Formalising Combinatorial Optimisation

Mohammad Abdulaziz

King's College London

September 16, 2025

Combinatorial Optimisation (CO)

Wikipedia:

'Subfield of mathematical optimisation that consists of finding an optimal object from a finite set of objects...'

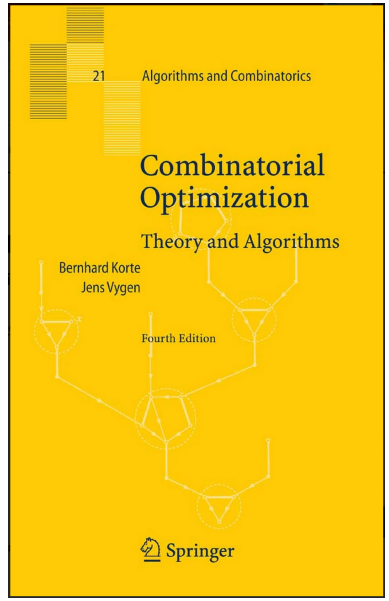
Examples:

- Shortest Paths, Spanning Trees, Matching, Flows, Travelling Salesman, Set Covers, Linear Programming, etc.

Combinatorial Optimisation (CO)

Project:

- A formal library of graduate/research-level results in CO
  - Books like Korte and Vygen's or Schreijver's
  - Also, research-y results
  - Focus on polynomial time algorithms
- Verified, efficient, executable implementations of CO algorithms
  - Covering most CO algorithms in CAS's like Magma, Sage, or Macaulay



Combinatorial Optimisation (CO)

Other work on formalising combinatorial optimisation

- Shortest paths: Dijkstra's, Floyd-Warshall, etc
- Maximum flows in Mizar [Lee 2005] and Isabelle [Lammich and Sefidgar 2019]
- Approximation algorithms in Isabelle [Nipkow et al. 2020]
- The Simplex algorithm in Isabelle [Maric et al. 2018]
- Spanning trees in Isabelle [Lammich and Nipkow 2019]

Using many different representations and methodologies

Combinatorial Optimisation (CO)

Why have a formal library of CO results?

- Mathematics behind many of the results is complex and needs verification
  - Matching algorithms, graph colouring algorithms, etc.
- Verified SW for applications
  - Safety-critical applications: Kidney exchange (matching), air traffic control (flows), path planning for UAVs (shortest path)
  - Reliable tool for mathematical proof discovery: many depend on CO computations, e.g. matroid computations in CAS
- Unified representation and methodology
  - Previous attempts used different representations
  - Enabling reuse of results and focus on mathematics

Combinatorial Optimisation (CO)

Started as a collaboration between me and Kurt Mehlhorn

Grew into ca. 123K loc

Main contributors:

- Thomas Ammer, Ralista Dordjonova, Lukas Koller, Mitja Krebs, Christoph Madlener, Shriya Meenakshisundaram, Kurt Mehlhorn, Adem Rempapa

Initially, all contributions were on my disk or in separate repo's

- A lot of work by students doing masters or other projects

Contributions are now added as pull requests

- 12 PRs

Each new contribution is an Isabelle session

- Like the AFP

Combinatorial Optimisation (CO)

Paths: DFS, BFS, Floyd-Warshall

Matchings:

- Edmonds' blossom shrinking algorithm
- RANKING algorithm for online matching
- Tutte's theorem, Tutte/Berge formula

Maximum flows: Dinic's algorithm

Minimum cost flows: Orlin's algorithm

Matching/LP connection: Integrality of the matching polytope

Matroids and greedoids:

- Greedy algorithms for matroid and greedoid optimisation
- Maximising matroid intersection

Spanning trees: Kruskal and Prim's algorithms

TSP: Christofides' algorithm

Combinatorial Optimisation (CO)

Goals here:

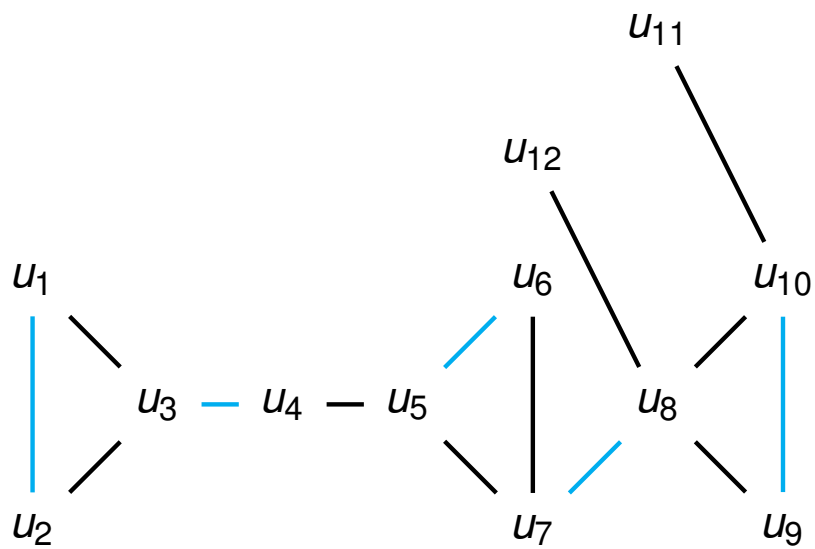
- Demonstrate
  - The mathematics of CO via an example
  - Reasoning patterns one needs to perform
  - Objects one needs to model

Discuss future directions

Maximum Cardinality Matching

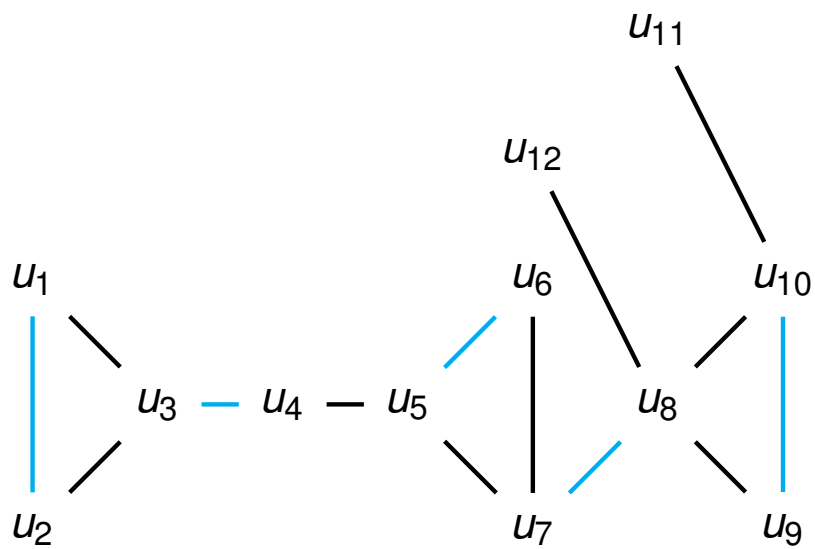
A matching M is a set of edges no two of which share a vertex

