PACE Solver Description: Hydra Prime

Yosuke Mizutani
School of Computing, University of Utah, USA

David Dursteler
School of Computing, University of Utah, USA

Blair D. Sullivan
School of Computing, University of Utah, USA

Abstract

This note describes our submission to the 2023 PACE Challenge on the computation of twin-width. Our solver Hydra Prime combines modular decomposition with a collection of upper- and lower-bound algorithms, which are alternatingly applied on the prime graphs resulting from the modular decomposition. We introduce two novel approaches which contributed to the solver's winning performance in the Exact Track: *timeline encoding* and *hydra decomposition*. Timeline encoding is a new data structure for computing the width of a given contraction sequence, enabling faster local search; the hydra decomposition is an iterative refinement strategy featuring a small vertex separator.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Twin-width, PACE 2023

Supplementary Material Code submitted to the competition: `https://doi.org/10.5281/zenodo.7996823`. Code repository on GitHub: `https://github.com/TheoryInPractice/hydraprime`

1 Introduction

The 2023 Parameterized Algorithms & Computational Experiments (PACE) Challenge (`https://pacechallenge.org/2023/`) was to compute twin-width [3], a structural graph parameter which measures how close a given graph is to a *cograph* – a graph which can be reduced to a single vertex by repeatedly merging (*contracting*) pairs of *twins* – vertices with identical open neighborhoods. More generally, twin-width measures the minimum number of "mistakes" made in such a process when the pairs being contracted are no longer twins. If u and v are being merged, we say uy becomes a *red edge* if y is a neighbor of u but not v (and analogously for edges vy). The width of a contraction sequence is then the maximum number of red edges incident on any vertex (*red degree*) at any time during the process, and the twin-width of a graph is the minimum width of all valid contraction sequences. While graphs with bounded twin-width admit many FPT algorithms, computing the parameter is NP-hard, and prior to the PACE challenge its exact computation had remained impractical even on relatively small graphs.

Most twin-width solvers naturally begin by removing twins, as all groups of twins can be collapsed without incurring any red edges, making it a safe operation. In Hydra Prime, we employ a stronger notion of this via *modular decompositions* [5], which hierarchically decompose a graph into its twin-free *prime* components. A key property of these decompositions is that the twin-width of the original graph is exactly the maximum of the twin-width of the resulting prime graphs (Theorem 3.1 from [7]). We thus begin by running a re-implemented linear-time modular decomposition solver based on [9], then process each prime graph separately, maintaining a global lower bound. If a prime graph is a tree, we run PrimeTreeSolver, otherwise we run a series of lower- and upper-bound algorithms (listed at the end of this section) alternatively until the bounds match, from the quickest algorithms to the slowest. Those algorithms marked with (*) use a SAT solver as a subroutine; the implementation submitted to PACE uses the Kissat solver [2].

Algorithm List

- Exact algorithms
  - PrimeTreeSolver: Linear-time exact solver for trees without twins.
  - BranchSolver: Brute-force solver equipped with caching mechanism and reduction rules.
  - DirectSolver (*): SAT-based solver implementing the relative encoding presented in [7].
- Lower-bound algorithms
  - LBGreedy: Greedily removes a vertex u from the graph G such that $|\triangle(u, v)|$ is minimized for some v. Reports the maximum value of $\min_{u,v \in V(G), u \neq v} |\triangle_G(u, v)|$.
  - LBCore (*): SAT-based algorithm to find $\max_{S \subseteq V(G)} \min_{u,v \in S, u \neq v} |\triangle_{G[S]}(u, v)|$.
  - LBSample: Sampling-based algorithm. Finds a connected induced subgraph G' of G by random walk and computes the exact or lower-bound twin-width of G'.
  - LBSeparate (*): Similar to LBSample, but uses the hydra decomposition to find an induced subgraph to check for the lower-bound.
- Upper-bound algorithms
  - UBGreedy: Iteratively contract a vertex pair minimizing the weak red potential.
  - UBLocalSearch: Using the timeline encoding, we make small changes to the elimination ordering and the contraction tree to see if there is a better solution.
  - UBSeparate (*): Iterative refinement algorithm using the hydra decomposition.

