Detecting Almost Symmetries of Graphs

Ben Knueven and Jim Ostrowski
Department of Industrial and Systems Engineering
University of Tennessee, Knoxville, TN
bknueven@vols.utk.edu jostrows@utk.edu

Sebastian Pokutta
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology, Atlanta, GA
sebastian.pokutta@isye.gatech.edu

August 4, 2017

Abstract

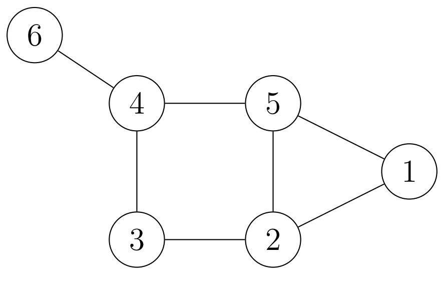
We present a branch-and-bound framework to solve the following problem: Given a graph G and an integer k, find a subgraph of G formed by removing no more than k edges that minimizes the number of vertex orbits. We call the symmetries on such a subgraph "almost symmetries" of G. We implement our branch-and-bound framework in PEBBL to allow for parallel enumeration and demonstrate good scaling up to 16 cores. We show that the presented branching strategy is much better than a random branching strategy on the tested graphs. Finally, we consider the presented strategy as a heuristic for quickly finding almost symmetries of a graph G.

1 Introduction

Two graphs are *isomorphic* if there is a bijection between their vertices that preserves adjacency; such a bijection is called an *isomorphism.* An isomorphism from a graph onto itself is called an *automorphism*, and the set of all automorphisms of a given graph G, denoted $\text{Aut}(G)$, forms a group under composition.

The Graph-Isomorphism problem is that of determining the existence (or not) of an isomorphism between two input graphs. It is a notorious problem in complexity theory, as no polynomial time algorithm is known, and at the same time Graph-Isomorphism is generally not believed to be NP-complete [13]. Babai recently presented a quasi-polynomial time algorithm for Graph-Isomorphism [2]. It is well-known that Graph-Isomorphism and the problem of determining the orbits of $\text{Aut}(G)$ (Automorphism-Partition) are polynomial time equivalent [27, 20].

Figure 1: A graph with almost symmetry.



Recent results have demonstrated the hardness of various robust or approximate versions of Graph-Isomorphism [16, 1, 25]. All these are either NP-hard [16, 1] or believed to be NP-hard [25]. However, there has been little study of the computational feasibility of such problems. In this paper we propose and implement a branch-and-bound algorithm for solving a robust version of Automorphism-Partition.

1.1 Preliminaries

We need to first lay out some notation that we will use throughout. Given an undirected graph $G = (V, E)$ (later we always write $G(V, E)$), for some set of edges $F \subset E$, we define the graph $(G - F) := (V, E \setminus F)$, that is, $(G - F)$ is the graph G with the edges in F removed. We may also use the notion $V(G)$ to refer to the vertices of the graph G and $E(G)$ to refer to the edges of G.

Definition 1. For an n-vertex graph G, a permutation $\pi\colon V(G) \to V(G)$ is an automorphism of G if for every $\{u, v\} \in E(G)$, $\{\pi(u), \pi(v)\} \in E(G)$ and for every $\{u, v\} \notin E(G)$, $\{\pi(u), \pi(v)\} \notin E(G)$.

Definition 2. For a graph $G(V, E)$, the mapping $\sigma : V(G) \to V(G)$ is a *k-almost symmetry* of G if there exists a set of edges $E_D \subseteq E$ with $|E_D| \leq k$ such that σ is an automorphism for the graph $(G - E_D)$. We denote the set of k-almost symmetries of G as $\mathcal{AS}_k(G)$.

The 0-almost symmetries are exactly the automorphisms of G. We have the following motivating result courtesy of [7]; an examination of the proof of Theorem 1 in [7] shows that only edge deletions are used.

Theorem 1. *For an n-vertex graph G and $k \geq \lfloor \frac{n-1}{2} \rfloor$, it holds $|\mathcal{AS}_k(G)| > 1$, that is, there exists a non-trivial k-almost symmetry.*

Example 1. Consider the graph shown in Figure 1 from [11]. The permutation (25)(34) is a 1-almost symmetry of the graph because it is an automorphism of $G - \{\{4, 6\}\}$. The permutations (16)(24) and (35) are also 1-almost symmetries, as seen by removing edge $\{1, 5\}$.

From this example we also obtain the following remark.

Remark 1. For $k \geq 1$, the set of k-almost symmetries is not necessarily a group.

To this end, consider the permutation $(35) \circ (25)(34) = (2345)$. Enumerating all 7 possibilities shows that (2345) is not a 1-almost symmetry of the graph. Since there is no group structure to be exploited during the search for almost symmetries, intuitively the problem of finding almost symmetries seems much harder than that of finding symmetries. Indeed, one of the main contributions of McKay's venerable `nauty` program [21] for graph isomorphism and automorphism is its ability to prune symmetric regions of the search space *as* symmetries on the graph are detected. However, without a group structure the underlying theory developed by McKay falls apart.

It will be helpful to tie our definition of k-almost symmetry to the notion of α-automorphism from [25]. First consider α-isomorphism between two graphs G and H:

Definition 3. For non-empty n-vertex graphs G and H, and $0 \leq \alpha \leq 1$, a permutation $\pi \colon V(G) \to V(H)$ is an α-isomorphism if

$$\frac{|\{\{u,v\} \in E(G) : \{\pi(u), \pi(v)\} \in E(H)\}|}{\max\{|E(G)|, |E(H)|\}} \geq \alpha. \qquad (\alpha\text{-ISO})$$

In this definition, α is a lower bound on the ratio of edges that get mapped to edges. O'Donnell et al. [25] proved, assuming Feige's R3XOR hypothesis [8], that given two $(1-\epsilon)$-isomorphic graphs, finding a $(1-r(\epsilon))$-isomorphism is NP-hard (for some function $r(\epsilon) \to 0$ as $\epsilon \to 0^+$). Arvind et al. [1] have shown that finding a permutation π which maximizes α is NP-hard. An *α-automorphism* is an α-isomorphism from a graph to itself:

Definition 4. For a non-empty n-vertex graph G, and $0 \leq \alpha \leq 1$, a permutation $\pi \colon V(G) \to V(G)$ is an α-automorphism of G if

$$\frac{|\{\{u,v\} \in E(G) : \{\pi(u), \pi(v)\} \in E(G)\}|}{|E(G)|} \geq \alpha. \qquad (\alpha\text{-AUT})$$

While it is beyond the scope of this paper to address the complexity question of k-almost symmetries, we do note that the k-almost symmetries of G are $\left(1 - \frac{k}{|E(G)|}\right)$-automorphisms of G, since at most k edges get mapped to non-edges under a k-almost symmetry. Similarly any α-automorphism of G is a $\lfloor (1-\alpha)|E| \rfloor$-almost symmetry of G.

1.2 Contribution

We present an algorithm capable of finding the k-almost symmetries of a graph G. However, we will be more interested in the following related problem for a given graph $G(V,E)$ and budget k:

$$\gamma_k^G = \min\{|\mathrm{orb}_{\mathrm{Aut}(G')}(V(G'))| : G' = (G - E_D), E_D \subseteq E, |E_D| \leq k\}, \qquad (1)$$

where $\mathrm{orb}_\Gamma(X)$ for some set X and group Γ is the orbital or automorphism partition of X under Γ's action on X, and $|\mathrm{orb}_\Gamma(X)|$ is the number of orbits. Notice in this framework $\mathrm{Aut}(G')$ will be a subset of the k-almost symmetries on G; further it will be a subset with a group structure. Put another way problem (1) is that of finding the subgraph G by removing at most k edges in such a way that the number of orbits is minimized. It is in this sense that we provide an algorithm for a generalized version of Automorphism-Partition. We present a branch-and-bound algorithm for solving (1) with a branching strategy that we show is much better than random, and is in fact by one measure often the best branching choice available. Additionally we show that with some modifications the branch-and-bound strategy presented can be used as an effective heuristic, and we demonstrate the robustness of our branching strategy.

1.3 Motivation

In integer programming (IP), it is well known that the presence of symmetry, if not properly addressed, can confound the branch-and-bound process. The authors hypothesize that a large amount of almost symmetry can have a similar effect by causing "almost symmetric" regions of the search space to be considered. While we will not address the question of almost symmetry in IP directly, it is worth noting that all the methods in the literature for dynamic symmetry breaking in IP rely on insights from symmetry detection on graphs [17, 18, 26]. We restrict ourselves to edge-deletions because they have a natural IP corollary. Suppose we have the following IP:

$$\max_{x \in \{0,1\}^n} \left\{ \mathbb{1}^T x \mid Ax \leq \mathbb{1} \right\}, \tag{2}$$

where $A \in \{0,1\}^{m \times n}$ and $\mathbb{1}$ is the appropriately-sized vector of 1's. Then there is a one-to-one correspondence between the symmetries of the formulation of (2) and the bipartite graph $G(A) = (N, M, E)$ where the partite set N represents the columns of A and the other partite set M represents the rows of A, and an edge is between a vertex $i \in N$ and $j \in M$ if and only if $a_{ij} = 1$ (see [26] for more details). An edge deletion in $G(A)$ therefore represents changing a 1 to a 0 in the constraint matrix A, leading to a relaxation of (2). Similarly an edge addition would result in a restriction of (2). Our method presented here also works for edge additions by simply considering the complement of G. Therefore, a natural starting point for the study of almost symmetry in IP is the detection of almost symmetries on graphs.

1.4 Literature Review

There have been several efforts for detecting near, fuzzy, or almost symmetries in graphs. Buchheim and Jünger present an integer programming approach in [3] which allows for the possibility of edge deletions and additions. They consider rotational and reflective symmetries separately and attempt to find a rotational or reflective symmetry that minimizes the number of edge modifications. Computational results are presented on graphs with no more than two dozen vertices. With advancements in MIP solvers since publication their method may be applicable to larger graphs today.

Markov [19] considers almost-symmetries on colored graphs, where some "chameleon" vertices are allowed to be mapped to vertices of any color. They extend the common graph automorphism algorithm used by `nauty` [22] and `saucy` [5] to consider these chameleon vertices. However, no computational results are presented.

Fox, Long, and Porteous provide a heuristic method for finding near symmetries under edge contractions [10]. They modify color refinement to look for possible edge contractions as `nauty` individualizes vertices. Additionally, their heuristic looks to minimize the number of fixed vertices under some near symmetry group, with online heuristic detection for when a vertex may be axial under a reflective symmetry. The heuristic is implemented and shown effective for graphs with a hundred vertices and edges and no more than 5 flaws.

The remainder of this paper is outlined as follows. In Section 2 we discuss the details of the branch-and-bound framework for detecting almost symmetries in graphs. Section 3 gives some implementation details. Accompanying computational results are in Section 4, along with some natural extensions. Finally we draw some conclusions in Section 5.

2 Algorithmic Overview

We will assume throughout that we are solving the problem of finding almost symmetries on an n-vertex graph $G(V, E)$ for a budget k of edge deletions. We maintain two sets throughout:

- A set E^D of deleted edges;
- A set E^F of fixed edges.

