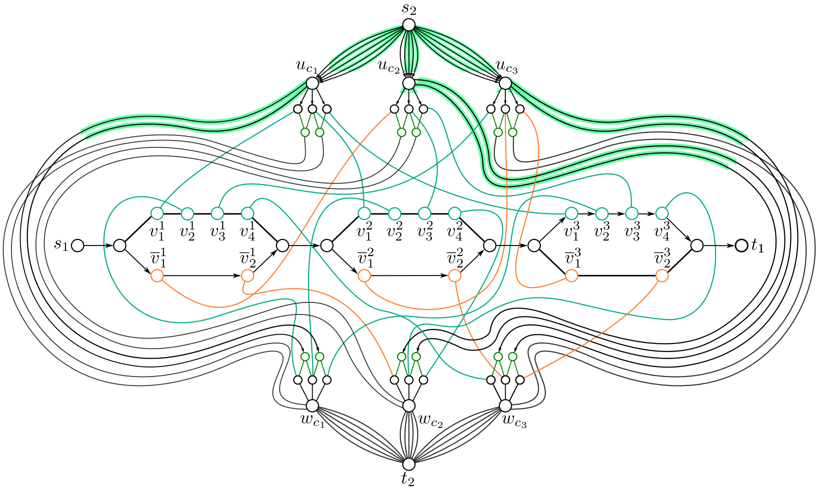
An example

$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



$\alpha(x_1) =$ *true* $\quad \alpha(x_2) =$ *true* $\quad \alpha(x_3) =$ *false*.

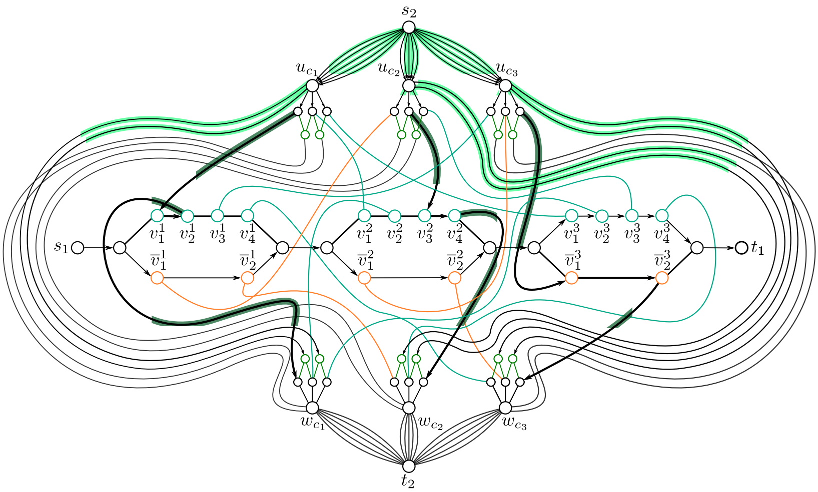
$$I = (\boldsymbol{x_1} \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \boldsymbol{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \boldsymbol{\overline{x}_3})$$



$\alpha(x_1) =$ *true* $\quad \alpha(x_2) =$ *true* $\quad \alpha(x_3) =$ *false*.

An example

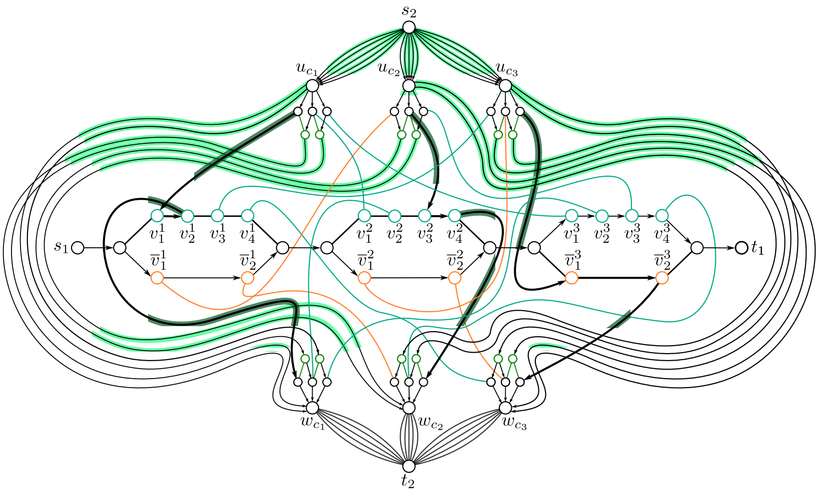
$$I = (\boldsymbol{x_1} \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \boldsymbol{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{\boldsymbol{x}}_{\boldsymbol{3}})$$