In this paper we focus on two additional contributions to solving twin-width which are used in the LocalSearch and Separate algorithms implemented in Hydra Prime: "timeline encoding" and "hydra decomposition". *Timeline encoding* is a novel data structure which enables faster computation of twin-width by storing red "sources" and "intervals" indicating the cause and window of each red edge. In the Separate upper- and lower-bound algorithms, we introduce *hydra decomposition*, an iterative refinement strategy using small vertex separators. After defining necessary notation, we briefly describe these in Sections 2 and 3, respectively. Additional details are in the appendix available on the code repository.

Notation We follow standard graph-theoretic notation (e.g. found in [4]), the original definition of twin-width [3], and terminology introduced by Schidler and Szeider [7]. Refer to [5] and [9] for the definitions of a *module*, *modular decomposition*, a *prime* graph, etc. We write $u \leftarrow v$ when vertex v is contracted into vertex u. Given a trigraph G, the *weak red potential* of $u, v \in V(G), u \neq v$ is the red degree of u after contraction $u \leftarrow v$. We further define the *unshared neighbors* of vertices u and v, denoted by $\triangle(u, v)$ as $N(u) \triangle N(v) \setminus \{u, v\}$, where $\triangle$ denotes the symmetric difference of two sets. We write $[n]$ for $\{1, \ldots, n\}$.

2 Timeline Encoding

In this work we developed the *timeline encoding*, a data structure to compute the width of a given contraction sequence. An instance of the timeline encoding stores the following data:

- G: input graph with n vertices.
- $\phi : V(G) \to [n]$: bijection that encodes an elimination ordering (vertex v is eliminated at time $\phi(v)$ if $\phi(v) < n$).
- $p : [n-1] \to [n]$: encoding of a contraction tree. For $i < j$, $p(i) = j$ if vertex $\phi^{-1}(i)$ is merged into vertex $\phi^{-1}(j)$ (i.e. j is the *parent* of i in the contraction tree).

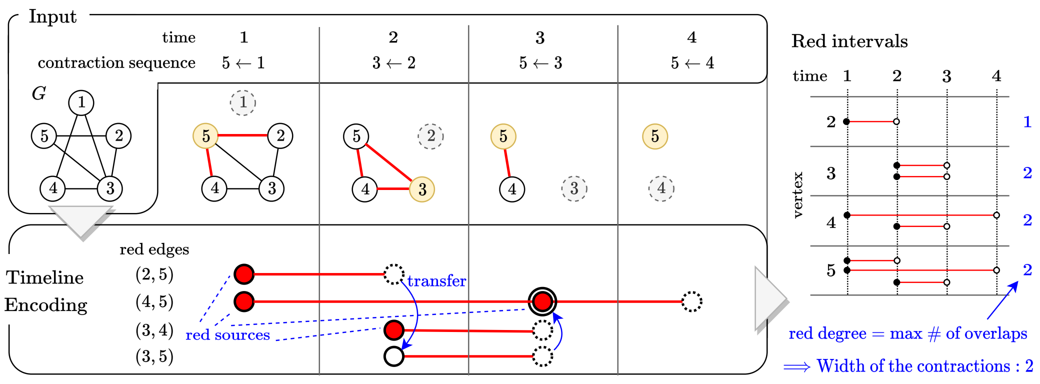
Input
time 1 2 3 4
contraction sequence $5 \leftarrow 1$ $3 \leftarrow 2$ $5 \leftarrow 3$ $5 \leftarrow 4$
G
Timeline Encoding
red edges
(2, 5)
(4, 5)
(3, 4)
(3, 5)
transfer
red sources
Red intervals
vertex
red degree = max # of overlaps
⟹ Width of the contractions : 2

Figure 1 An illustration of the timeline encoding given a graph and its contraction sequence. Vertex labels show elimination ordering. For each time i with contraction $j \leftarrow i$ $(i < j)$, we create red sources $\{k, j\}$ for every $k \in \triangle_>(j, i)$, which determines red intervals $[i, \min\{k, j\})$ that will then disappear or transfer at time $\min\{k, j\}$. The red degree corresponds to the number of overlaps of red intervals aggregated by vertices, and its maximum value is the width of the contraction sequence.