We construct a search tree $\mathcal{T}$, where each node $A \in \mathcal{T}$ is uniquely represented as a tuple (E^D_A, E^F_A). Disjunctions are created by taking an edge $e \in E(G) \setminus (E^F_A \cup E^D_A)$ and in one branch adding e to E^D_A, and in the other adding e to E^F_A. We reach a leaf whenever $|E^D_A| = k$ or $E^F_A = E(G) \setminus E^D_A$. Let $G_A = G - E^D_A$ for some node A. We see then, if we completely expand the tree $\mathcal{T}$ to its leaves, $\mathcal{AS}_k(G) = \bigcup_{A \in \mathcal{T}} \text{Aut}(G_A)$. We say A' is a *child* of A if $E^D_A \subseteq E^D_{A'}$ and $E^F_A \subseteq E^F_{A'}$. While the tuple (E^D_A, E^F_A) completely describes the node A of the tree, for convenience we will describe a node as the 4-tuple (G_A, P_A, E^F_A, k_A), where $k_A = k - |E^D_A|$ (the residual budget), and P_A is an n-vertex graph. P_A encodes those pairwise permutations or mappings that have not been proved impossible in A's children. Specifically for an edge $\{i, j\} \in P_A$ and some child A' of A, a permutation mapping vertex i to vertex j in $V(G)$ *may* exist in $\text{Aut}(G_{A'})$.

It is worth noting that the lack of group structure (Remark 1) necessitates storing possible k-almost symmetries as an n-vertex graph as opposed to a vertex partition. In particular, for vertices u, v, w, the existence of k-almost symmetries σ, π such that $\sigma(u) = v$ and $\pi(v) = w$ does not guarantee the existence of a k-almost symmetry mapping u to w. Notice, however, that each k-almost symmetry is in *some* group, and hence has an inverse which will also be a k-almost symmetry, allowing us to use an undirected graph. At the root node R we initialize $R = (\emptyset, \emptyset)$ and P_R to be the complete graph with self-loops, representing that no edges have been deleted, no edges have been fixed, and we have not yet eliminated any permutations as not being k-almost symmetries, respectively. We note also that P_A allows us to write down an additional stopping condition: If at some node A all that is left in P_A are the self-loops, then no additional k-almost symmetries can be found.

Our strategy for controlling the size of the tree is essentially this: at each node A we check necessary conditions for a k_A-almost symmetry mapping i to j for each $\{i, j\} \in E(P_A)$. Any such $\{i, j\}$ that does not satisfy the given necessary conditions is removed. However, since we will only test local consistency, the remaining edges in P_A need not represent actual k_A-almost symmetries of the graph G_A. As laid out in Section 2.3, P_A will also provide us a lower bound on the number of possible orbits in any of

Table 1: DegreeDiffElim for graph in Figure 1 at root node.

vertex v	$d_{G_R}(v)$	$P_R(v)$
1	2	{1, 2, 3, 4, 5, 6}
2	3	{1, 2, 3, 4, 5, ~~6~~}
3	2	{1, 2, 3, 4, 5, 6}
4	3	{1, 2, 3, 4, 5, ~~6~~}
5	3	{1, 2, 3, 4, 5, ~~6~~}
6	1	{1, ~~2~~, 3, ~~4~~, ~~5~~, 6}

A's children. Finally, sufficiency is captured by `nauty`, which at node A computes the automorphisms on the graph G_A, and hence the k-almost symmetries at this node.

2.1 Eliminating mappings

We present three approaches for proving that two vertices of G are not k-almost symmetric. The first two are an extension of simple vertex invariants to the k-almost symmetry case, and the third makes use of the fact that neighbors must be mapped to neighbors, and dominates the other two, at an additional computational cost. For the discussion of all three, suppose we are at some node in the tree denoted by (G_A, P_A, E_A^F, k_A), as described above.

The first approach, listed in Algorithm 1, is based on the following simple fact.

Fact 1. If two vertices' degrees differ by more than k_A, then they are not symmetric in any of node A's children.

If the graph has a good amount of irregularity then Algorithm 1 will be rather effective. Also note that as edges are deleted and k_A is decreased this becomes more powerful, and hence this is run at the root node and after every edge deletion.

Algorithm 1 (DegreeDiffElim) Eliminates mappings between vertices whose degree difference is more than k_A.

```
procedure DegreeDiffElim($G_A$, $P_A$, $k_A$)
    for all $\{i, j\} \in E(P_A)$ do
        if $|d_{G_A}(i) - d_{G_A}(j)| > k_A$ then
            remove edge $\{i, j\}$ from $E(P_A)$
```

Example 2. Consider the graph from Example 1, and suppose $k = 1$ and we are at the root node R, so P_R is the complete graph with self-loops. See Table 1 for how Algorithm 1 updates P_R.

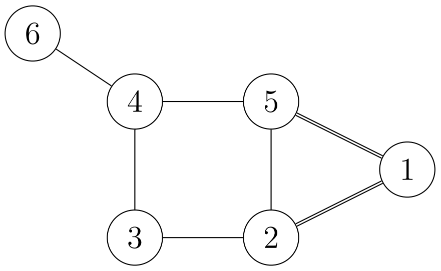
The second approach, listed in Algorithm 2 is based on another simple observation.

Fact 2. For $i \in V$, define $d_{E_A^F}(i)$ to be the number of fixed edges incident to i. For $i, j \in V$, if $d_{E_A^F}(i) > d_{G_A}(j)$, then i and j are not symmetric in any of node A's children.

Algorithm 2 (FixedDegElim) Eliminates mappings between vertices where one's fixed degree exceeds the other's degree.

```
procedure FixedDegElim($G_A$, $P_A$, $E_A^F$)
    for all $\{i, j\} \in E(P_A)$ do
        if $d_{E_A^F}(i) > d_{G_A}(j)$ then
            remove edge $\{i, j\}$ from $E(P_A)$
```

Figure 2: Edges (1,2) and (1,5) are fixed.



Since i has more fixed neighbors than j has neighbors, this is just a restatement of the fact that vertices of different degrees cannot be symmetric. While this is useless at the root node, it becomes more and more powerful as edges are fixed, and so is run after every edge fixing.

Example 3. Picking up where Example 2 left off, suppose we are at the node A, where $E_A^D = \emptyset$ and $E_A^F = \{(1, 2), (1, 5)\}$, represented by the graph in Figure 2. In this case since $d_{E_A^F}(1) = 2$ and $d_{G_A}(6) = 1$, Algorithm 2 allows us to rule out a mapping between vertex 1 and vertex 6 and updates P_A accordingly.

The last, and most powerful method for testing the feasibility of a mapping comes from the observation below.

Fact 3. For graph G, if a permutation $\pi \in \mathrm{Aut}(G)$ maps vertex i to vertex j, it must map neighbors of i to neighbors of j.

Its extension to the almost-symmetry case is outlined in Algorithms 3 and 4. We use these to check the feasibility of such a mapping with k_A edge deletions.

Let us begin with Algorithm 3. We construct a bipartite graph as follows. First, create two partite sets, the left one being $N_{G_A}(i) \setminus \{j\}$ and the right one being $N_{G_A}(j) \setminus \{i\}$, and label their associated vertices u_i for $u \in N_{G_A}(i) \setminus \{j\}$ and v_j for $v \in N_{G_A}(j) \setminus \{i\}$ to distinguish them from the vertices in G_A. (We leave j and i out of these partite sets since a mapping from $i \to j$ implies a mapping from $j \to i$.) Add to the left partite set $k_A + \max\{d_{G_A}(j) - d_{G_A}(i), 0\}$ vertices labeled $\times_j$, and to the right partite set add $k_A + \max\{d_{G_A}(i) - d_{G_A}(j), 0\}$ vertices labeled $\times_i$, so as to create two equal-sized partite sets. The weight of the edge between u_i and v_j is the minimum number of edges that need to be deleted so that $u \to v$. If we have already determined $u \nrightarrow v$, we set this to $+\infty$ to represent that the matching cannot happen (it suffices to set it to $(2k_A + 1)$). It is important to note here that in actuality this weight is only considered to be half the degree difference between u and v (since everything else is multiplied by 2 – which is done so all

Algorithm 3 (BuildCostMatrix) Creates the bipartite graph for testing the map between neighbors.

```
function BuildCostMatrix($i$, $j$, $G_A$, $P_A$, $E_A^F$, $k_A$)
    for all $u \in N(i) \setminus \{j\}$ do
        Create vertex $u_i$
    for all $v \in N(j) \setminus \{i\}$ do
        Create vertex $v_j$
    if $d_{G_A}(i) < d_{G_A}(j)$ then
        Exchange $i$ and $j$
    $degDiff \leftarrow d_{G_A}(i) - d_{G_A}(j)$
    Create $k_A + degDiff$ copies of vertices $\times_i$ and $k_A$ copies of $\times_j$
    for all $u \in N_{G_A}(i) \setminus \{j\}$, $v \in N_{G_A}(j) \setminus \{i\}$ do
        Draw an edge between $u_i$ and $v_j$,
        if $\{u, v\} \in P_A$ then
            if $d_{G_A}(u) > d_{G_A}(v)$ and $u$ independent of $(N(i) \setminus \{j\}) \cup (N(j) \setminus \{i\})$ then
                $w_{u_i,v_j} \leftarrow 2(d_{G_A}(u) - d_{G_A}(v))$                ▷ Cost of mapping $u \rightarrow v$
            else if $d_{G_A}(v) > d_{G_A}(u)$ and $v$ indep. of $(N(i) \setminus \{j\}) \cup (N(j) \setminus \{i\})$ then
                $w_{u_i,v_j} \leftarrow 2(d_{G_A}(v) - d_{G_A}(u))$
            else
                $w_{u_i,v_j} \leftarrow |d_{G_A}(u) - d_{G_A}(v)|$
        else
            $w_{u_i,v_j} \leftarrow +\infty$                ▷ We have already determined $u \nrightarrow v$
    for all $u \in N_{G_A}(i) \setminus \{j\}$, $\times_i$ do
        Draw an edge between $u_i$ and $\times_i$,
        if $\{i, u\} \in E_A^F$ then
            $w_{u_i,\times_i} \leftarrow +\infty$                ▷ Edge $\{i, u\} \in E(G_A)$ cannot be deleted
        else
            $w_{u_i,\times_i} \leftarrow 2$                ▷ Edge $\{i, u\} \in E(G_A)$ can be deleted
    for all $v \in N_{G_A}(j) \setminus \{i\}$, $\times_j$ do
        Draw an edge between $v_j$ and $\times_j$,
        if $\{j, v\} \in E_A^F$ then
            $w_{v_j,\times_j} \leftarrow +\infty$                ▷ Edge $\{j, v\} \in E(G_A)$ cannot be deleted
        else
            $w_{v_j,\times_j} \leftarrow 2$                ▷ Edge $\{j, v\} \in E(G_A)$ can be deleted
    for all $\times_i$, $\times_j$ do
        Draw an edge between $\times_i$ and $\times_j$ with $w_{\times_i,\times_j} \leftarrow 0$.
    return the constructed graph as an assignment matrix
```