The cyan edges form a matching of maximum cardinality



Edmonds' Blossom Shrinking Algorithm

Computes a maximum cardinality matching for undirected graphs

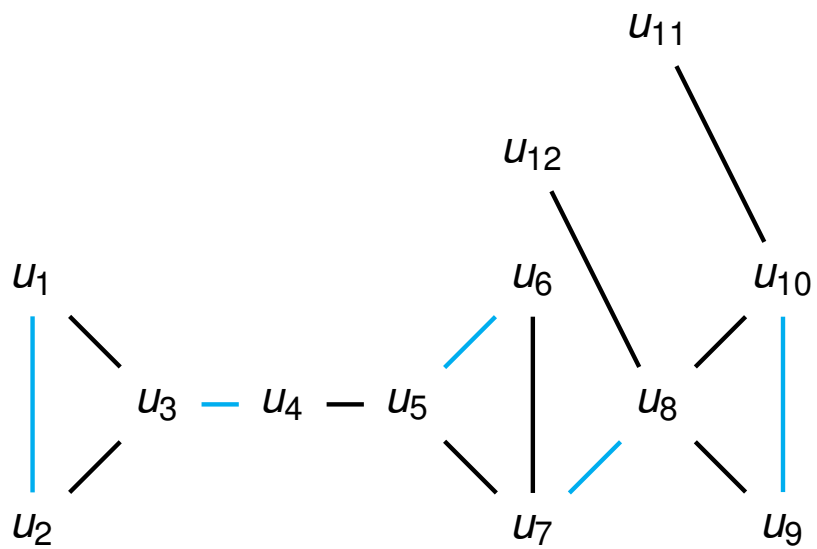


Edmonds' Blossom Shrinking Algorithm

Goal:

- Formalise correctness proof in LEDA [Mehlhorn & Näher 1998]

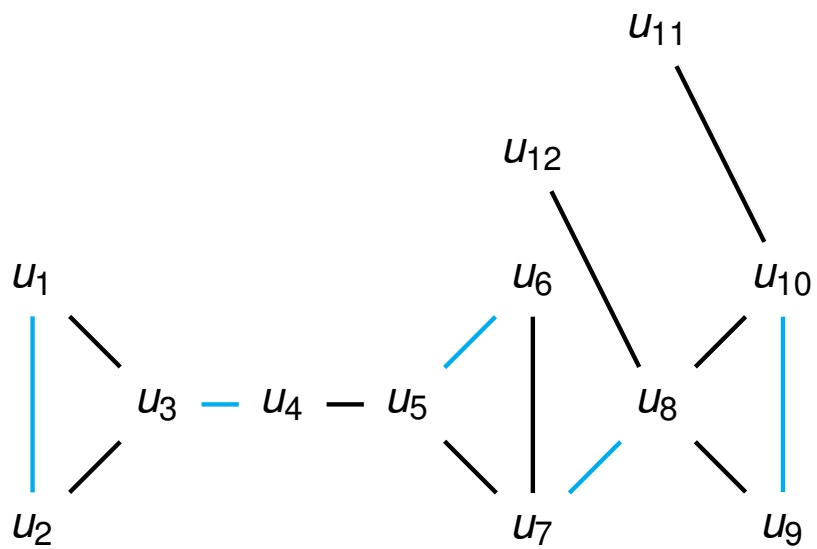
In collaboration with Kurt Mehlhorn



Edmonds' Blossom Shrinking Algorithm

Why?

- Inspired Edmonds to see polytime as effective computation
- Benchmark in the scalability of formalisation
  - Second most "complicated" algorithm in LEDA
  - Mehlhorn estimated it would be a many person-year project



Top Loop

FIND_MAX_MATCHING $(\mathcal{G}, \mathcal{M})$

$\gamma :=$ AUG_PATH_SEARCH$(\mathcal{G}, \mathcal{M})$
if γ is some augmenting path
  return FIND_MAX_MATCHING $(\mathcal{G}, \mathcal{M} \oplus E(\gamma))$
else
  return $\mathcal{M}$

$\oplus$ is the symmetric difference of two sets

The algorithm grows the matching using *augmenting paths*

Top Loop

FIND_MAX_MATCHING $(\mathcal{G}, \mathcal{M})$

$\gamma :=$ AUG_PATH_SEARCH$(\mathcal{G}, \mathcal{M})$
if γ is some augmenting path
  return FIND_MAX_MATCHING $(\mathcal{G}, \mathcal{M} \oplus E(\gamma))$
else
  return $\mathcal{M}$

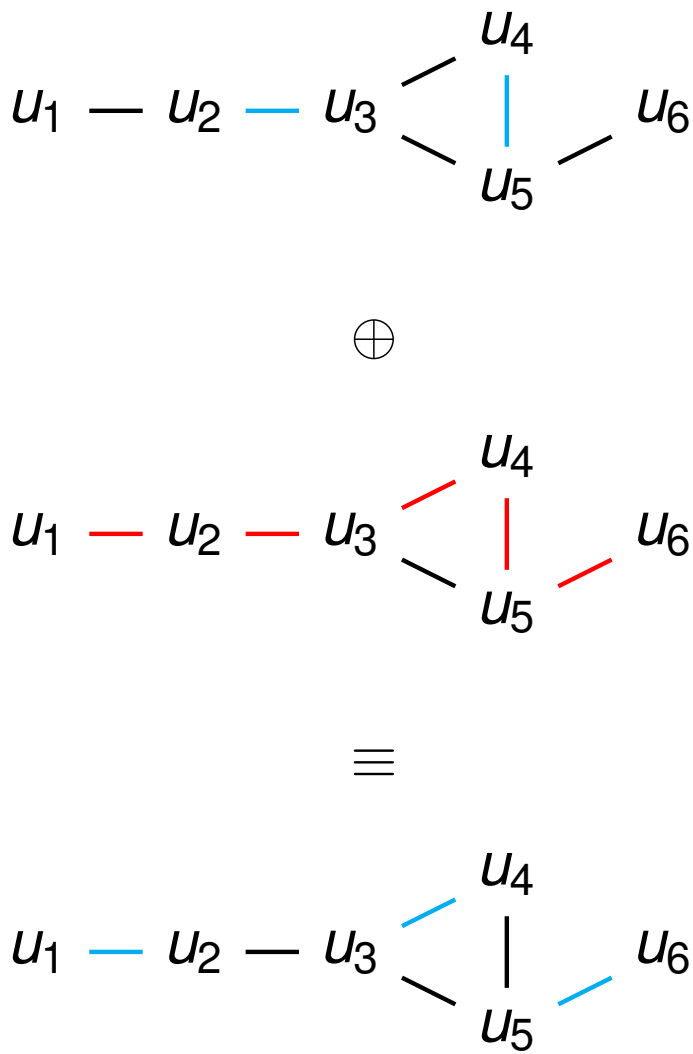
$\oplus$ is the symmetric difference of two sets