For internal data structures, we introduce a few terms. First, define $\triangle_>(j, i) := \{\phi(w) \mid w \in \triangle(\phi^{-1}(i), \phi^{-1}(j)), \phi(w) > i\}$. Then, the *red sources* at time t are a set of red edges introduced at time t, defined as: $\{\{p(t), k\} \mid k \in \triangle_>(p(t), t)\}$. Red sources determine the *red intervals* – non-overlapping, continuous intervals where an edge is red, defined as follows: for $i < j$, red source (i, j) at time t creates an interval $[t, i)$ (red edge ij disappears at time i). If $p(i) \neq j$, then we recurse this process as if red source $\{p(i), j\}$ was created (red edge ij transfers to $\{p(i), j\}$), as illustrated in Figure 1.

Now we aggregate red intervals by vertices. We maintain a *multiset* of intervals for each vertex such that a red interval of an edge accounts to its both endpoints. The maximum number of the overlaps of such intervals gives the maximum red degree at a vertex over time. Finally, we obtain the width of the contraction sequence by taking the maximum of the red degrees over all vertices.

A key observation is that we can dynamically compute the number of overlaps of a multiset of intervals efficiently with a balanced binary tree (e.g. modification in time $\mathcal{O}(\log n)$, getting the maximum number of overlaps in $\mathcal{O}(1)$, etc.). For local search, we implemented methods for modifying a contraction tree and also updating a bijection ϕ.

3 Hydra Decomposition

We also implemented an iterative refinement strategy which we term *hydra decomposition*, based on finding a small vertex separator. A *hydra* is a structured trigraph which consists of a (possibly empty) set of *heads* and a (possibly empty) vertex set *tail*. Each head is a set of vertices containing one *top vertex* and a nonempty set of *boundary vertices*. The neighbors of the top vertex must be a subset of the boundary vertices. All red edges in the trigraph must be incident to one of the top vertices. Heads must be vertex-disjoint, but the tail may contain boundary vertices (but not a top vertex). A *compact hydra* is a hydra consisting of its tail and one extra vertex, with no heads. A head of a hydra H can additionally be viewed as a compact hydra C, where the boundary vertices of H are the tail of C. Now that we have defined the parts of a hydra, we will now show the operations performed in hydra decomposition:

1. *Separate*: partitions the vertices of a hydra into three parts: S, A, B such that S separates

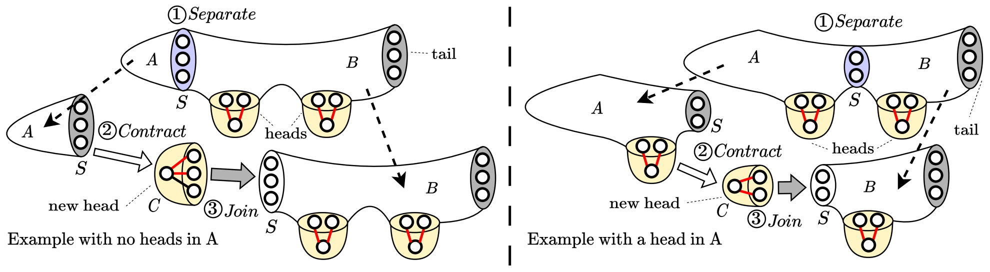

Figure 2 Structure of the hydra and two examples of performing a round of hydra decomposition.

A from B. S should not contain any vertices from the heads, and any tail vertices cannot be in A. Figure 2 shows two ways of choosing a separator S of a hydra.

2. *Contract*: takes a hydra and contracts all vertices but its tail. The output is a contraction sequence and the resulting compact hydra.
3. *Join*: combines a compact hydra C and another hydra H such that $V(C) \cap V(H)$ is the tail of C. The output is the union of C and H, where the heads and tail of H remain and C becomes an additional head.

We now present a description of UBSeparate. Given a graph H and a target width d for a contraction sequence, UBSeparate runs *contract* on the original graph without any heads or tails. The *contract* operation works as follows: If the input H is small enough, or a vertex separator of size at most d is not found, we directly search for a contraction sequence of width at most d for all but tail vertices, which can be done by modified UBGreedy and other exact algorithms. Otherwise, we perform *separate* to obtain a partition S, A, B. We recursively call *contract* with $H[A \cup S]$ with S being the tail. Then, we have a contraction sequence s_1 and a compact hydra C. Next, we *join* C with $H[B \cup S]$ and obtain a hydra H'. Notice that the tail of C must be S. We again call *contract* with H' and get a contraction sequence s_2, resulting in a compact hydra C' with the original tail of H. Finally, C' is returned along with the concatenation of s_1 and s_2 for the result of the original *contract* operation.