Algorithm 4 (RefineByMatching) Eliminates mappings by attempting to map neighbors to neighbors.

```
function RefineByMatching($G_A$, $P_A$, $E^F_A$, $k_A$)
    for all $e \in E(G_A)$ do
        $edgeUse(e) = 0$
    for all $\{i,j\} \in E(P_A)$ do
        $CostMatrix \leftarrow$ BuildCostMatrix($i$, $j$, $G_A$, $P_A$, $E^F_A$, $k_A$)
        $cost$, $deleteEdges \leftarrow$ HungarianSolve($CostMatrix$)
        if $cost > 2k_A$ then
            remove edge $\{i,j\}$ from $E(P_A)$
        else
            for all $e \in deletedEdges$ do
                $edgeUse(e) \leftarrow edgeUse(e) + 1$
    return $edgeUse$
```

values remain integer). This is because any edge deletion also lowers *some other* vertex w's degree by 1. If w is a neighbor of i or j this may "help" it with its matching. Since an edge deletion only has two endpoints it suffices to consider $\frac{1}{2}$ the degree difference. Suppose WLOG $d_{G_A}(u) > d_{G_A}(v)$ and u is independent of $(N(i) \setminus \{j\}) \cup (N(j) \setminus \{i\})$ (that is, u has no neighbors in $(N(i) \setminus \{j\}) \cup (N(j) \setminus \{i\})$). Then we know such a w does not exist, so any edge deletions from u will count only once. Hence we multiply by 2 in this case (recalling everything is doubled). Second, each vertex u_i is connected to at least k_A vertices of the type $\times_i$. These edges represent the deletion of edge $\{i, u\}$ and so get weight 2 or $+\infty$ if through branching the edge $\{i, u\}$ has become fixed. A similar process occurs for each v_j. Finally these "deletion" nodes $\times_i$ and $\times_j$ all have weight 0 between them since *not* deleting an edge from i or j is free.

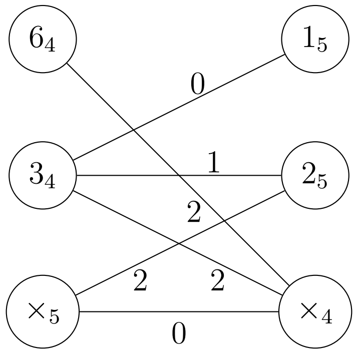
Now we turn to Algorithm 4. For each edge $\{i, j\} \in E(P_A)$ we use Algorithm 3 to construct a weighted bipartite graph based on the neighbors of i and j, and the number of edge deletions allowed. The Hungarian Algorithm [15, 24] is used to a find minimum cost perfect matching. From the solution we determine when an edge $\{u_i, \times_i\}$ or $\{v_j, \times_j\}$ is in the optimal assignment. The former corresponds to the deletion of edge $\{i, u\}$ in G_A, the later to the deletion of edge $\{j, v\}$. Finally we determine if the cost is more than $2k_A$, in which case we eliminate the mapping $i \to j$, and if not, we increment *edgeUse* based on the deleted edges.

Based on the preceding discussion, we arrive at the following:

Theorem 2. *Algorithm 4 is valid, that is, for a given node (G_A, P_A, E^F_A, k_A), if Algorithm 4 deletes edge $\{i, j\}$ from P_A, then i and j are not symmetric in any of its children. Further, Algorithm 4 dominates Algorithms 1 and 2, in that any mapping deleted by either Algorithm 1 or 2 is also deleted by Algorithm 4.*

Proof. Assume there exists a permutation $\pi \in \mathcal{AS}_k(G)$ such that $\pi(i) = j$. Thus in some child node we must have $d(i) = d(j)$, with fewer than k_A edge deletions. Since this implies the existence of π^{-1} such that $\pi^{-1}(j) = i$, we exclude j from $N(i)$ and i from $N(j)$ to avoid double counting; we represent these edge deletions explicitly with the nodes $\times_i$ and $\times_j$. Since any permutation must move neighbors to neighbors, for some

Figure 3: Edges with weight $+\infty$ have been excluded for clarity.



$I \subseteq N(i) \setminus \{j\}$ and $J \subseteq N(j) \setminus \{i\}$, $\pi : I \to J$ bijectively. Such subsets are given to us by a feasible assignment; we need now consider the edge deletions implicit in such an assignment. For each $u \in I$ let R_u be set of edge removals needed so that $d(u) = d(\pi(u))$ (note: $|R_u| \geq |d_{G_A}(u) - d_{G_A}(\pi(u))|$). Let $\mathcal{R} = \{R_{u_1}, R_{u_2}, \ldots, R_{u_{|I|}}\}$. We need to prove then that any edge e occurs at most twice in such a list; further if $d_{G_A}(u) > d_{G_A}(\pi(u))$ and u is independent of $(N(i) \setminus \{j\}) \cup (N(j) \setminus \{i\})$, any edge occurs at most once. For contradiction suppose such an edge e occurred m times in $\mathcal{R}$. Since I has distinct vertices and J has distinct vertices, if $m > 2$ then e has more than two endpoints, a contraction. Similarly suppose $d_{G_A}(u) > d_{G_A}(\pi(u))$ and u independent of $(N(i) \setminus \{j\}) \cup (N(j) \setminus \{i\})$. At least $(d_{G_A}(u) - d_{G_A}(\pi(u)))$ edges in R_u have an endpoint at u. Therefore at least $(d_{G_A}(u) - d_{G_A}(\pi(u)))$ many edges appear only once in $\mathcal{R}$ by the independence of u. Hence, assuming a minimum matching, line 6 of Algorithm 4 will return at least twice the minimal number of edge removals needed for $i \to j$.

The last part follows from noting that any feasible solution will be at least $2|d_{G_A}(i) - d_{G_A}(j)|$ and that if $d_{E_A^F}(i) > d_{G_A}(j)$ then in a perfect matching one of the fixed neighbors of i must be matched to one of the vertices $\times_i$. $\square$

Example 4. Continuing where Example 3 left off, we have the graph shown in Figure 2. Suppose $k_A = 1$ and P_A is as follows:

1 : {1, 2, 3, 5}
2 : {1, 2, 3, 4, 5}
3 : {1, 2, 3, 5, 6}
4 : {2, 3, 4, 5}
5 : {1, 2, 4, 5}
6 : {3, 6}

and suppose we enter the **for all** loop on line 5 of Algorithm 4 with $\{i, j\} = \{4, 5\} \in P_A$. Algorithm 3 constructs the bipartite graph shown in Figure 3. We see by inspection that the minimum cost perfect matching has cost $4 > 2k_A$, so mapping $\{4, 5\}$ can be deleted.

Finally, we note that in spite of Theorem 2, Algorithms 1 and 2 are still useful. A worst-case complexity bound on both is $\mathcal{O}(n^2)$, whereas assuming HungarianSolve is implemented efficiently (i.e., $\mathcal{O}(n^3)$), Algorithm 4 may need $\mathcal{O}(n^2(n + k_A)^3)$ time.

2.2 Branching

As mentioned above, at a given node (G_A, P_A, E_A^F, k_A), an edge $e \in E(G_A) \setminus E_A^F$ is selected for branching. Two children are created based on the selection of e, one where e is deleted (added to E_A^D) and k_A is decreased, and the other where e is added E_A^F.

We use the following rule for selecting e using the *edgeUse* array collected in Algorithm 4. Our first choice for a branching edge e will be a maximal element of this array. The hope is that in the deletion child, an edge that is "getting in the way" of symmetry is deleted. Conversely, in the child where the edge is fixed it is expected that such an edge will cause P_A to lose several edges at the next pass of Algorithm 4. Computational experiments (Section 4) show this to be a much better rule than random branching, and is in fact often among the best edges to branch on. If no such maximal edge is found, we fall back to the rule of branching on the first edge found in $E(G_A) \setminus E_A^F$. If it happens that $E(G_A) \setminus E_A^F = \emptyset$, this node is pruned.

We know that any automorphism maps edges to edges and non-edges to non-edges. Therefore if we have determined that two vertices are *fixed* with respect to automorphisms in all this node's children, we get the following helpful observation.

Observation 1. If $N_{P_A}(i) = \{i\}$ and $N_{P_A}(j) = \{j\}$, we need not branch on edge $\{i, j\}$.

Since i can only be mapped to i and j to j, then $\{i, j\}$ will always be mapped to itself, whether it is an edge or non-edge. The preceding discussion is summarized in Algorithm 5.

Algorithm 5 (FindBranchEdge) Selects an edge to branch on.

```
    function FindBranchEdge($G_A$, $P_A$, $E_A^F$, $edgeUse$)
        if max($edgeUse$) $> 0$ then
            return argmax($edgeUse$)
        for all $\{i,j\} \in E(G_A) \setminus E_A^F$ do
5:          if $N_{P_A}(i) \neq \{i\}$ or $N_{P_A}(j) \neq \{j\}$ then
                return $\{i,j\}$
        return prune        ▷ If here either $E(G_A) \setminus E_A^F = \emptyset$ or all edges satisfy Remark 1
```

2.3 Bounding

Bounding is done in two ways. First, after an edge is deleted we compute the symmetries of the modified graph G_A. The number of orbits in the orbital partition gives an upper bound on the solution value, since this is a feasible solution for (1). Lower bounding is done using the information in P_A. For a lower bound using P_A, we can partition $V(= V(G) = V(G_A) = V(P_A))$ using the following rule:

> Two vertices in V can belong to the same partition if there exists an edge between them in P_A.

The set of all such partitionings of V represent all possible orbital partitions at this node. Therefore in order to generate a valid lower bound we would need the minimum of such partitionings. This leads to the following observation.

Remark 2. Partitioning V as described above is the same as vertex coloring P_A's complement, $\overline{P_A}$.

Since finding the chromatic number of a graph [12] and approximating the chromatic number of a graph [9] are both NP-hard, we settle for a "bad" lower bound on the partition size, namely we use the size of a greedily constructed independent set for P_A. This has the merit though of only requiring $\mathcal{O}(n)$ time. Defining $\chi(G)$, $\alpha(G)$, and $\omega(G)$ to be the chromatic number, independence number, and clique number of G, respectively, we know from basic graph theory $\chi(\overline{P_A}) \geq \omega(\overline{P_A}) = \alpha(P_A) \geq |I|$, for any independent set I. Therefore we have the following bounding procedure:

> After any call to Algorithms 1, 2, or 4, compute the size of a maximal independent set of P_A. If this is greater than or equal the current incumbent, we prune.

2.4 An Algorithm

We now tie together the preceding discussion in Algorithm 6 that solves (1) using a depth-first search. The routines HungarianSolve, ComputeAutomorphisms and GreedyIndependentSetSize are treated as a black-box. We similarly assume PopFromStack and AppendToStack manage *NodeStack*.

Theorem 3. *Algorithm 6 is valid.*

Proof. This follows from Theorem 2. If we exclude the logic which prunes by bound (lines 20-22, 25-27) and collected the permutations computed in line 10 we find all the k-almost symmetries of G. □

3 Implementation

In this section we discuss some of the implementation choices made and libraries used; however, the interested reader is referred directly to the source code for details. C++ is used throughout.