The algorithm grows the matching using *augmenting paths*

A path augments a matching iff

- its two end vertices are *free*
- for every two consecutive edges in the path, one belongs to the matching and one does not

Top Loop



Top Loop

Top Loop

FIND_MAX_MATCHING $(\mathcal{G}, \mathcal{M})$

$\gamma :=$ AUG_PATH_SEARCH$(\mathcal{G}, \mathcal{M})$
if γ is some augmenting path
  return FIND_MAX_MATCHING $(\mathcal{G}, \mathcal{M} \oplus E(\gamma))$
else
  return $\mathcal{M}$

$\oplus$ is the symmetric difference of two sets

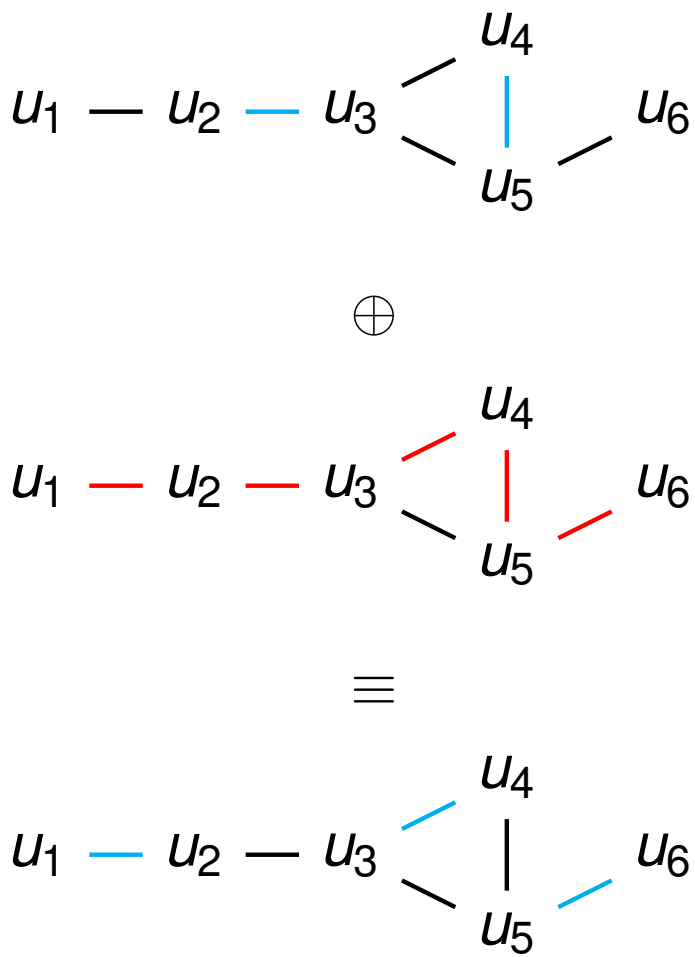
Correctness:

Lemma (Berge 1957)

A matching $\mathcal{M}$ has maximum cardinality iff it has no augmenting paths

Top Loop

($\Rightarrow$): the symmetric difference of an augmenting path with a matching is a bigger matching

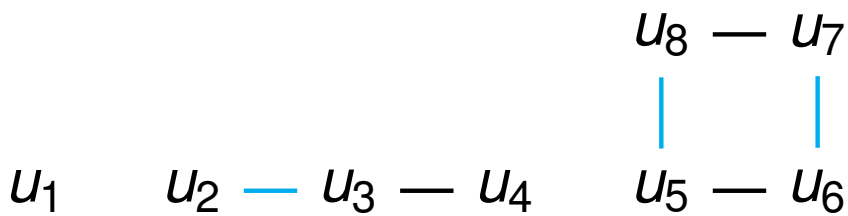


Top Loop

Top Loop

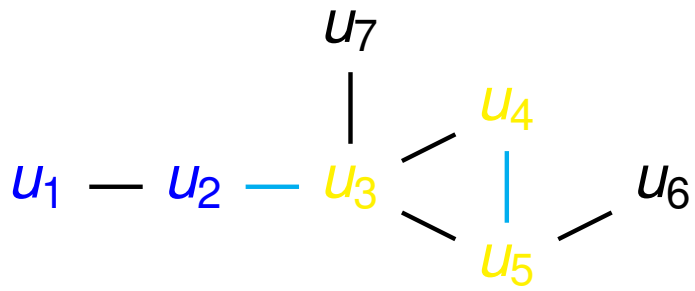
Standard proof of completeness is long

- Lemma: for any two matchings $\mathcal{M}$ and $\mathcal{M}'$, every connected component of the graph $\mathcal{M} \oplus \mathcal{M}'$ is either
  - a singleton vertex,
  - an alternating path, or
  - an even alternating cycle.