A key observation is that since red edges reside only in heads and the size of separators are bounded by d, the red degree of a hydra is also upper-bounded by d, which helps construct a d-contraction sequence part by part. For $d = 1$ we use a linear-time algorithm to find a vertex separator, or a cut vertex (articulation point); for $d \geq 2$, we instead call a SAT solver.

References

1 Jungho Ahn, Kevin Hendrey, Donggyu Kim, and Sang-il Oum. Bounds for the Twin-width of Graphs. *SIAM Journal on Discrete Mathematics*, 36(3):2352–2366, September 2022.

2 Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022.

3 Édouard Bonnet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width I: tractable FO model checking. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 601–612, November 2020.

4 Reinhard Diestel. *Graph Theory*. Springer Publishing Company, Inc., 5th edition, 2017.

5 Michel Habib and Christophe Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4(1):41–59, 2010.

6 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental Cardinality Constraints for MaxSAT. In B. O'Sullivan, editor, *Principles and Practice of Constraint Programming (CP14)*, Lecture Notes in Computer Science, pages 531–548. Springer International Publishing, 2014.

7 André Schidler and Stefan Szeider. A SAT Approach to Twin-Width, 2021. arXiv:2110.06146.

8 Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, CP'05, pages 827–831, Berlin, Heidelberg, 2005. Springer-Verlag.

9 Marc Tedder, Derek Corneil, Michel Habib, and Christophe Paul. Simple, linear-time modular decomposition, March 2008. arXiv:0710.3901.

A Known Theoretical Results

We utilized the following known facts extensively in our implementation.

- If G' is an induced subgraph of a graph G, then $\mathsf{tww}(G') \leq \mathsf{tww}(G)$ [3].
- If $\overline{G}$ is the complement graph of a graph G, then $\mathsf{tww}(G) = \mathsf{tww}(\overline{G})$ [3].
- $\mathsf{tww}(G) = \max_{H \in \mathcal{P}} \mathsf{tww}(H)$, where $\mathcal{P}$ is a set of prime graphs found in the modular decomposition of G [7].
- For a graph G, $\min_{u,v \in V(G), u \neq v} |\triangle(u,v)| \leq \mathsf{tww}(G)$ [7].
- If every component of a graph G has at most one cycle, then $\mathsf{tww}(G) \leq 2$ [1].
- For a tree T, $\mathsf{tww}(T) \leq 1$ if and only if T is a caterpillar[1] [1].

B SAT Formulation

Recent PACE challenges have shown that modern SAT solvers are versatile at solving NP-hard problems; TWIN-WIDTH is not an exception. As part of our solver, we adopted a known SAT encoding by Schidler and Szeider [7] with a few additional clauses which we will detail in Section B.3. We also used novel SAT encodings to find lower bounds (LBCore) as well as to find vertex separators (LBSeparate, UBSeparate). For our SAT encodings, we assumed that a given graph G is prime and $V(G) = [n]$, and we work with binary variables having a value either 0 (false) or 1 (true).

We use the Kissat SAT solver 3.0.0 [2] as an external library. In addition, we implemented the sequential counter [8] for encoding cardinality constraints as it performed better than the iterative totalizer [6] (typically used in MaxSAT[2]) and other algorithms.

B.1 Lower-bound: LBCore

For a graph G, we define the *minimum neighborhood difference* ℓ_G as $\min_{u,v \in V(G), u \neq v} |\triangle_G(u,v)|$ (refer to Section 1 for notation). The algorithm LBCore finds an induced subgraph maximizing the minimum neighborhood difference, using the fact that $\ell_{G'} \leq \mathsf{tww}(G') \leq \mathsf{tww}(G)$ for any induced subgraph G' of G.

To find a vertex set S maximizing $\ell_{G[S]}$, we guess an integer d with $d \geq 1$ and query a SAT solver if there exists a set $S \subseteq V(G)$ such that $\ell_{G[S]} \geq d$. Note that since the given graph is prime, there are no twins in G and we have $\ell_G \geq 1$; there is no need to test for $d = 0$. We observed that SAT solvers quickly find solutions for small graphs.

