PACE Solver Description: RedAlert - Heuristic Track

Édouard Bonnet

Univ Lyon, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP UMR5668, France

Julien Duron

Univ Lyon, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP UMR5668, France

Abstract

We present `RedAlert`, a heuristic solver for twin-width, submitted to the Heuristic Track of the 2023 edition of the Parameterized Algorithms and Computational Experiments (PACE) challenge.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases twin-width, contraction sequences, heuristic, pair sampling, pair filtering

Related Version Clone the sources at `https://bitbucket.org/tennobe/red-alert` or download the code at `10.5281/zenodo.8079499`.

Funding This work was supported by the ANR projects TWIN-WIDTH (ANR-21-CE48-0014) and Digraphs (ANR-19-CE48-0013).

1 Twin-width and contraction sequences

To keep our description short, we refer the reader to the first two sections of [2] for the definitions and motivations behind contraction sequences and twin-width. A *trigraph* has two disjoint edge relations: red edges and black edges. Its *total graph* consists of the union of these two relations. The *red degree* (resp. *total degree*) is the degree in the graph formed by the red edges (resp. in the *total graph*). We aim to find a contraction sequence (that iteratively identifies two vertices and updates the color of their incident edge, until there is only one vertex left) with overall maximum red degree as low as possible.

2 Overview of RedAlert

In the search of contraction sequences of low width, a natural subroutine consists of finding a *good pair of vertices*, that is, one whose contraction results in a trigraph with maximum red degree as low as possible. An oracle providing the greedily best pair in 10^{-5}s would have likely won the competition. However, this is far from what is physically possible. In theory, getting a best pair within the allowed 300s is already challenging, since the largest instances had order of 10^7 vertices, and Closest Vector Pair –more or less the *best pair* problem when starting from a graph– (like the task of finding an orthogonal pair of 0,1-vectors) has no truly subquadratic algorithm unless the Strong Exponential-Time Hypothesis fails [3].

We thus resolve to sampling the pairs of vertices as candidates for the next contraction. Based on the remaining time and number of vertices, we compute an integer `minPairs` indicating the minimum number of pairs to be contracted from the sampled pairs (to finish in time). We then contract at least `minPairs` candidate pairs according to a cost function detailed below. Challenging instances produce denser and denser trigraphs as we contract them. This results in a progressive increase of the time to sample and select pairs to contract. In such cases, we may resort to faster (and rougher) subroutines to finish the contraction sequences: `totDegAlert` and `balancedScheme`. We will briefly detail them.

To summarize, the main loop of `RedAlert` is as follows:

1. Estimation of `minPairs`;
2. Sampling of the *candidate pairs*;
3. Selection of at least `minPairs` most favorable pairs among the candidate pairs;
4. Contraction of these pairs;
5. If running out of time, finish with increasingly cruder heuristics.

Also see Figure 1.

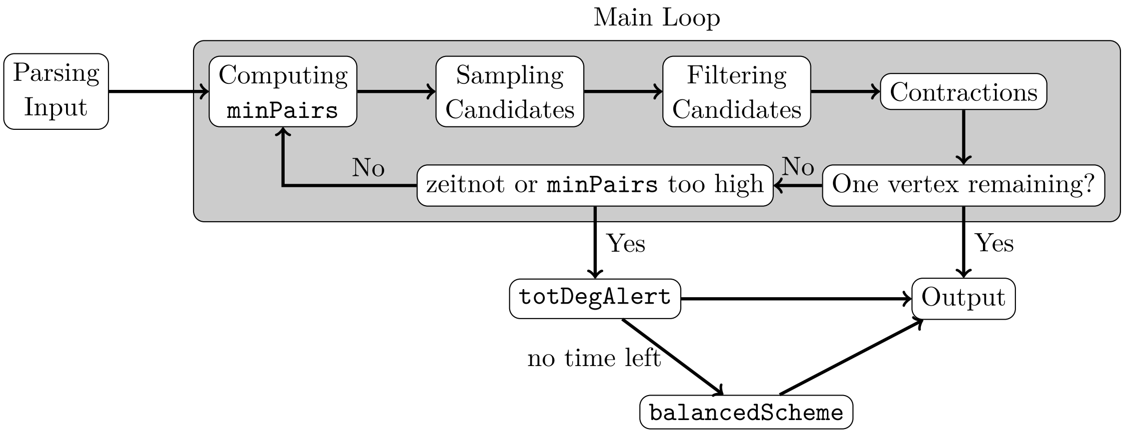

Figure 1 Sketch of `RedAlert`. We exit the main loop when (we have a solution or) the time budget of 260s is depleted or we would have to contract over 40% of our sampled pairs. We then call `totDegAlert` up to second 285. If by then no complete contraction sequence was found, we call `balancedScheme` which takes less than a second to terminate.