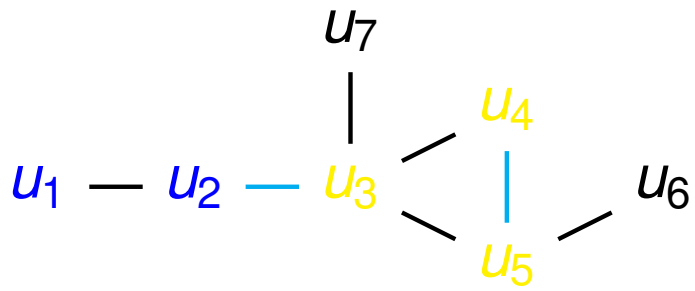
- This is a rather complex construction
- Avoided during formalisation, leading to a shorter proof

Augmenting Path/Blossom Search



Build an alternating path forest and use it to find augmeting paths or blossoms

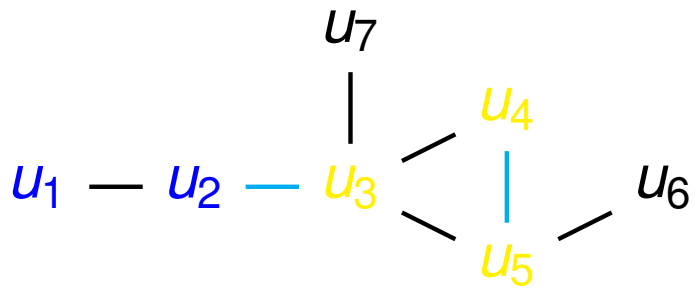
Augmenting Path/Blossom Search



Build an alternating path forest and use it to find augmeting paths or blossoms

Every vertex is labelled as odd/even depending on its distance from a forest root

Augmenting Path/Blossom Search



Build an alternating path forest and use it to find augmeting paths or blossoms

Every vertex is labelled as odd/even depending on its distance from a forest root

Edges are examined only once

Augmenting Path/Blossom Search

compute_alt_path($\mathcal{G}, \mathcal{M}$)

ex $= \emptyset$ // Set of examined edges
foreach $u \in \bigcup \mathcal{G}$ **do** label $u =$ None; parent $u =$ None **done**
$U = \bigcup \mathcal{G} \setminus \bigcup \mathcal{M}$ // Set of unmatched vertices
foreach $u \in U$ **do** label $u = \langle u, \text{even} \rangle$ **done**
while $(\mathcal{G} \setminus \text{ex}) \cap \{e \mid \exists u \in e, r \in \bigcup \mathcal{G}.\text{label } u = \langle r, \text{even} \rangle\} \neq \emptyset$
  // Choose a new edge and label it examined
  $\{u_1, u_2\} =$ choose $(\mathcal{G} \setminus \text{ex}) \cap \{\{u_1, u_2\} \mid \exists r.\text{label } u_1 = \langle r, \text{even} \rangle\}$
  ex = ex $\cup \{\{u_1, u_2\}\}$
  if label $u_2 =$ None
    // Grow the discovered set of edges from r by two
    $u_3 =$ choose $\{u_3 \mid \{u_2, u_3\} \in \mathcal{M}\}$
    ex = ex $\cup \{\{u_2, u_3\}\}$
    label $u_2 = \langle r, \text{odd} \rangle$; label $u_3 = \langle r, \text{even} \rangle$; parent $u_2 = u_1$; parent $u_3 = u_2$
  else if $\exists s \in \bigcup \mathcal{G}.\text{label } u_2 = \langle s, \text{even} \rangle$
    // Return two paths from current edge's tips to unmatched vertex(es)
    return $\langle$follow parent u_1, follow parent $u_2\rangle$
return No paths found

Augmenting Path/Blossom Search

Augmenting Path/Blossom Search

Partial Correctness: the loop always returns two paths s.t.

- They are alternating
- They each end in a free vertex
- Odd length
- …

Augmenting Path/Blossom Search

Partial Correctness: the loop always returns two paths s.t.

- They are alternating
- They each end in a free vertex
- Odd length
- …

Total Correctness:

- An “odd set cover” (OSC) as big as the matching is a certificate the matching is maximum
- If the loop terminates w/o returning two paths, then there is a certificate

Augmenting Path/Blossom Search

Rigorous proofs about while loops need “loop invariants”

- Statements about variables that are preserved along the loop’s execution

Augmenting Path/Blossom Search

Rigorous proofs about while loops need “loop invariants”

- Statements about variables that are preserved along the loop’s execution

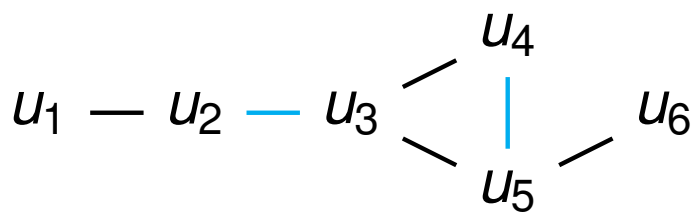
Our proof needs 18 loop invariants, e.g.

- If one vertex in a matching edge is labelled, then the other is labelled with opposite parity
- If a matching edge is examined, then both its vertices are labelled
- Any examined edge has an odd labelled vertex
- …

Augmenting Path/Blossom Search

Interesting insights gained:

- LEDA had 11 of those invariants
- Enough for proving partial correctness
- LEDA's certificate is w.r.t. full graph: $\{\{u_2\}, \{u_3, u_4, u_5\}, \{u_6\}\}$

$\mathcal{G} \equiv$ 

- Formalised certificate w.r.t. contracted graph: $\{\{u_2\}, \{u_6\}\}$

$\mathcal{G}/P_C \equiv$ 

Augmenting Path/Blossom Search

Interesting insights gained:

- LEDA had 11 of those invariants
- Enough for proving partial correctness
- The 11 invariants are not enough to prove that
- Invariants to do LEDA's certificate construction are more complicated than the 18 we formalised
- A much shorter proof of Berge's lemma

Augmenting Path/Blossom Search

I gained a deeper understanding of the standard correctness proof. I had never understood one step in the standard argument and therefore developed a different correctness argument for the LEDA book which avoids this step. The fact that I now understand this step is directly connected to the formalization effort. Working on the formalization forced me to understand the material more deeply than I had done before. The work also convinced me that my alternative proof is no simpler

Augmenting Path/Blossom Search

What is needed:

- We need a graph representation
  - Executable (for implementation) and for (abstract) mathematical reasoning
  - Need to connect both representations!
- We need complex combinatorial reasoning about cases
  - Matchings, alternating paths, connected components, etc
- Algorithm modelling:
  - While-loop: modelled using recursion
  - Program state: modelled using records
  - Loop invariants: modelled as predicates on states

Online Bi-Partite Matching

Setting:

- Bipartite graphs
- One party of the graph arrives online
- The algorithm has to compute its output as its input arrives
  - *Without* knowing the rest of the future input!