$\alpha(x_1) =$ *true* $\quad \alpha(x_2) =$ *true* $\quad \alpha(x_3) =$ *false*.

An example

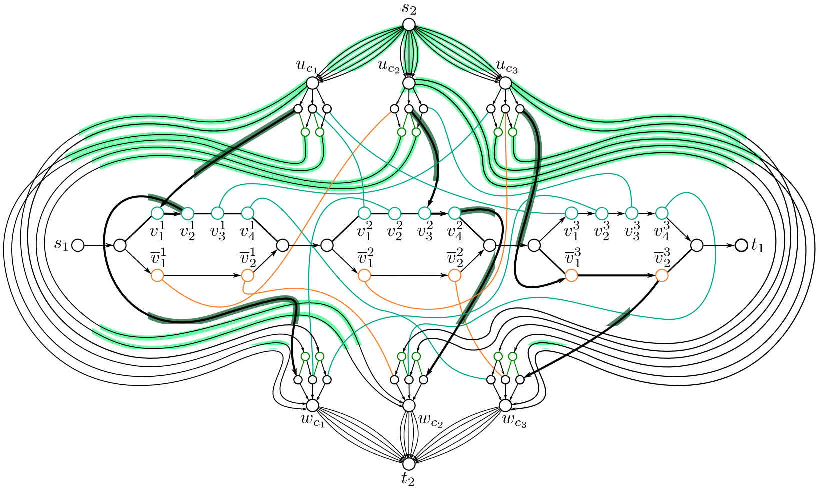
$$I = (\boldsymbol{x_1} \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \boldsymbol{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \boldsymbol{\overline{x}_3})$$



$\alpha(x_1) = \textit{true} \quad \alpha(x_2) = \textit{true} \quad \alpha(x_3) = \textit{false}.$

An example

$$I = (\boldsymbol{x_1} \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \boldsymbol{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{\boldsymbol{x}}_{\boldsymbol{3}})$$

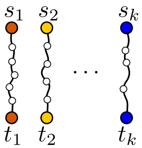


$\alpha(x_1) = \textit{true} \quad \alpha(x_2) = \textit{true} \quad \alpha(x_3) = \textit{false}.$

K+1 VERTEX-DISJOINT PATHS

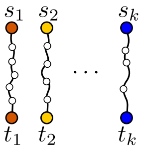
Disjoint paths problems

- SIMPLE U2CIF is closely related to *disjoint paths* problems
  - It can be reduced to the k-EDGE-DISJOINT PATHS problem.



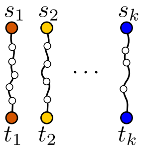
Disjoint paths problems

- SIMPLE U2CIF is closely related to *disjoint paths* problems
  - It can be reduced to the k-EDGE-DISJOINT PATHS problem.
  - Thus, k-EDGE-DISJOINT PATHS is NP-complete.



Disjoint paths problems

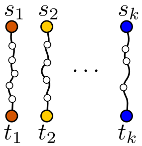
- SIMPLE U2CIF is closely related to *disjoint paths* problems
  - It can be reduced to the k-EDGE-DISJOINT PATHS problem.
  - Thus, k-EDGE-DISJOINT PATHS is NP-complete.



- In 1975, Karp proved the NP-completeness of k-VERTEX-DISJOINT PATHS.