SAT Variables

- $x(i)$ for $1 \leq i \leq n$: $x(i)$ is true if and only if vertex i is in a solution S.

Clauses

- Semantics of $x(i)$:
  - $\sum_{i=1}^{n} x(i) \geq 4$.
- Semantics of $c(i,j)$: for every $1 \leq i < j \leq n$
  - $x(i) \wedge x(j) \to \sum_{k \in \triangle(i,j)} x(k) \geq d$. [3]

[1] A *caterpillar* is a tree containing a path P such that all other vertices are adjacent to some vertex in P.
[2] A problem to maximize satisfying SAT clauses. We did not use MaxSAT for the solver.
[3] Implication denoted by $x \to y$ is equivalent to $\neg x \vee y$.

Correctness First, suppose that all clauses are satisfied. Then, the solution S has at least 4 vertices and for every vertex pair i, j in S, $|\triangle_{G[S]}(i,j)| \geq d$ since $x(i) \wedge x(j) \to \sum_{k \in \triangle(i,j)} x(k) \geq d$, which leads to $\ell_{G[S]} \geq d$. Conversely, suppose that there is a set S such that $\ell_{G[S]} \geq d \geq 1$. Since S does not contain twins, $|S| \geq 4$, satisfying the first constraint. Also, we know that for every distinct vertex pair i, j in S, $|\triangle_{G[S]}(i,j)| = |\triangle_G(i,j) \cap S| = \sum_{k \in \triangle_G(i,j)} x(k) \geq d$. This satisfies the second constraint.

B.2 Finding vertex separators

To perform the hydra decomposition (used in algorithms LBSeparate and UBSeparate), we need to find a vertex separator under certain constraints. We formulate a problem as follows: given a connected graph G and its hydra structure (defined in Section 3) with head $H \subseteq V(G)$ and tail $T \subseteq V(G)$ vertices, and an integer k, find a vertex separator $S \subseteq V(G) \setminus H$ of size at most k, such that $G - S$ can be partitioned into two parts A, B with the following constraints. (1) A and B must be nonempty, (2) there are no edges between A and B, (3) B is a superset of $T \setminus S$, and (4) at least one vertex from T must be in B.

When $k = 1$, we can solve this problem in linear-time by testing all articulation points (or cut vertices) in G. Otherwise, we use color coding on $V(G)$ indicating the guess of a partition A, B, S with colors $\dot{A}, \dot{B}, \dot{S}$: $\chi : V(G) \to \{\dot{\mathsf{A}}, \dot{\mathsf{B}}, \dot{\mathsf{S}}\}$. If all vertices are properly colored, we have $A = \chi^{-1}(\dot{\mathsf{A}}), B = \chi^{-1}(\dot{\mathsf{B}}), S = \chi^{-1}(\dot{\mathsf{S}})$.

SAT Variables

- $x(i)$ for $i \in V(G)$: $x(i) = 1$ if and only if vertex i has color $\dot{\mathsf{A}}$.
- $y(i)$ for $i \in V(G)$: $y(i) = 1$ if and only if vertex i has color $\dot{\mathsf{B}}$.
- $z(i)$ for $i \in V(G)$: $z(i) = 1$ if and only if vertex i has color $\dot{\mathsf{S}}$.

Clauses

- Every vertex has a color: for every $i \in V(G)$
  - $x(i) + y(i) + z(i) = 1$.
- Semantics of the heads:
  - Head vertex cannot be in solution: for every $i \in H$, $\neg z(i)$.
- Semantics of the tail:
  - Tail vertex cannot be in part A: for every $i \in T$, $\neg x(i)$
  - At least one tail vertex is not in solution: $\bigvee_{i \in T} y(i)$
- Encoding reachability: for every edge $ij \in E(G)$
  - $x(i) \to \neg y(j)$.
  - $y(i) \to \neg x(j)$.
- Parts A and B must be nonempty.
  - $\sum_{i \in V(G)} x(i) \geq 1$.
  - $\sum_{i \in V(G)} y(i) \geq 1$.
- Constraints on the solution size.
  - $\sum_{i \in V(G)} z(i) \leq k$.