Online Bi-Partite Matching

x_1

x_2

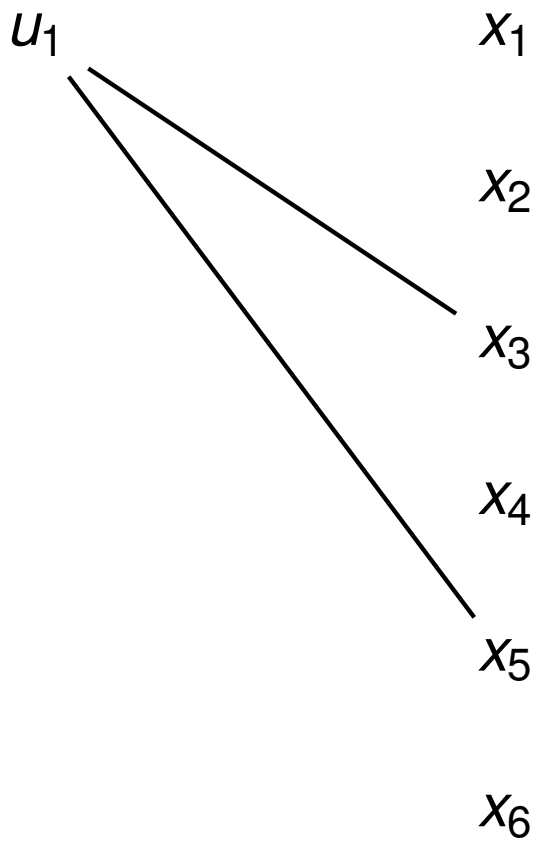
x_3

x_4

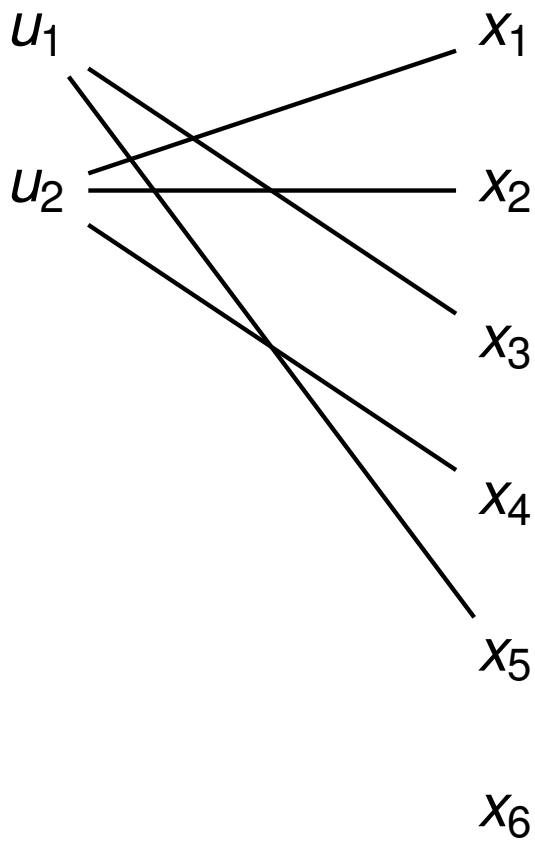
x_5

x_6

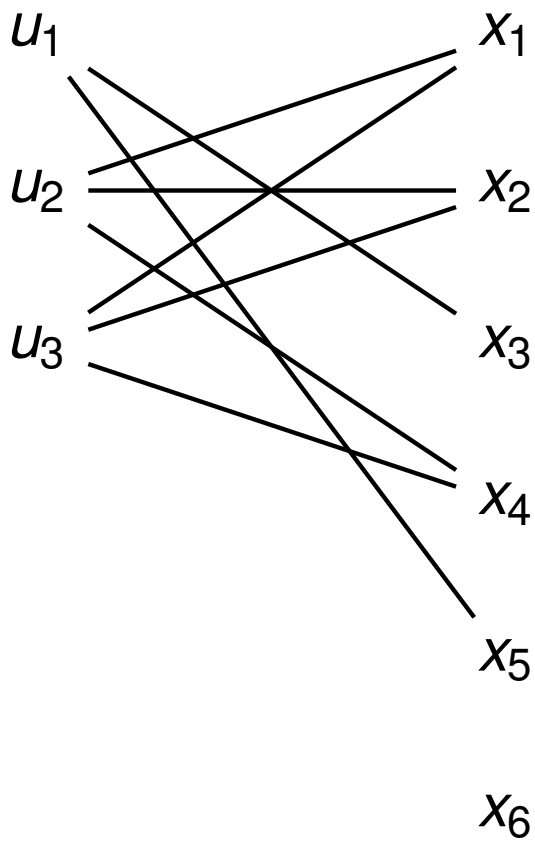
Online Bi-Partite Matching



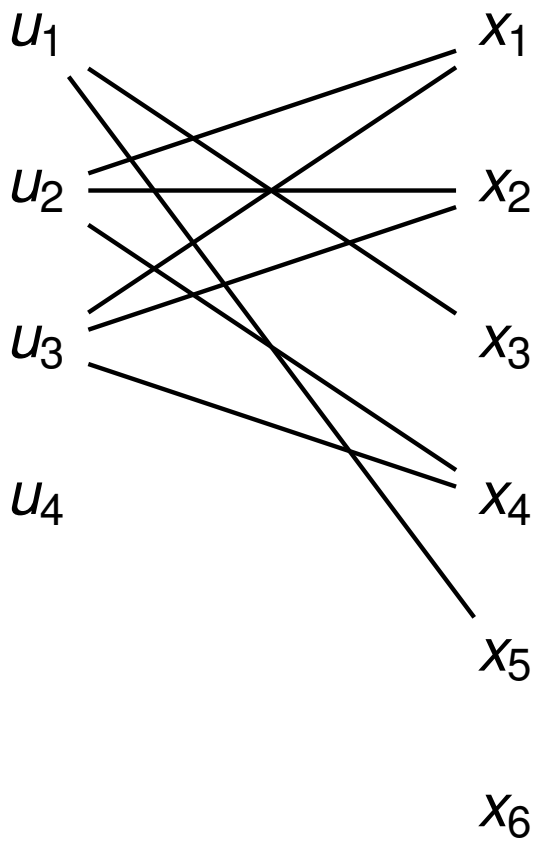
Online Bi-Partite Matching



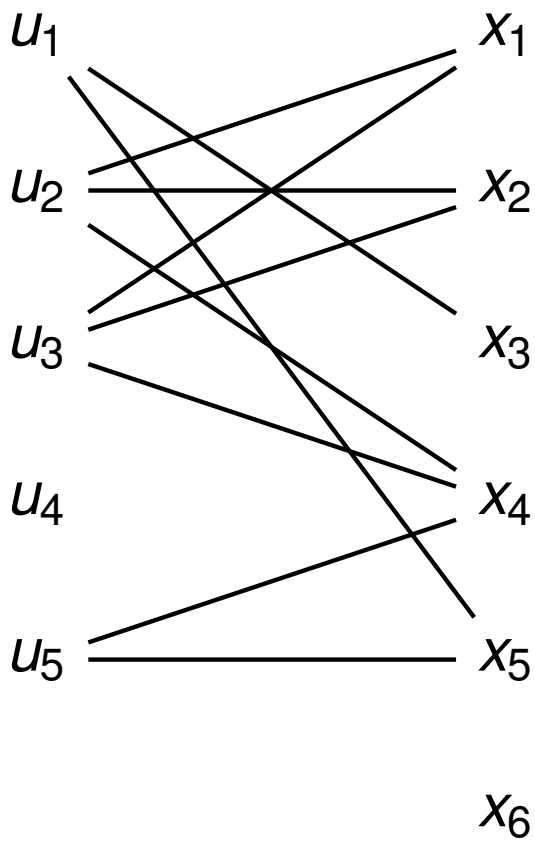
Online Bi-Partite Matching