Disjoint paths problems

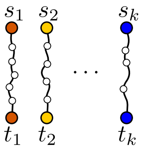
- SIMPLE U2CIF is closely related to *disjoint paths* problems
  - It can be reduced to the k-EDGE-DISJOINT PATHS problem.
  - Thus, k-EDGE-DISJOINT PATHS is NP-complete.



- In 1975, Karp proved the NP-completeness of k-VERTEX-DISJOINT PATHS.
- On the other hand, in 1995, Robertson and Seymour gave an $\mathcal{O}(n^3)$-time algorithm for k-VERTEX-DISJOINT PATHS, for **fixed** k.

Disjoint paths problems

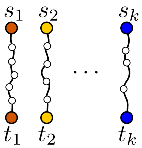
- SIMPLE U2CIF is closely related to *disjoint paths* problems
  - It can be reduced to the *k*-EDGE-DISJOINT PATHS problem.
  - Thus, *k*-EDGE-DISJOINT PATHS is NP-complete.



- In 1975, Karp proved the NP-completeness of *k*-VERTEX-DISJOINT PATHS.
- On the other hand, in 1995, Robertson and Seymour gave an $\mathcal{O}(n^3)$-time algorithm for *k*-VERTEX-DISJOINT PATHS, for **fixed** *k*.
- In 2011, Kawarabayashi et al. proposed an $\mathcal{O}(n^2)$-time algorithm for *k*-VERTEX-DISJOINT PATHS and *k*-EDGE-DISJOINT PATHS, for **fixed** *k*.

Disjoint paths problems

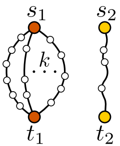
- SIMPLE U2CIF is closely related to *disjoint paths* problems
  - It can be reduced to the k-EDGE-DISJOINT PATHS problem.
  - Thus, k-EDGE-DISJOINT PATHS is NP-complete.



- In 1975, Karp proved the NP-completeness of k-VERTEX-DISJOINT PATHS.
- On the other hand, in 1995, Robertson and Seymour gave an $\mathcal{O}(n^3)$-time algorithm for k-VERTEX-DISJOINT PATHS, for **fixed** k.
- In 2011, Kawarabayashi et al. proposed an $\mathcal{O}(n^2)$-time algorithm for k-VERTEX-DISJOINT PATHS and k-EDGE-DISJOINT PATHS, for **fixed** k.
- Consequently, SIMPLE U2CIF is polynomial-time solvable if **both demands** are bounded by constants.

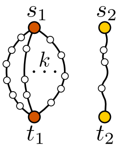
The $k+1$ disjoint paths problems

- Besides SIMPLE U2CIF to be related to *k*-EDGE-DISJOINT PATHS problem, note that SIMPLE U2CIF **coincides** with the $k+1$ EDGE-DISJOINT PATHS problem.



The $k + 1$ disjoint paths problems

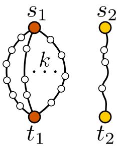
- Besides SIMPLE U2CIF to be related to k-EDGE-DISJOINT PATHS problem, note that SIMPLE U2CIF **coincides** with the $k + 1$ EDGE-DISJOINT PATHS problem.



- In this work, we additionally analyse the complexity of $k + 1$ VERTEX-DISJOINT PATHS problem.

The $k+1$ disjoint paths problems

- Besides Simple U2CIF to be related to k-edge-disjoint paths problem, note that Simple U2CIF **coincides** with the $k+1$ edge-disjoint paths problem.



- In this work, we additionally analyse the complexity of $k+1$ vertex-disjoint paths problem.

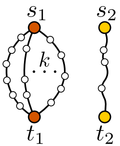
$k+1$ vertex-disjoint paths ($k+1$ VDP)

Input: An undirected graph G, two unordered pairs $\{s_1, t_1\}$ and $\{s_2, t_2\}$ of vertices of G and $k \in \mathbb{Z}^+$.

Question: Does G admit $k+1$ **vertex-disjoint paths**, such that one path is between s_1 and t_1 and k paths are between s_2 and t_2?