`PEBBL` is a general-purpose parallel branch-and-bound framework written in C++ [6]. `PEBBL` allows the easy implementation of a parallel branch-and-bound algorithm provided the user already has a serial implementation in mind (such as Algorithm 6). `PEBBL` is therefore used for tree management. `PEBBL` has many user-configurable options, in particular the user can specify the search order to be breadth-first, depth-first, or the default best-first, which in the minimization case will select a problem with the lowest bound. One particularly powerful feature of `PEBBL` is its ability to exploit parallelism during the *ramp-up* phase. Ramping-up occurs when the number of active nodes in the search tree is smaller than the number of available processors. During ramp-up, before parallel enumeration begins, `PEBBL` has all threads synchronously explore the same nodes of the branch-and-bound tree near the root node. This allows the programmer to exploit parallelism that may be present within each node using MPI communication. Since the running time of the Hungarian Algorithm is $\mathcal{O}(n^3)$, a parallel version of Algorithm 4 is used during ramp-up. Lines 5-6 are parallelized; we do an MPI reduce operation to

Algorithm 6 (FindAlmostSymmetry) Solves (1)

```
    function FindAlmostSymmetry($G$, $k$)
        initialize $P_R \leftarrow$ {complete graph with self-loops}; $E_R^D \leftarrow \emptyset$; $E_R^F \leftarrow \emptyset$
        initialize $delChild \leftarrow$ true
        initialize $incumbentValue \leftarrow |G|$; $incumbentSolution \leftarrow \emptyset$
 5:     initialize $NodeStack \leftarrow \{ (P_R, E_R^D, E_R^F, delChild) \}$
        while $NodeStack \neq \emptyset$ do
            $(P_A, E_A^D, E_A^F, delChild) \leftarrow$ PopFromStack
            $G_A \leftarrow G - E_A^D$.
            $k_A \leftarrow k - |E_A^D|$
10:         if $delChild$ then                 ▷ This node is either root or has a new edge in $E_A^D$
                $orbitNum \leftarrow$ ComputeAutomorphisms($G_A$)
                if $orbitNum < incumbentValue$ then
                    $incumbentValue \leftarrow orbitNum$; $incumbentSolution \leftarrow E_A^D$
                if $k_A = 0$ or $|E_A^F| = |E(G_A)|$ then
15:                 prune                                 ▷ (Delete this node and go to line 6)
                DegreeDiffElim($G_A$, $P_A$, $k_A$)
            else                                          ▷ This node has a new edge in $E_A^F$
                if $|E_A^F| = |E(G_A)|$ then
                    prune
20:             FixedDegElim($G_A$, $P_A$, $E_A^F$)
            $lowerBound \leftarrow$ GreedyIndependentSetSize($P_A$)
            if $lowerBound \geq incumbentValue$ then
                prune
            while $P_A$ is changed by RefineByMatching() do
25:             $edgeUse \leftarrow$ RefineByMatching($G_A$, $P_A$, $E_A^F$, $k_A$)
                $lowerBound \leftarrow$ GreedyIndependentSetSize($P_A$)
                if $lowerBound \geq incumbentValue$ then
                    prune
            $branchEdge \leftarrow$ FindBranchEdge($G_A$, $P_A$, $E_A^F$, $edgeUse$)
30:                               ▷ Once we have arrived here all the bounding we can do is done
            if $branchEdge =$ prune then
                prune
                                                          ▷ Else we will create the children
            AppendToStack($P_A$, $E_A^D$, $E_A^F + branchEdge$, false)        ▷ Edge fixing child
            AppendToStack($P_A$, $E_A^D + branchEdge$, $E_A^F$, true)        ▷ Edge deletion child
35:     return $incumbentValue$, $incumbentSolution$
```

collect P_A *or edgeUse.* Notice in RefineByMatching we only use *edgeUse* if P_A does not change, so it does not need to be reduced, and we only need to reduce P_A if it does change, in which case we will call RefineByMatching again and not use *edgeUse.* Since we would expect P_A to be fairly dense high in the tree this is helpful. Once *cross-over* occurs and `PEBBL` is doing parallel enumeration the serial version of Algorithm 4 is used.

A C implementation of the Hungarian Method from [28] is used for Hungarian-Solve, which itself is an enhancement of the implementation provided by the Stanford GraphBase [14]. This code was slightly modified to avoid redundant memory allocation/deallocation in the second **for all** loop of Algorithm 4, and thus is included with the source code.

Finally, `nauty` 2.5r9 [22] is used as the implementation of ComputeAutomorphisms in Algorithm 6, with canonical labeling turned off. `nauty`'s packed graph format is used for graph representation. GreedyIndependentSetSize uses the standard greedy procedure to construct an independent set and returns the cardinality of the constructed set. OpenMPI was the MPI implementation used for all tests.

4 Computational Results

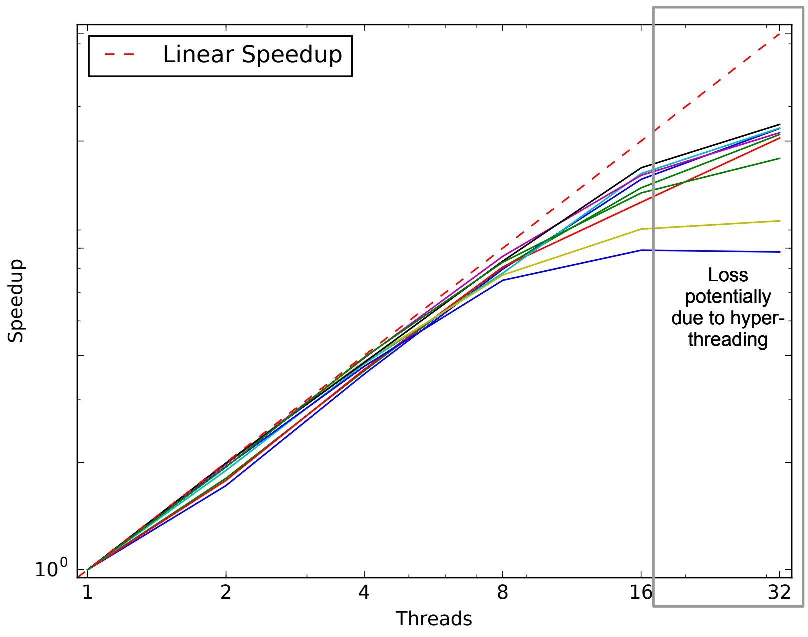
All computational experiments were done on a Dell PowerEdge T620 with 2 Intel Xeon E5-2670 processors and 256GB of memory running Ubuntu 14.04.2. Hyper-threading was enabled for a total of 16 cores and 32 threads. Random graph instances from [23] and DIMACS coloring instances from [4] were analyzed. Coloring instances that already exhibited large amounts of symmetry (i.e., $\gamma_0^G \leq \frac{1}{2}|V(G)|$) were excluded from testing. The test set was further reduced by only selecting a subset of the Leighton graphs. All times are wall-clock times reported by `PEBBL`.

The results of the main experiment are reported in Table 2. γ_k^G (equation (1)) was computed for each graph, incrementing k. A wall clock limit of 60 minutes was used, and k was no longer incremented after hitting the time limit. We also report on the size of the automorphism group for the optimal subgraph G', labeled $|\text{Aut}(G')|$. Note that for presentation reasons, we sometimes scale this value by a constant. Since `PEBBL`'s parallel search is non-deterministic, each experiment is repeated five times, and the average search time is reported. If every run timed-out we report that with a †. On instances where at least one repetition timed-out we report the best objective value of a solution found across all five. An asterisk indicates that we were not able to prove the objective value optimal in any of the five repetitions. For conciseness some levels of k are excluded. All 32 available threads were used for this test, and `PEBBL`'s best-first search was used to explore the tree.

From the objective value in Table 2 we can see that for many of the structured graphs, many fewer edges than Theorem 1 requires need be removed to induce a non-trivial k-almost symmetry. Additionally there is often a commensurate increase in group size as the number of orbits decreases. Inducing symmetry on random graphs is more difficult, as is to be expected given Theorem 2 in [7] (the bound from Theorem 1 is tight in the limit for random graphs).

We also examined how the algorithm scales. Given the results in Table 2 some easier instances were selected and ran with a varying number of threads up to 32. Note that

Figure 4: Scaling on various problems



the system used only has 16 cores, so linear speedup cannot be expected past that even in the ideal case. The results are in Figure 4. We can see good scaling through 8 cores on the test examples, starting to taper off at 16 cores. Individual test cases are detailed in the appendix.

4.1 Branching Strategy

We compare the branching strategy dictated by *edgeUse* (*edgeUse* branching) against selecting a random eligible edge to branch on. We also consider a "local branching" rule, in which after we have done all the valid refining that can be done at a node, we select the branching edge by doing an additional single pass at RefineByMatching with $k = 1$. Several easy instances from above were tested using a single thread. Each random branching trial was replicated 50 times, and compared to the *edgeUse* branching strategy and the local branching strategy. The results are summarized in the graphs in Figure 5, where the box-plots are the random branching trials, the red square with the text on the right represents the *edgeUse* branching strategy, and the blue $\times$ with the text on the left represents the local branching strategy. We show four examples here, the remaining are available in the appendix. The *edgeUse* branching strategy laid out in Section 2.2 was always at least as good (and often much better) than random in the examples tested, and in addition was always better than the local branching strategy. While the local branching strategy was sometimes competitive with *edgeUse* branching, in other cases it does worse (in a few instances significantly so) than the random branching mean.