Correctness Notice that each SAT clause encodes all constraints in the problem definition. Let $A = x^{-1}(1)$, $B = y^{-1}(1)$, and $S = z^{-1}(1)$. Then, from the reachability constraints, there cannot be an edge between A and B, as if there is an edge ij such that $i \in A, j \in B$, $x(i) = y(j) = 1$, which violates $x(i) \to \neg y(j)$. Also, since $\neg x(i)$ for every $i \in T$, we have $T \cap A = \emptyset$. Then, $T \subseteq B \cup S$ and $T \setminus S \subseteq B$. For others, it is straightforward to see relations between the problem definition and SAT clauses.

B.3 Direct Solver

Our SAT formulation for computing exact twin-width is based on the relative encoding presented in [7], which is faster than the absolute encoding in most instances. We further add extra hints for vertex pairs having many unshared neighbors. Specifically, for a given target twin-width d and any integer $s > 0$, if vertices $u, v \in V(G)$ has $d + s$ unshared neighbors, then contraction $u \leftarrow v$ cannot appear in the first s contractions. Otherwise, the red degree of u after the contraction will exceed d. Even stronger, we can state that at least s of such unshared neighbors must be contracted before contraction $u \leftarrow v$. With these hints, some instances witnessed speed-ups in a SAT solver's running time.

SAT Variables

- $o(i,j)$ for $1 \leq i < j \leq n$: encodes an elimination ordering (relative encoding); $o(i,j) = 1$ if and only if vertex i is earlier than vertex j in the elimination ordering.
- $p(i,j)$ for $1 \leq i < j \leq n$: encodes a contraction tree; $p(i,j) = 1$ if vertex i is merged into vertex j.
- $r(i,j,k)$ for $1 \leq i,j \leq n$ and $j < k \leq n$: encodes red edges; $r(i,j,k) = 1$ if after eliminating vertex i, there is a red edge jk.
- $a(i,j)$ for $1 \leq i < j \leq n$: auxiliary variable for computing transferred red edges; $a(i,j) = 1$ if a red edge ij is present at any time.

For any distinct $1 \leq i,j \leq n$, we use shorthand notation $o^*(i,j)$ for $o(i,j)$ if $i < j$ and $\neg o(i,j)$ if $i > j$. Also, let $a^*(i,j) = a(\min\{i,j\}, \max\{i,j\})$ and similarly $r^*(i,j,k) = r(i, \min\{j,k\}, \max\{j,k\})$.

Clauses

- Semantics of o:
  - transitivity: for every mutually distinct $1 \leq i,j,k \leq n$, $o^*(i,j) \wedge o^*(j,k) \rightarrow o^*(i,k)$.
- Semantics of p:
  - all but the root vertices must have one parent: for every $1 \leq i < n$, $\sum_{j=i+1}^{n} p(i,j) = 1$.
  - parent must be present: for every $1 \leq i < j \leq n$, $p(i,j) \rightarrow o(i,j)$.
- Semantics of r:
  - encodes red edges: for every $1 \leq i < j \leq n$ and $k \in \triangle(i,j)$, $p(i,j) \wedge o^*(i,k) \rightarrow r^*(i,j,k)$.
  - additional hints described above: for every $1 \leq i < j \leq n$ and $1 \leq s \leq d$, $p(i,j) \rightarrow \sum_{1 \leq k \leq n : i,j \neq k} o^*(k,i) \geq s$.
  - transfer red edges: for every mutually distinct $1 \leq i,k \leq n$ and $i < j \leq n$, $p(i,j) \wedge o^*(i,k) \wedge a^*(i,k) \rightarrow r^*(i,j,k)$.
  - maintain red edges: for every mutually distinct $1 \leq i,j \leq n$ and $1 \leq k < m \leq n$, $o^*(i,j) \wedge o^*(j,k) \wedge o^*(j,m) \wedge r^*(i,k,m) \rightarrow r^*(j,k,m)$.
- Semantics of a:
  - for every mutually distinct $1 \leq i,j,k \leq n$, $o^*(k,i) \wedge o^*(k,j) \wedge r^*(k,i,j) \rightarrow a^*(i,j)$.
- Constraints on the solution size
  - for every distinct $1 \leq i,x \leq n$, $\sum_{1 \leq y \leq n : i,x \neq y} r(i,x,y) \leq d$.

Correctness The main proof is in [7]. Additional hints do not affect the fidelity of the encoding.