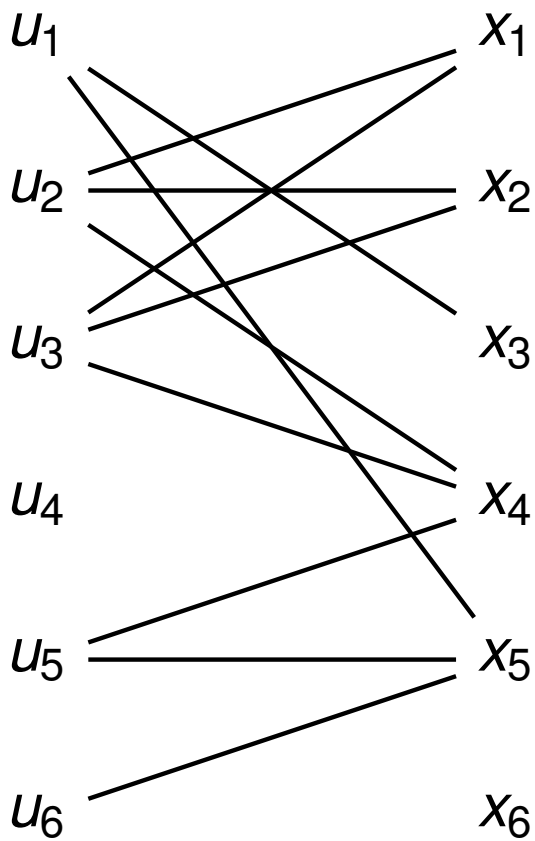
Online Bi-Partite Matching



Online Bi-Partite Matching



Online Bi-Partite Matching



Online Bi-Partite Matching

RANKING:

- By Karp, Vazirani and Vazirani 1990
- Generalised in many ways
  - Weights on vertices, edges, etc.
  - Notably, Adwords [Mehta et al. 2007]: models Google's Ad. market