In order to test the strength of *edgeUse* branching, we test it against strong branching at the root node. Strong branching is implemented at the root node by considering the number of permutations refined in the "edge fixing" child for every edge in the graph

Table 2: Computational Results

`games120.col` $n = 120$ $e = 638$

k	0	1	2	3	4	5	6	7	8	9
γ_k^G	119	118	117	114	113	112	112	111	111	112*
$\lvert\mathrm{Aut}(G')\rvert$	2	4	8	8	16	48	48	16	16	48*
seconds	0.0	0.0	0.1	0.2	0.2	0.4	2.3	27.8	896.9	†

`miles250.col` $n = 128$ $e = 387$

k	0	1	2	3	4	5	6	7
γ_k^G	108	106	104	102	100	99	97	97*
$\lvert\mathrm{Aut}(G')\rvert/10^7$	0.26542	3.1851	6.3701	12.740	203.84	1019.2	815.37	6115.3*
seconds	0.0	0.2	0.3	0.8	3.1	57.6	844.8	†

`miles500.col` $n = 128$ $e = 1170$

k	0	1	2	3	4	5	6	7	8
γ_k^G	114	113	111	110	109	108	107	106	107*
$\lvert\mathrm{Aut}(G')\rvert/10^6$	0.82944	1.6580	3.3178	6.6355	46.449	92.897	185.79	53.084	371.59*
seconds	0.0	0.1	0.2	0.4	1.5	14.7	186.1	1868	†

`miles750.col` $n = 128$ $e = 2113$

k	0	1	2	3	4	5	6	7	8
γ_k^G	122	121	120	119	118	117	116	115	115*
$\lvert\mathrm{Aut}(G')\rvert$	96	288	576	1152	2304	4608	9216	18432	18432*
seconds	0.0	0.1	0.3	0.5	1.4	9.1	86.3	849.1	†

`miles1000.col` $n = 128$ $e = 3216$

k	0	1	2	3	4	5	6	7
γ_k^G	123	122	121	120	119	118	118	119*
$\lvert\mathrm{Aut}(G')\rvert$	72	144	288	576	1152	2304	2304	1152*
seconds	0.0	0.1	0.2	0.5	2.2	33.5	416.9	†

`miles1500.col` $n = 128$ $e = 5198$

k	0	1	2	3	4	5
γ_k^G	102	101	100	99	98	97*
$\lvert\mathrm{Aut}(G')\rvert/10^{12}$	0.11466	0.80263	3.2105	4.5865	19.263	57.790*
seconds	0.0	0.8	2.6	40.0	913.5	†

`le450_5b.col` $n = 450$ $e = 5734$

k	0	13	14	15	16	17	18	19
γ_k^G	450	450	450	450	450	450	450	449*
$\lvert\mathrm{Aut}(G')\rvert$	1	1	1	1	1	1	1	2*
seconds	0.0	1.1	1.2	1.6	2.0	5.0	564.1	†

`le450_15b.col` $n = 450$ $e = 8169$

k	0	1	2	4	5	7	8	10	11	14	15	16	17
γ_k^G	450	450	449	449	448	448	447	447	446	446	445	445	446*
$\lvert\mathrm{Aut}(G')\rvert$	1	1	2	2	6	6	24	24	120	120	720	720	120*
seconds	0.0	0.1	0.2	0.4	0.5	0.8	0.9	1.1	1.2	8.3	19.7	93.2	†

`le450_25b.col` $n = 450$ $e = 8263$

k	0	1	2	3	4	5	6	7	8	9	10
γ_k^G	450	450	449	449	449	448	448	447	447	447	447*
$\lvert\mathrm{Aut}(G')\rvert$	1	1	2	2	2	6	6	12	12	12	12*
seconds	0.0	0.1	0.2	0.3	0.6	0.7	0.9	1.6	7.1	43.8	†

Table 2: Computational Results, continued

`ran10_100_a.bliss`								$n = 100$ $e = 502$
k	0	6	7	8	9	10	11	12
γ_k^G	100	100	99	99	99	99	99	100*
$\lvert\mathrm{Aut}(G')\rvert$	1	1	2	2	2	2	2	1*
seconds	0.0	0.0	0.1	0.4	2.8	22.8	372.0	†

`ran10_100_b.bliss`									$n = 100$ $e = 464$
k	0	3	4	7	8	9	10	11	12
γ_k^G	100	100	99	99	99	99	98	98	100*
$\lvert\mathrm{Aut}(G')\rvert$	1	1	2	2	2	2	3	3	1*
seconds	0.0	0.0	0.0	0.2	4.0	11.1	116.6	1924.3	†

`ran10_100_c.bliss`										$n = 100$ $e = 525$
k	0	5	6	7	8	9	10	11	12	13
γ_k^G	100	100	99	99	99	99	99	98	98	100*
$\lvert\mathrm{Aut}(G')\rvert$	1	1	2	2	2	2	2	6	6	1*
seconds	0.0	0.0	0.0	0.1	0.5	2.3	9.1	117.5	1712	†

`ran10_100_d.bliss`								$n = 100$ $e = 514$
k	0	1	7	8	9	10	11	12
γ_k^G	100	100	100	99	99	99	99	100*
$\lvert\mathrm{Aut}(G')\rvert$	1	1	1	2	2	2	2	1*
seconds	0.0	0.0	0.1	0.9	5.2	25.8	560.3	†

G. We then rank the potential edge branches by the number of permutations removed by RefineByMatching for the candidate edge in the edge fixing child node. The results are presented in Table 3, where we present the ranks in percentiles. Instances which are solved at the root node are discarded. In particular, in the column labeled "*edgeUse* branch percentile rank" we give the percentage of edges with a worse score than the *edgeUse* branch. The column labeled $\frac{\text{\# refined by } \textit{edgeUse} \text{ branch}}{\text{\# refined by strong branch}}$ we present the ratio of permutations refined by the *edgeUse* branch against the permutations refined by the strongest branch. Hence, if this is 1, that means the *edgeUse* branch is as good as the strong branch. First, we can see based on the *edgeUse* branch percentile rank that it is often in the top 1% of possible edge choices based on our strong branching score, and in only 6 of the 99 cases is it below the 80th percentile. Turning to the ratio, we see that in 59 of the 99 cases *edgeUse* branching selects an edge with the highest rank. However, in 18 of the 99 cases the *edgeUse* branch is 80% worse by score than the strong branch. Note that deciding an *edgeUse* branch is a linear check after RefineByMatching, whereas strong branching is a potentially $\mathcal{O}(m \cdot n^2(n+k)^3)$ check at the root node (where n is the number of vertices and m is the number of edges in the graph G). In this light, *edgeUse* branching seems both practical and effective for selecting a disjunction. Additionally, these results (along with the results in Section 4.2) verify the notion that the edges that end up being deleted in the matchings are often those that are "in the way" of symmetry.

We also demonstrate the robustness of *edgeUse* branching. Recall, for a given node, *edgeUse* branching selects the most frequently deleted edge in the refinement. To test the robustness we examine how the ranking changes after branching and refining. In particular, how often is the second place edge chosen for branching in the child problems? If it has a high rank, this suggests that (1) our choice of edge is somewhat insensitive to changes in the graph, and (2) we may be able to get away with refining less frequently, while still maintaining the strength of our branching strategy. If its rank is 1, then this

Table 3: *edgeUse* branching vs. strong branching at root node

`games120.col`									$n = 120$ $e = 638$
k	1	2	3	4	5	6	7	8	9
edgeUse branch percentile rank	99.9	49.4	99.8	100	100	99.2	1.0	99.2	94.9
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	1.0	0.0	0.50	1.0	1.0	0.185	1.0	0.033	0.020

`miles250.col`							$n = 128$ $e = 387$
k	1	2	3	4	5	6	7
edgeUse branch percentile rank	99.8	99.4	94.8	83.0	94.9	85.5	82.6
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	1.0	0.812	0.342	0.094	0.345	0.214	0.279

`miles500.col`								$n = 128$ $e = 1170$
k	1	2	3	4	5	6	7	8
edgeUse branch percentile rank	99.9	99.9	100.0	99.9	99.9	99.7	99.9	100
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	1.0	1.0	1.0	1.0	0.692	0.850	0.857	1.0

`miles750.col`								$n = 128$ $e = 2113$
k	1	2	3	4	5	6	7	8
edgeUse branch percentile rank	99.9	99.8	99.7	1.0	99.9	99.9	96.7	99.8
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	1.0	1.0	1.0	1.0	1.0	0.900	0.500	0.916

`miles1000.col`							$n = 128$ $e = 3216$
k	1	2	3	4	5	6	7
edgeUse branch percentile rank	99.9	99.9	99.8	99.9	100	99.9	99.9
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0

`miles1500.col`					$n = 128$ $e = 5198$
k	1	2	3	4	5
edgeUse branch percentile rank	99.9	100	99.9	99.9	99.9
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	1.0	1.0	1.0	1.0	1.0

`le450_5b.col`		$n = 450$ $e = 5734$
k	18	19
edgeUse branch percentile rank	87.5	78.8
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	0.030	0.019

`le450_15b.col`								$n = 450$ $e = 8169$
k	2	3	4	5	6	7	8	9
edgeUse branch percentile rank	99.9	99.9	99.9	99.9	99.9	99.9	99.9	99.9
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
k	10	11	12	13	14	15	16	17
edgeUse branch percentile rank	99.9	99.9	99.9	99.9	99.9	100	100	99.9
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

`le450_25b.col`									$n = 450$ $e = 8263$
k	2	3	4	5	6	7	8	9	10
edgeUse branch percentile rank	99.9	100	99.9	100	99.9	99.9	100	100	100
$\frac{\#\text{ refined by } \textit{edgeUse} \text{ branch}}{\#\text{ refined by strong branch}}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Figure 5: Random branching (box plot) vs. *edgeUse* branching (🟥, number of nodes on right) vs. local branching (×, number of nodes on left)

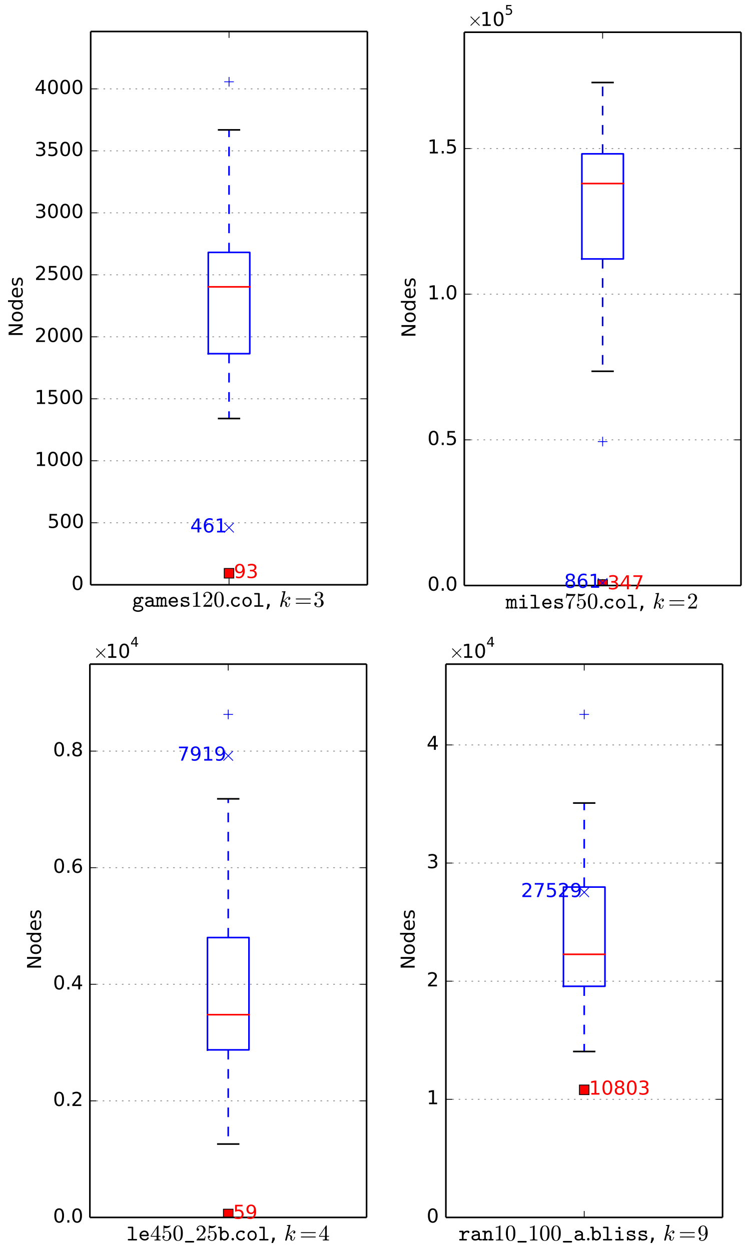


Table 3: *edgeUse* branching vs. strong branching at root node

`ran10_100_a.bliss`							$n = 100$ $e = 502$
k	7	8	9	10	11	12	
edgeUse branch percentile rank	100	96.4	84.7	91.2	93.0	91.6	
$\frac{\text{\# refined by } \textit{edgeUse} \text{ branch}}{\text{\# refined by strong branch}}$	1.0	0.130	0.250	0.276	0.219	0.093	