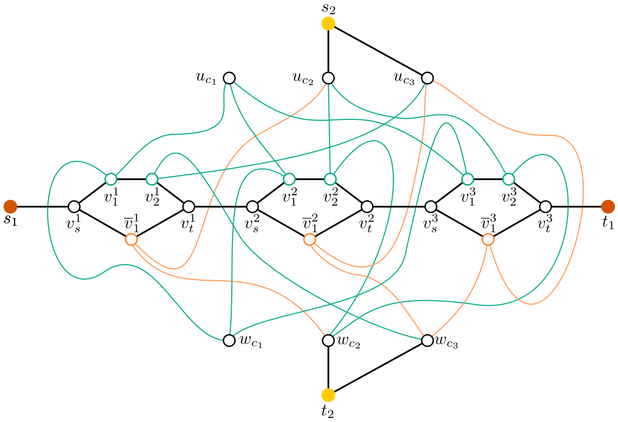
The $k+1$ disjoint paths problems

- Besides SIMPLE U2CIF to be related to k-EDGE-DISJOINT PATHS problem, note that SIMPLE U2CIF **coincides** with the $k+1$ EDGE-DISJOINT PATHS problem.

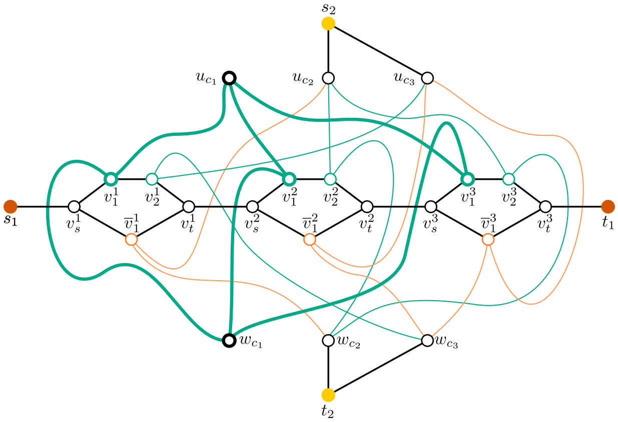


- In this work, we additionally analyse the complexity of $k+1$ VERTEX-DISJOINT PATHS problem.
- Similarly to SIMPLE U2CIF, we prove that $k+1$ VERTEX-DISJOINT PATHS is a NP-complete problem
  - By a polynomial-time reduction from **3-SAT**.

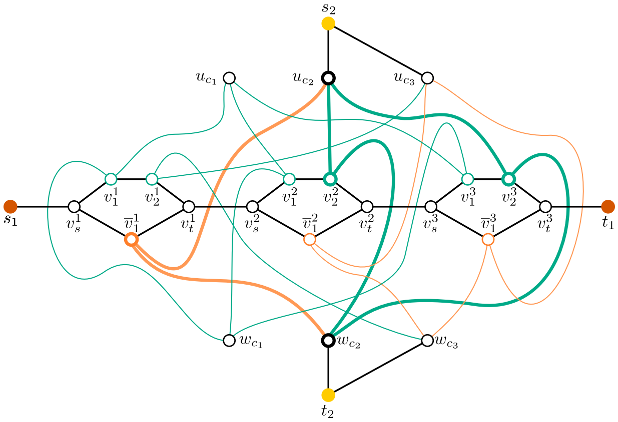
$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



$$I = (\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3}) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$

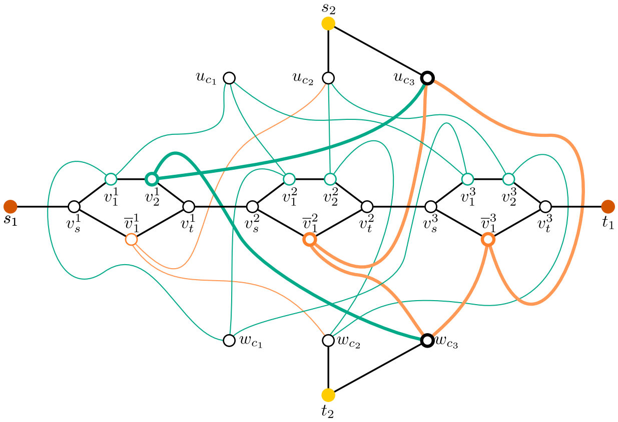


$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{\boldsymbol{x}}_{\mathbf{1}} \vee \boldsymbol{x}_{\mathbf{2}} \vee \boldsymbol{x}_{\mathbf{3}}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$