3 Dense and Small vs. Sparse and Large Inputs

Before we start parsing the input graph, we decide based on its number of vertices and edges whether we want to work with adjacency matrices or adjacency lists. Typically the former shall be preferred on graphs with few vertices but high edge density, while it is simply not an option when the number of vertices becomes too large. Only the first three or four instances (below 2500 vertices) of the Heuristic Track were such that our treatment with adjacency matrices performed better. Thus we will mainly describe the part of our algorithm using adjacency lists. Nevertheless, let us mention one nice feature of using adjacency matrices: one can test the *quality* of a pair of vertices by sampling an appropriate number of indices, and computing the number of disagreements (potential red edges) at these indices.

In the *sparse* case, we represent our trigraphs with slightly-modified adjacency lists: the neighbors of a vertex are stored in a set. Each vertex has a set for its black neighbors, and a set for its red neighbors. While contracting the trigraph, we maintain other useful elements such as the remaining number of vertices, edges, maximum red degree, overall maximum red degree, list of pairs *vertex/red degree*, list of pairs *vertex/total degree*, and some arrays to keep the conversion between *local* labels and *global* labels. The contraction operation remains reasonably fast and takes a typical 10^{-4}s on the large instances.

4 Sampling candidates

In the *sparse* case, our distribution is biased towards pairs of vertices that are close to each other. We pick a vertex v uniformly at random. Then with probability $1/2$, we uniformly pick a first neighbor of v (in the total graph) to complete the pair, and with probability $1/2$, we uniformly choose a second neighbor of v (still in the total graph). This performs well on the sparsest instances. As half of our sampled pairs consist of vertices at distance 2, we naturally find contractions that are decreasing the red degree of high-degree vertices.

We experimented a bit with favoring vertices v with low red degree or low total degree, or adding pairs of red neighbors of a vertex with highest red degree. This did not seem to improve the overall performance of the heuristic, so we opted for this simple distribution.

We adjust the sampling size to the situation by increasing it while `minPairs` is below 30, decreasing it if later `minPairs` gets above 70, and resetting it to its minimum of 10000 when `minPairs` reaches 500.

5 Filtering candidates

We evaluate the sampled pairs based on a cost function f, defined as follows. If G is the current trigraph, u, v two vertices of G, and G' the resulting trigraph if u and v were contracted into w, we set $f(u, v) = (r, p, e)$ where

- $r \in \{0, 1\}$, and $r = 0$ iff the maximum red degree of G' is smaller than that of G;
- p is the maximum red degree among vertices of G' in the closed red neighborhood of w;
- e is the total number of red edges in G'.

When comparing two pairs of vertices, we prefer the one whose image by f is lexicographically smaller. We select a minimum of `minPairs` pairs, and every pair tying with the worst such pair is also added (their cost is recomputed in the current trigraph). This way, we can sometimes contract a lot more pairs than necessary, and spare some time for an uncertain and more complicated future.

6 When time gets shorter or `minPairs` gets too large

It can happen that `minPairs`, which is computed based on the number of remaining time and vertices (or contractions), and the time spent in the previous loop iteration per performed contraction, steadily increases. This typically happens when the *densification of the current trigraph accelerates*. This may result in a situation when, to meet its deadline, the heuristic has to contract a large fraction of the sampled pairs, making it close to the random heuristic.

In that case, we break out of the main loop and call a faster subroutine, `totDegAlert`, which greedily contracts pairs of smallest total degree. This subroutine still requires to explicitly perform contractions, which takes some time on the largest instances. It is thus possible that we run out of time even inside `totDegAlert`. Therefore, when we have only 15 seconds left, we call `balancedScheme`. This subroutine is based on the $O(\sqrt{m})$-sequence for m-edge graphs (see [1] for a more precise bound). It partitions the vertex set into $O(\sqrt{m})$ *buckets* of roughly equal sum of total degrees, plus an additional bucket with vertices of total degree $\Omega(\sqrt{m})$ (large degree). The idea is then to contract every bucket into a single vertex, finishing with the bucket of large-degree vertices, and end the contraction sequence arbitrarily. What makes `balancedScheme` particularly fast is that we do not need to make these contractions explicitly. As a simple but effective optimization, we contract each bucket in such a way that the *contraction tree* is a balanced binary tree rather than a caterpillar.

When `RedAlert` is about to output a solution for which it knows the actual width (i.e., without invoking `balancedScheme`), we first compare it to some small multiple of $\sqrt{m}$ where m is the number of edges of the input graph. In some cases, indeed, running `balancedScheme` from scratch on the original graph gives a better contraction sequence.

References

1 Jungho Ahn, Kevin Hendrey, Donggyu Kim, and Sang-il Oum. Bounds for the twin-width of graphs. *SIAM J. Discret. Math.*, 36(3):2352–2366, 2022. doi:10.1137/21m1452834.

2 Édouard Bonnet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width I: tractable FO model checking. *J. ACM*, 69(1):3:1–3:46, 2022. doi:10.1145/3486655.

3 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.