`ran10_100_b.bliss`									$n = 100$ $e = 464$
k	4	5	6	7	8	9	10	11	12
edgeUse branch percentile rank	99.6	99.6	100	100	95.0	68.5	72.7	83.1	96.7
$\frac{\text{\# refined by } \textit{edgeUse} \text{ branch}}{\text{\# refined by strong branch}}$	1.0	1.0	1.0	1.0	0.297	0.102	0.102	0.095	0.179

`ran10_100_c.bliss`								$n = 100$ $e = 525$
k	6	7	8	9	10	11	12	13
edgeUse branch percentile rank	99.5	99.8	92.0	97.2	79.7	89.5	97.1	95.6
$\frac{\text{\# refined by } \textit{edgeUse} \text{ branch}}{\text{\# refined by strong branch}}$	1.0	1.0	0.036	0.666	0.101	0.098	0.343	0.238

`ran10_100_d.bliss`						$n = 100$ $e = 514$
k	8	9	10	11	12	
edgeUse branch percentile rank	99.2	85.8	69.6	97.7	99.2	
$\frac{\text{\# refined by } \textit{edgeUse} \text{ branch}}{\text{\# refined by strong branch}}$	0.364	0.175	0.081	0.454	0.692	

Table 4: *edgeUse* branching robustness, selected instances

(a) `games120.col`, $k = 7$

Rank	% of Nodes
1	56.54%
2	16.02%
3	4.70%
4	3.24%
5	2.51%
6	2.05%
7	1.71%
8	1.33%
9	1.41%
10	1.00%
11+	9.48%

(b) `miles750.col`, $k = 5$

Rank	% of Nodes
1	85.56%
2	1.88%
3	0.09%
4	0.10%
5	0.12%
6	0.63%
7	1.42%
8	0.98%
9	0.66%
10	0.64%
11+	7.92%

(c) `le450_25b.col`, $k = 9$

Rank	% of Nodes
1	72.43%
2	0.31%
3	0.14%
4	0.05%
5	0.10%
6	0.09%
7	0.08%
8	0.04%
9	0.11%
10	0.22%
11+	26.41%

(d) `ran10_100_a.bliss`, $k = 10$

Rank	% of Nodes
1	76.81%
2	0.37%
3	0.17%
4	0.28%
5	1.24%
6	3.07%
7	3.26%
8	3.15%
9	0.92%
10	0.04%
11+	10.70%

is the exact edge we branched on in the child problems. The tests were all done on one thread, and the results are reported in Table 4. We report the rank in the child (Rank) of the second most frequently deleted edge in the parent refinement, and the percentage of nodes in the tree where it has a given rank (% of Nodes) for selected instances. (The other instances tests are reported in the appendix.) As we can see, it is often the case that the next edge selected for branching is that with second rank in the parent node. Note that as implemented, we refine as much as possible at each node to attempt to lift the bound, so it may be computationally advantageous to refine less frequently, at the possible cost of more nodes, while not losing much in the strength of our branching decisions. In the appendix we present additional computational results that attempt to exploit this using a modified branching strategy.

4.2 Heuristics

To exploit almost symmetries in a problem such as (2), it is not necessary to compute the optimal group of k-almost symmetries. Toward that end, we examine how well both *edgeUse* branching and local branching work as a heuristic for the problems in Table 2 by just diving left and never creating the edge fixing child. That is, we consider Algorithm 6 with line 33 excluded.

All heuristic tests were done on a single thread. The main result is in Table 5, where we compare the optimality gap closed by both the *edgeUse* branching and local branching rule, where the optimality gap closed is defined as

$$\text{optimality gap closed} := \frac{\gamma_0^G - \lambda_k^G}{\gamma_0^G - \gamma_k^G}, \tag{3}$$

where λ_k^G is the objective value of the heuristic solution, and γ_k^G, γ_0^G are the optimal values from Table 2. Recall that γ_0^G is the number of vertex orbits in the graph G, so we measure performance relative to this trivial solution. That is, a value of 0% means $\lambda_k^G = \gamma_0^G$, i.e., no improvement in objective value, and a value of 100% means $\lambda_k^G = \gamma_k^G$, i.e., the heuristic found a globally optimal solution. For k such that $\gamma_0^G = \gamma_k^G$ we show a "–". Timing results are reported in Table 6. As we can see, for `games120.col` and the `miles` graphs, the performance of the *edgeUse* branching dive is not impressive, especially for larger values of k. That being said, for the `le450` and random graphs it perpforms better, and usually either finds the optimal objective value or is only one away. Turning to the local branching heuristic, we see the situation is exactly reversed from before. The local branching dive is able to do well with `games120.col` and the `miles` graphs, whereas it is not as successful finding the little almost symmetry in the `le450` and random graphs. Both heuristics can be run usually in under a few seconds for graphs of this size, with the exception being `le450_5b.col` for large values of k, which is a graph with least almost symmetry examined. Overall the *edgeUse* branching heuristic performs the best, usually capturing a solution that is within 50% of optimal with only a few seconds of computational effort on a single thread.

5 Conclusion

We presented and tested a branch-and-bound algorithm for solving a generalized version of Automorphism-Partition. We provide a branching strategy which is much more effective at controlling the size of the tree, compared to random branching, even on random graphs, and provide computational evidence that it is relatively robust. The branching strategy presented can often be used heuristically to find a non-trivial set of almost-symmetries (i.e., not Aut(G)) in a relatively short time period by diving left. Finally, we demonstrated that parallel enumeration is effective in speeding up the wall-clock solution times for our implementation.

Acknowledgments

The research reported in this paper was funded in part by NSF award number 1332662 and NSF award number 1333789.

Table 5: % gap reduced for heuristics

edgeUse branching heuristic:

Graph / k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`games120.col`	100%	100%	100%	17%	14%	14%	0%	13%										
`miles250.col`	0%	75%	33%	25%	22%	0%												
`miles500.col`	0%	67%	50%	60%	50%	43%	38%											
`miles750.col`	0%	50%	67%	75%	60%	50%	43%											
`miles1000.col`	0%	0%	33%	50%	40%	60%												
`miles1500.col`	0%	50%	33%	25%														
`le450_5b.col`	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
`le450_15b.col`	–	100%	100%	100%	100%	100%	100%	100%	100%	100%	75%	100%	100%	100%	80%	80%		
`le450_25b.col`	–	100%	100%	100%	50%	50%	67%	67%	67%									
`ran10_100_a.bliss`	–	–	–	–	–	–	100%	100%	100%	100%	100%							
`ran10_100_b.bliss`	–	–	–	100%	100%	100%	100%	100%	100%	50%	50%							
`ran10_100_c.bliss`	–	–	–	–	–	100%	100%	100%	100%	100%	50%	50%						
`ran10_100_d.bliss`	–	–	–	–	–	–	–	0%	0%	0%	0%							

Local branching heuristic:

Graph / k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`games120.col`	100%	100%	40%	100%	86%	86%	75%	75%										
`miles250.col`	0%	25%	50%	38%	44%	45%												
`miles500.col`	0%	0%	25%	40%	33%	29%	25%											
`miles750.col`	0%	50%	67%	50%	40%	50%	57%											
`miles1000.col`	0%	0%	33%	50%	60%	80%												
`miles1500.col`	0%	50%	67%	75%														
`le450_5b.col`	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
`le450_15b.col`	–	100%	100%	100%	50%	100%	100%	67%	67%	67%	50%	50%	50%	50%	40%	40%		
`le450_25b.col`	–	0%	0%	0%	0%	0%	0%	0%	0%									
`ran10_100_a.bliss`	–	–	–	–	–	–	0%	0%	0%	0%	0%							
`ran10_100_b.bliss`	–	–	–	0%	0%	100%	0%	0%	0%	0%	0%							
`ran10_100_c.bliss`	–	–	–	–	–	0%	100%	100%	100%	100%	50%	50%						
`ran10_100_d.bliss`	–	–	–	–	–	–	–	0%	0%	0%	0%							

Table 6: Time in seconds for heuristics

edgeUse branching heuristic:

Graph / k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`games120.col`	0.1	0.1	0.1	0.2	0.7	0.9	1.1	1.2										
`miles250.col`	0.0	0.0	0.1	0.1	0.1	0.3												
`miles500.col`	0.0	0.0	0.1	0.1	0.1	0.2	0.2											
`miles750.col`	0.0	0.1	0.1	0.1	0.2	0.3	0.3											
`miles1000.col`	0.0	0.1	0.2	0.2	0.3	0.4												
`miles1500.col`	0.2	0.5	0.7	1.0														
`le450_5b.col`	0.6	0.9	1.3	1.4	1.8	1.9	2.2	2.4	2.4	2.7	2.9	3.2	3.3	3.8	4.7	6.3	19.6	51.0
`le450_15b.col`	0.4	0.8	0.9	1.2	1.2	1.4	1.7	1.8	1.9	2.2	2.4	2.4	2.8	2.9	3.2	3.2		
`le450_25b.col`	0.3	0.7	0.8	0.9	1.2	1.2	1.3	1.7	1.7									
`ran10_100_a.bliss`	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.4	0.7	0.9	0.7							
`ran10_100_b.bliss`	0.0	0.0	0.0	0.0	0.1	0.1	0.3	0.6	0.7	0.9	0.8							
`ran10_100_c.bliss`	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.5	0.8	0.8	0.8	1.1						
`ran10_100_d.bliss`	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.4	0.8	0.8	0.8							

Local branching heuristic:

Graph / k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`games120.col`	0.1	0.1	0.1	0.2	0.6	1.0	1.1	1.2										
`miles250.col`	0.0	0.0	0.1	0.1	0.2	0.3												
`miles500.col`	0.0	0.0	0.1	0.1	0.1	0.2	0.2											
`miles750.col`	0.0	0.1	0.1	0.2	0.2	0.3	0.4											
`miles1000.col`	0.0	0.1	0.2	0.2	0.3	0.5												
`miles1500.col`	0.2	0.5	0.9	1.0														
`le450_5b.col`	0.7	1.0	1.4	1.6	1.8	2.0	2.0	2.4	2.5	2.5	3.4	3.1	3.3	3.8	4.6	6.4	19.5	35.3
`le450_15b.col`	0.4	0.7	0.9	1.0	1.1	1.4	1.5	1.9	2.0	2.0	2.4	2.5	2.8	2.5	3.0	3.2		
`le450_25b.col`	0.3	0.6	0.9	0.9	1.0	1.1	1.4	1.5	1.6									
`ran10_100_a.bliss`	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.4	0.6	1.0	0.9							
`ran10_100_b.bliss`	0.0	0.0	0.0	0.0	0.1	0.1	0.3	0.6	0.9	1.0	1.0							
`ran10_100_c.bliss`	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.5	0.9	0.9	1.0	0.9						
`ran10_100_d.bliss`	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.4	0.8	0.9	1.0							

References

[1] Vikraman Arvind, Johannes Köbler, Sebastian Kuhnert, and Yadu Vasudev. Approximate graph isomorphism. In *Mathematical Foundations of Computer Science 2012*, pages 100–111. Springer, 2012.