Online Bi-Partite Matching

u_4

u_2

u_6

u_1

u_5

u_3

Online Bi-Partite Matching

u_1

u_2

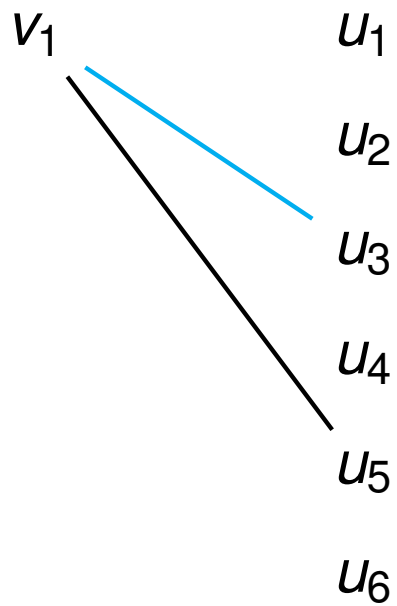
u_3

u_4

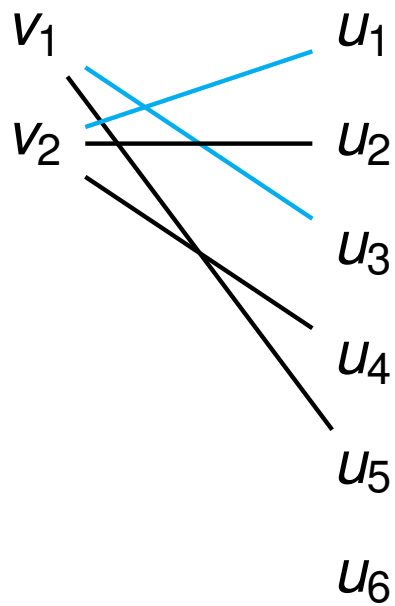
u_5

u_6

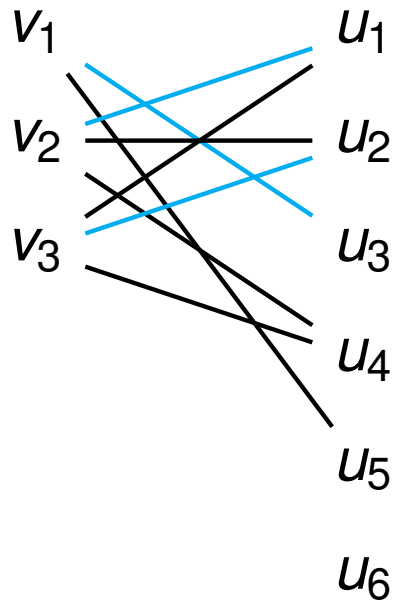
Online Bi-Partite Matching



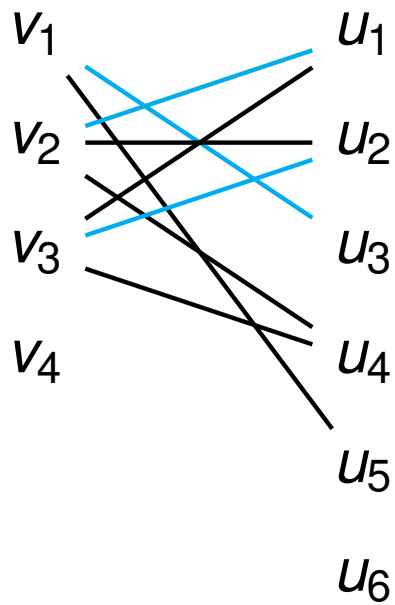
Online Bi-Partite Matching



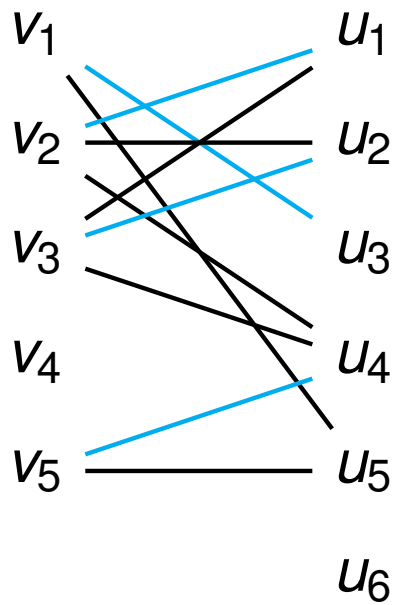
Online Bi-Partite Matching



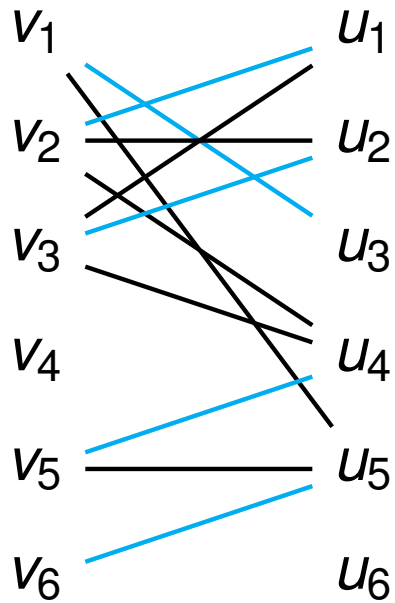
Online Bi-Partite Matching



Online Bi-Partite Matching



Online Bi-Partite Matching



Online Bi-Partite Matching

Algorithm 5: Pseudo-code of *RANKING*

function *RANKING(G, π)* **begin**
    $\sigma \leftarrow$ a random permutation of U
    return *online-match*$(\mathcal{G}, \pi, \sigma)$
end

Algorithm 6: Finding the highest ranked free neighbour

function *online-match(G, π, σ)* **begin**
    $\mathcal{M} \leftarrow \emptyset$
    for *every arriving vertex v in* π **do**
        if $\exists$ *unmatched u s.t. u is v's neighbour* **then** $\mathcal{M} \leftarrow \mathcal{M} \cup \{u, v\}$, where u is the top-rank unmatched neighbour of v
    return $\mathcal{M}$
end

Online Bi-Partite Matching

Theorem statement:

$$(1 - \frac{1}{e})|\mathcal{M}| \leq |\mathit{RANKING}(\mathcal{G}, \pi)|$$

where $\mathcal{M}$ is the largest matching one could compute knowing the entire graph a priori.

Average case over all possible permutations of the offline side

- A.k.a. competitiveness of the algorithm
- Probabilistic argument

The ratio is in the limit, w.r.t. the size of the given matching

- Asymptotic argument

Online Bi-Partite Matching

What is needed:

- We need a graph representation
- We need complex combinatorial reasoning about cases
- Algorithm modelling:
- Modelling online algorithms
  - Recursion over list of inputs
- Asymptotic reasoning
  - Tooling in Isabelle [Eberl 2019]
- Modelling and reasoning about randomised algorithms
  - Giry monads [Eberl et al. 2015]

Online Bi-Partite Matching

Proof of its competitiveness intensely studied

- Karp, Vazirani and Vazirani 1990
- Goel and A. Mehta 2008
- Birnbaum and C. Mathieu 2008
- Devanur, Jain, and Kleinberg 2013
- Eden, Feldman, Fiat, and Segal 2021

Online Bi-Partite Matching

Goal: Formalise the proof of Birnbaum and C. Mathieu

- “On-line bipartite matching made simple.”
- “Simple” combinatorial proof

Joint work with Christoph Madlener

Paper (ITP’23) url:

Formalising Karp-Vazirani-Vazirani

Outcome:

- One step is extremely hard to write down in detail
  - “Easy structural observation”
  - Graphical reasoning is usually very far from a formal proof

Hypothesis:

- The problem is more than just formalising the proof
- Verbalising a proof uncovers new insights and challenges

Matching-LP Connections

Finding a maximum matching in a graph can be encoded as a linear programming problem

- Could be used to compute matching using LP solvers
- The LP-perspective brought important insights
  - E.g. parallelisation of computing perfect matchings [Anari and Vazirani 2022]

Matching-LP Connections

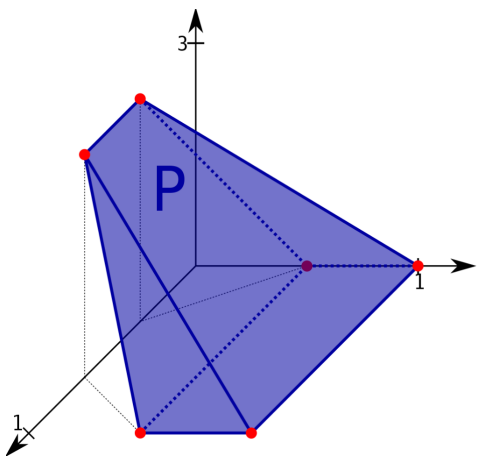
Until now, together with Ralitsa Dordjonova, we proved:

Theorem

Let $\mathcal{G}$ be a bipartite graph and let

$$P \equiv \textit{maximise} \sum_{e \in E(\mathcal{G})} x_e : \textit{Subject to} \sum_{e \in \textit{incident}(v)} x_e \leq 1$$

P is integral.