$k + 1$ VDP: construction of the instance $g(I)$

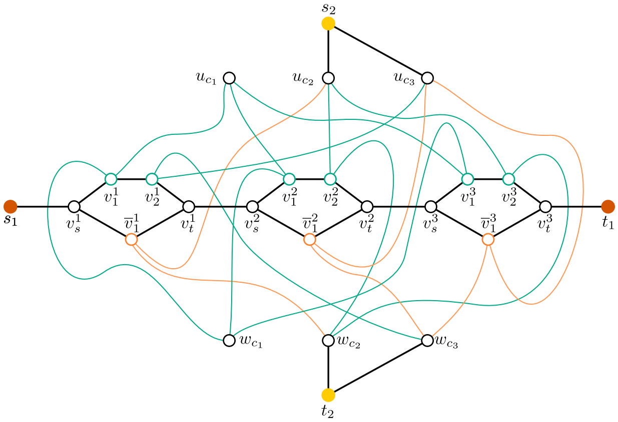
$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\boldsymbol{x_1} \vee \overline{\boldsymbol{x_2}} \vee \overline{\boldsymbol{x}}_{\boldsymbol{3}})$$



$k+1$ VDP: construction of the instance $g(I)$

$$I = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x}_3)$$

- $k = |\mathcal{C}|$ paths between s_2 and t_2 and 1 path between s_1 and t_1.



Conclusion

Conclusion

- In this work, we have proved that SIMPLE U2CIF is NP-complete even if the demand of one commodity is unitary, closing a forty-year complexity gap.

Conclusion

- In this work, we have proved that SIMPLE U2CIF is NP-complete even if the demand of one commodity is unitary, closing a forty-year complexity gap.
- Additionally, we have proved that $k + 1$ VERTEX-DISJOINT PATHS is also a NP-complete problem.

Conclusion

- In this work, we have proved that SIMPLE U2CIF is NP-complete even if the demand of one commodity is unitary, closing a forty-year complexity gap.
- Additionally, we have proved that $k+1$ VERTEX-DISJOINT PATHS is also a NP-complete problem.
- As future work, we intend to analyse the complexity of these problems when they are restricted to some specific graphs, *e.g.* planar graphs.

Thank you for your attention!

Alexsander A. Melo
`aamelo@cos.ufrj.br`

References I

- Cormen, H. T., Leiserson, E. C., Rivest, L. R and Stein, C., *Introduction to Algorithms*, MIT Press, third edition, 2009.
- Even, S., Itai, A. and Shamir, A., *On the complexity of timetable and multicommodity flow problems*, SIAM J. Comput. **5:4** (1976), 691–703.
- Edmonds, J. and Karp, R. M., *Theoretical improvements in algorithmic efficiency for network flow problems*, J. ACM **19:2** (1972), 248–264.
- Fortune, S., Hopcroft, J. and Wyllie, J., *The directed subgraph homeomorphism problem*, Theoret. Comput. Sci. **10** (1980), 111–121.
- Karp, R. M., *On the computational complexity of combinatorial problems*, Networks **5** (1975), 45–68.
- Kawarabayashi, K., Kobayashi, Y. and Reed, B., *The disjoint paths problem in quadratic time*, J. Combin. Theory Ser. B **102** (2012), 424–435
- Perl, Y. and Shiloach, Y., *Finding two disjoint paths between two pairs of vertices in a graph*, J. ACM **25:1** (1978), 1–9.
- Roberson, N. and Seymour, P. D., *Graph minors. XIII. The disjoint paths problem*, J. Combin. Theory Ser. B **63:1** (1995), 651–10.
- Tali, E., *The disjoint shortest paths problem*, Discrete Appl. Math. **85** (1998), 113–138.