[2] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2016, pages 684–697, New York, NY, USA, 2016. ACM.

[3] Christoph Buchheim and Michael Jünger. An integer programming approach to fuzzy symmetry detection. In *International Symposium on Graph Drawing*, pages 166–177. Springer, 2003.

[4] Joe Culberson, David Johnson, Gary Lewandowski, and Michael Trick. Graph coloring instances. `http://mat.gsia.cmu.edu/COLOR/instances.html`, March 2015.

[5] Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 149–154, New York, NY, USA, 2008. ACM.

[6] Jonathan Eckstein, William E Hart, and Cynthia A Phillips. PEBBL: an object-oriented framework for scalable parallel branch and bound. *Mathematical Programming Computation*, 7(4):429–469, 2015.

[7] Paul Erdős and Alfréd Rényi. Asymmetric graphs. *Acta Mathematica Hungarica*, 14(3):295–315, 1963.

[8] Uriel Feige. Relations between average case complexity and approximation complexity. In *Proceedings of the thiry-fourth annual ACM symposium on theory of computing*, pages 534–543. ACM, 2002.

[9] Uriel Feige and Joe Kilian. Zero knowledge and the chromatic number. In *Computational Complexity, 1996. Proceedings., Eleventh Annual IEEE Conference on*, pages 278–287. IEEE, 1996.

[10] Maria Fox, Derek Long, and Julie Porteous. Discovering near symmetry in graphs. In *Proceedings of the 22nd national conference on Artificial intelligence-Volume 1*, pages 415–420. AAAI Press, 2007.

[11] Carl Fürstenberg. A drawing of a graph. `http://en.wikipedia.org/wiki/Graph_theory\#mediaviewer/File:6n-graf.svg`, March 2015.

[12] Michael R Garey, David S Johnson, and Larry Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM symposium on theory of computing*, pages 47–63. ACM, 1974.

[13] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38(3):690–728, 1991.

[14] Donald Ervin Knuth. *The Stanford GraphBase: a platform for combinatorial computing*, volume 37. Addison-Wesley Reading, 1993.

[15] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[16] Chih-Long Lin. Hardness of approximating graph transformation problem. In *Algorithms and Computation*, pages 74–82. Springer, 1994.

[17] François Margot. Pruning by isomorphism in branch-and-cut. *Mathematical Programming*, 94(1):71–90, 2002.

[18] François Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming*, 98(1-3):3–21, 2003.

[19] I Markov. Almost-symmetries of graphs. In *Proc. International Symmetry Conference (ISC)*, pages 60–70, 2007.

[20] Rudolf Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–136, 1979.

[21] Brendan D McKay. *Practical graph isomorphism.* Department of Computer Science, Vanderbilt University, 1981.

[22] Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014.

[23] Brendan D McKay and Adolfo Piperno. Nauty traces – graphs. `http://pallini.di.uniroma1.it/Graphs.html`, March 2015.

[24] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial & Applied Mathematics*, 5(1):32–38, 1957.

[25] Ryan O'Donnell, John Wright, Chenggang Wu, and Yuan Zhou. Hardness of robust graph isomorphism, lasserre gaps, and asymmetry of random graphs. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1659–1677. SIAM, 2014.

[26] James Ostrowski, Jeff Linderoth, Fabrizio Rossi, and Stefano Smriglio. Orbital branching. *Mathematical Programming*, 126(1):147–178, 2011.

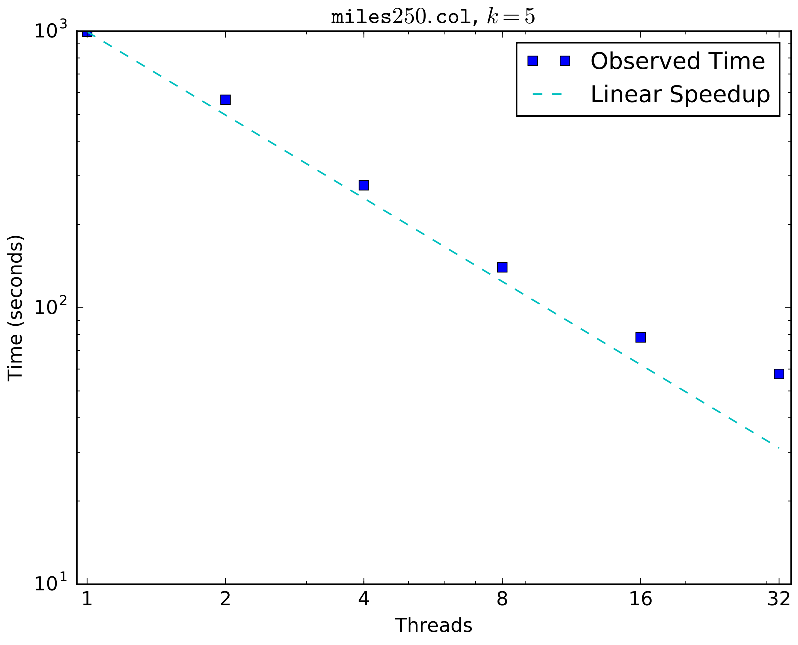
[27] Ronald C Read and Derek G Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

[28] Cyrill Stachniss. C implementation of the hungarian method. `http://www2.informatik.uni-freiburg.de/~stachnis/misc.html`, March 2015.

Appendix

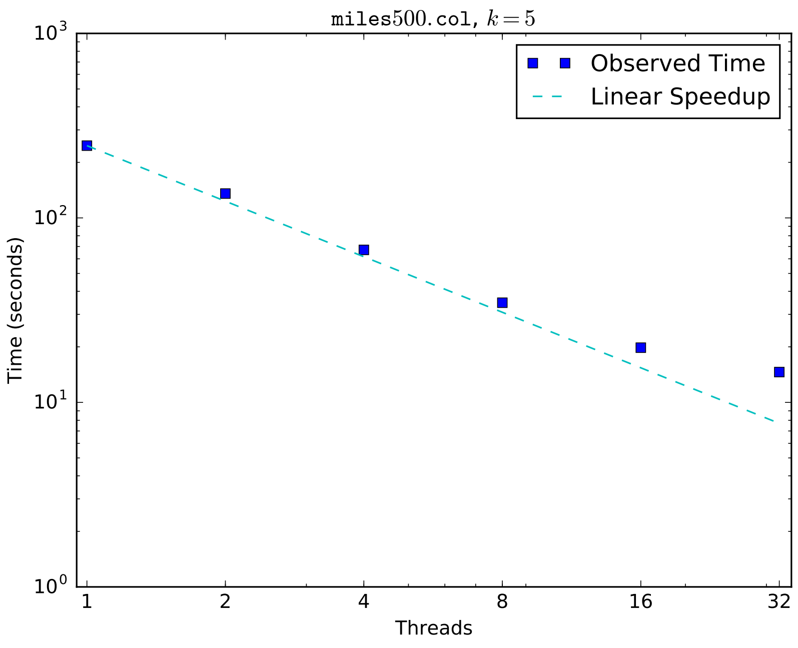
In this appendix we break-out the scaling graph in Figure 4 into scaling graphs for the individual cases tested. These are presented in Figure 6. We also present the rest of the results on branching choice robustness, similar to that in Table 4, in Table 7. Also, we provide more test-cases for the branching strategy similar to those given in Figure 5. These results are in Figure 7. Finally, in Table 8 we present some computational results on the test suite from Table 2 with a modified branching strategy. In particular, at each node we create three children, two of which are deletion nodes for the two edges highest-ranked by *edgeUse* and the third fixing both these edges. The hope is this makes the fixing child stronger, making for a more balanced tree while also not loosing much due to the robustness of the branching selection demonstrated in Section 4.1. As we can see, this branching strategy is better for the random graphs, `miles1500.col`, and `le450_5b.col`, but is worse for the remaining graphs. Overall this strategy apparently does not effectively exploit robustness of *edgeUse* branching.

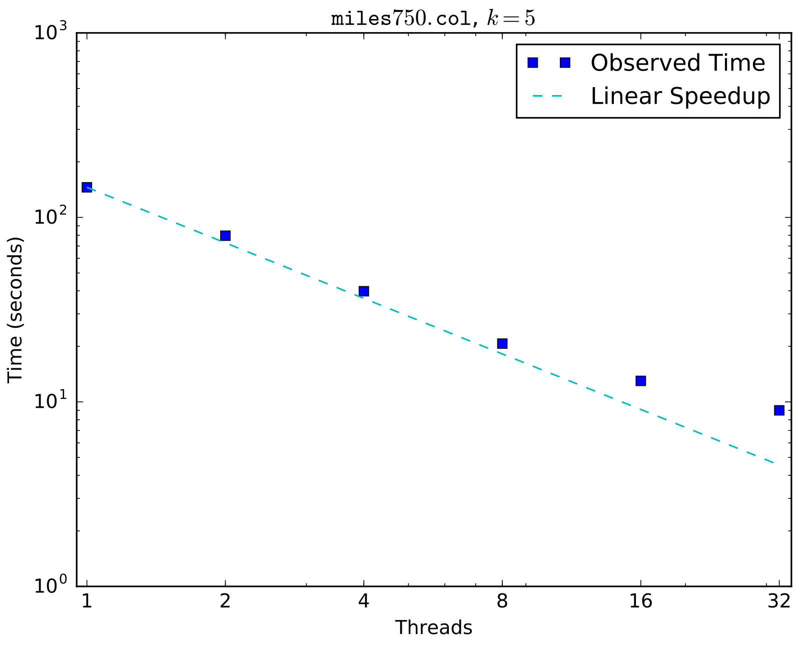
Figure 6: Scaling on various problems



(continued)