Matching-LP Connections

What is needed:

- We need a graph representation
- We need complex combinatorial reasoning about cases
- Algorithm modelling
- Modelling online algorithms
- Asymptotic reasoning
- Modelling and reasoning about randomised algorithms
- Totally unimodular matrices [Divason 2020]
- Linear programs, their duality [Thiemann 2023]
- Integral polyhedra, e.g. $\{x \mid Ax \leq b \wedge 0 \leq x\}$
- Lemmas/reasoning principles for geometric reasoning

Other work

Algorithms for minimum cost flows

- Scaling: a main technique for designing efficient algorithms
- In collab. with Ammer

Paper (ITP'24) url:

Other work

Algorithms for minimum cost flows

Algorithms for matroids

- Focus there is on algebraic reasoning
- Much more streamlined compared to directly reasoning about graphs
- In collab. with Ammer, Meenakshisundaram, and Rimpapa

Paper (ITP'25) url:

Other work

Algorithms for minimum cost flows

Algorithms for matroids

Approximation for TSP

- Approximation algorithm; uses a spanning tree and maximum-weight matching algorithms

Verifying Executable Algorithms: Data Types

Another topic I didn't discuss is how we reason about algorithms

- Recall: I mentioned a goal here is executable algorithms

I want to highlight one big part: data types

- I will focus on graphs, which are a principal object in combinatorial optimisation

Verifying Executable Algorithms: Data Types

We want an abstract representation to perform as much mathematical reasoning as possible

- E.g. Berge's lemma

Use existing concepts, e.g. sets, to represent digraphs:

$$'v \times' v\ set$$

Why?

- Less irrelevant details, better out of the box automation

Functions, e.g. neighbourhood of a vertex, is defined as

$$neighbourhood :: ('v \times 'v)\ set \Rightarrow 'v \Rightarrow 'v\ set$$
$$neighbourhood\ G\ u = \{v \mid (u, v) \in G\}$$

Verifying Executable Algorithms: Data Types

Problem: this representation is not immediately executable

For that, we need a programmatic data type

We use abstract data types (ADTs)

- (un)directed/multi/hyper graphs, linear programs, SAT formulae, etc

Verifying Executable Algorithms: Data Types

ADT *Pair_Graph_Specs* = *adjmap*: *Map* + *vset*: *Set_Choose* +

interface

$\emptyset_G$ :: *'adjmap* (**renaming** *adjmap.empty*)
update :: *'v* ⇒ *'vset* ⇒ *'adjmap* ⇒ *'adjmap* (**renaming** *adjmap.update*)
lookup :: *'adjmap* ⇒ *'v* ⇒ *'vset option* (**renaming** *adjmap.lookup*)
adjmap_inv :: *'adjmap* ⇒ *bool* (**renaming** *adjmap.inv*)

$\emptyset_V$ :: *'vset* (**renaming** *vset.empty*)
insert :: *'v* ⇒ *'vset* ⇒ *'vset* (**renaming** *vset.insert*)
isin :: *'vset* ⇒ *'v* ⇒ *bool* (**renaming** *vset.isin*)
t_set :: *'vset* ⇒ *'v set* (**renaming** *vset.set*)
vset_inv :: *'vset* ⇒ *bool* (**renaming** *vset.inv*)
sel :: *'vset* ⇒ *'v* (**renaming** *vset.sel*)

Verifying Executable Algorithms: Data Types

Based on such an ADT, an executable version of a vertex's neighbourhood is

$$\mathit{neighb} :: {'adjmap} \Rightarrow {'v} \Rightarrow {'vset}$$
$$\mathcal{N}_G\ G\ v = (\textbf{case}\ \mathit{lookup}\ G\ v\ \textbf{of}\ \mathit{None} \Rightarrow \emptyset_V \mid \mathit{Some}\ \mathit{vset} \Rightarrow \mathit{vset})$$

Recall that the abstract version was

$$\mathit{neighbourhood} :: ({'v} \times {'v})\ \mathit{set} \Rightarrow {'v} \Rightarrow {'v}\ \mathit{set}$$
$$\mathit{neighbourhood}\ G\ u = \{v \mid (u, v) \in G\}$$

Verifying Executable Algorithms: Data Types

Problem: we may have proved many facts on the mathematical representation of a graph *'v* $\times$ *'v set*

- Do we have to reprove these facts?

Verifying Executable Algorithms: Data Types

Solution: data type refinement

Devise abstraction functions connecting similar concepts

$$\textit{digraph\_abs} :: {'adjmap} \Rightarrow ({'v} \times {'v})\ \textit{set}$$
$$[G]_G = \{(u, v) \mid v \in_G \mathcal{N}_G\ G\ u\}$$

Using abstraction lemmas like these

$$\textit{graph\_inv}\ G \longrightarrow (v \in [\mathcal{N}_G\ G\ u]_s) = ((u, v) \in [G]_G)$$
$$\textit{graph\_inv}\ G \longrightarrow [\mathcal{N}_G\ G\ u]_s = \textit{neighbourhood}\ [G]_G\ u$$
$$\textit{graph\_inv}\ G \longrightarrow [\textit{add\_edge}\ G\ u\ v]_G = \textit{insert}\ (u, v)\ [G]_G$$

We have automation to 'transfer' facts about ADTs

Future Directions: Library Design

Graph representation

- One goal is to devise a single graph representation
- Difficult:
  - Executable algorithms vs. mathematical statements
  - Directed/undirected/multi/hyper graphs
  - Explicit set of vertices vs. not

Future Directions: Library Design

Graph representation

Computational object/algorithm representation

- Functional program: native to the theorem prover
- While-combinators: enable the use of program logics with
- Deeply embedded language: clear resource semantics
  - Modelling randomised, online, interactive computation

Future Directions: Library Design

Graph representation

Computational object/algorithm representation

Data type refinement using relational parametericity

- Automated refinement [Lammich 2013]
  - Steep learning curve, but less cumbersome if well-used
- Conditional transfer, a la [Cohen et al. 2025]

Future Directions: Formalisations

Edmonds' algorithm for weighted matching (with Ammer)

- Connects matching, LPs, primal-dual paradigm
- Most complicated implemented algorithm in LEDA

Future Directions: Formalisations

Edmonds' algorithm for weighted matching (with Ammer)

Resource/hardness analysis for optimisation algorithms/problems

- Complexity of approximation
- Done wrt a probabilistic computation model

Future Directions: Formalisations

Edmonds' algorithm for weighted matching (with Ammer)

Resource/hardness analysis for optimisation algorithms/problems

Micali-Vazirani algorithm for maximum cardinality matching

- First algorithm to achieve fastest running time of $O(\sqrt{n}m)$
- No accepted proof for 40 years

Future Directions: Formalisations

Giving a computer-aided proof of the Micali-Vazirani maximum matching algorithm, which is among the hardest combinatorial optimization results, is akin to climbing the Everest or building a World Champion Go program – it may not have direct uses, but it demonstrates the limits of our capabilities, so that faced with other challenging tasks, we may approach them with more confidence

Questions?