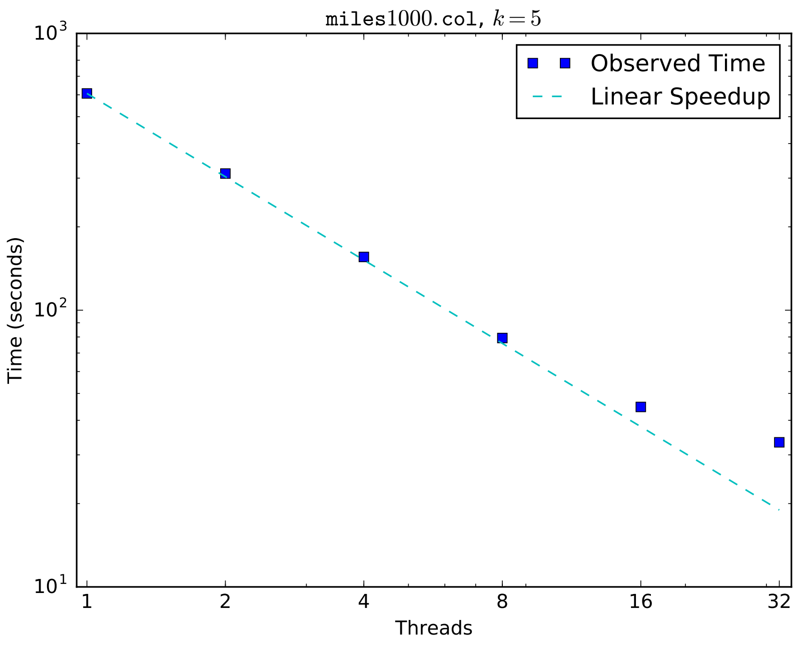
Figure 6: Scaling on various problems, continued

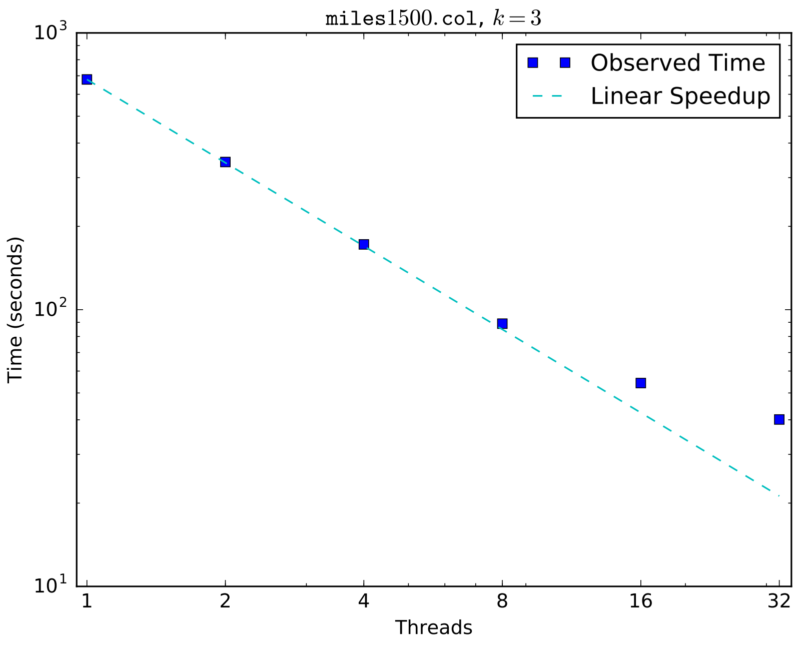




(continued)

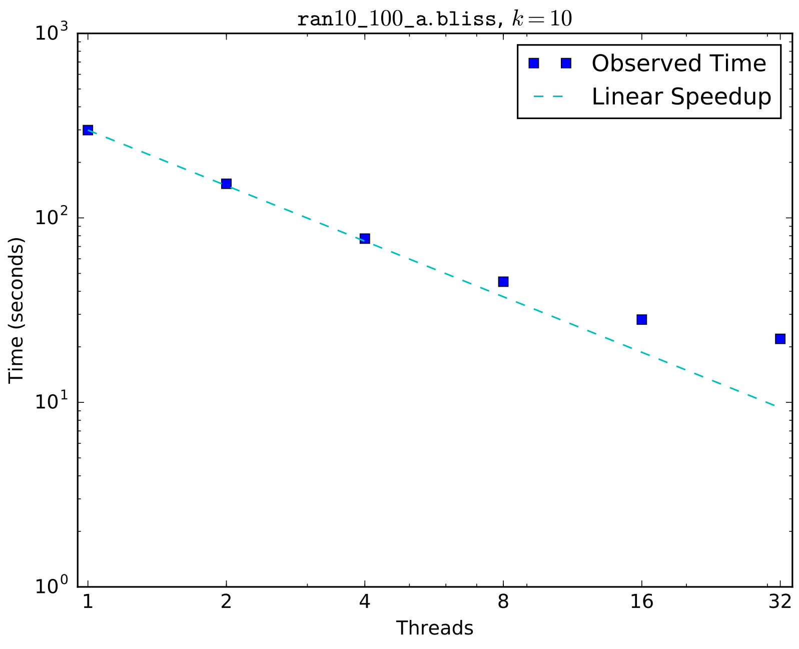
Figure 6: Scaling on various problems, continued

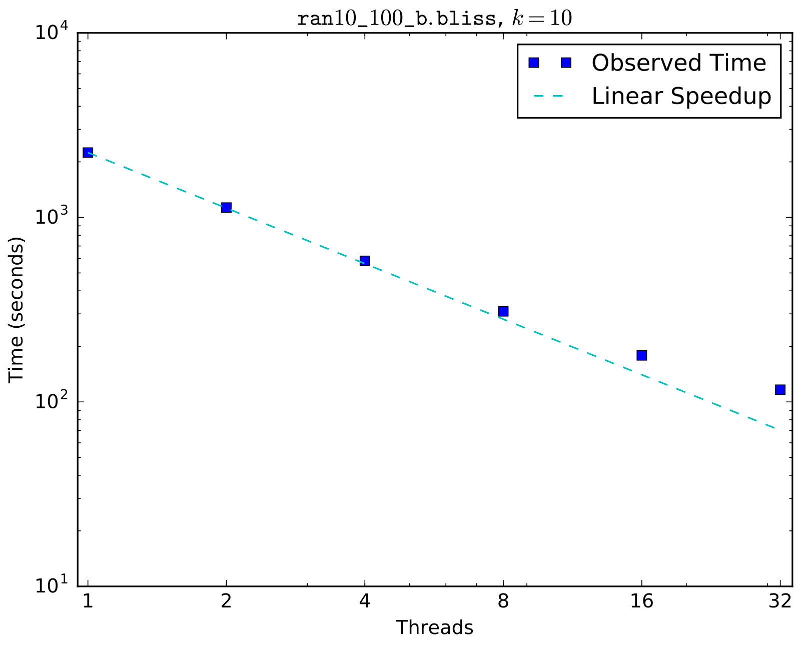




(continued)

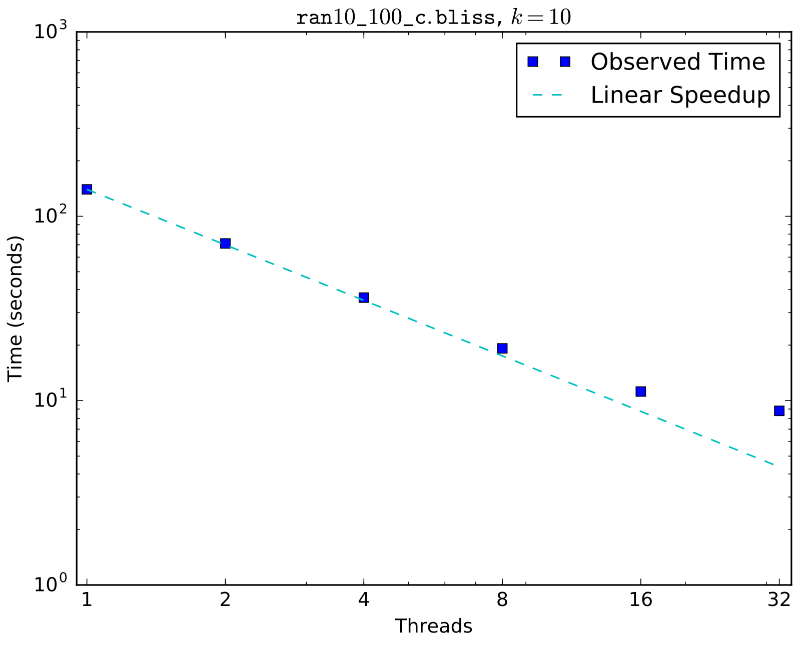
Figure 6: Scaling on various problems, continued





(continued)

Figure 6: Scaling on various problems, continued



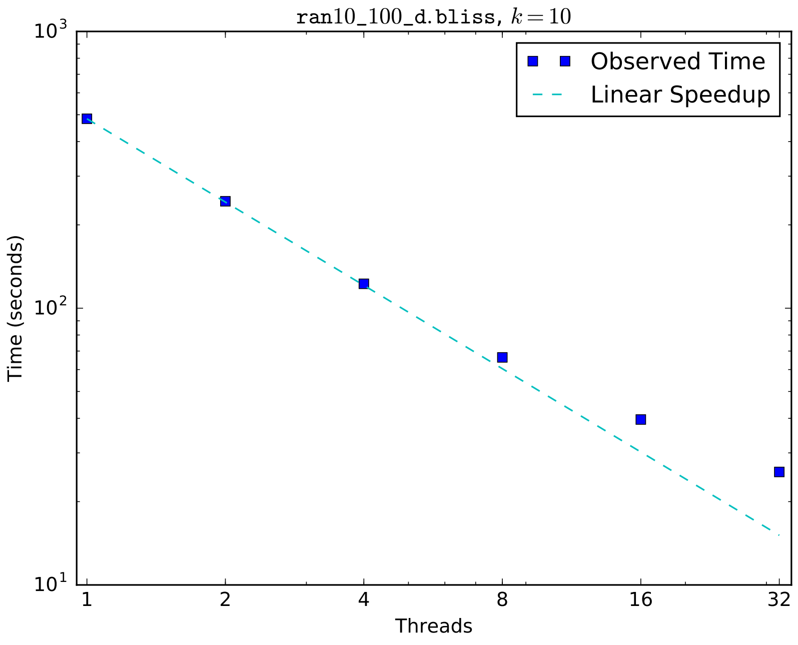


Table 7: *edgeUse* branching robustness, additional instances

(a) `miles250.col`, $k = 5$

Rank	% of Nodes
1	76.85%
2	8.34%
3	0.73%
4	0.47%
5	0.28%
6	0.24%
7	0.19%
8	0.17%
9	0.15%
10	0.15%
11+	12.43%

(b) `miles500.col`, $k = 5$

Rank	% of Nodes
1	67.44%
2	1.32%
3	0.20%
4	0.28%
5	0.13%
6	0.19%
7	0.19%
8	0.28%
9	0.39%
10	0.36%
11+	29.21%

(c) `miles1000.col`, $k = 5$

Rank	% of Nodes
1	80.33%
2	0.44%
3	0.13%
4	0.23%
5	2.44%
6	3.64%
7	1.74%
8	0.76%
9	0.34%
10	0.54%
11+	9.40%

(d) `miles1500.col`, $k = 3$

Rank	% of Nodes
1	96.95%
2	0.58%
3	0.37%
4	0.12%
5	0.09%
6	0.11%
7	0.06%
8	0.05%
9	0.10%
10	0.15%
11+	1.42%

(e) `le450_5b.col`, $k = 18$

Rank	% of Nodes
1	84.48%
2	5.17%
3	3.45%
4	1.72%
5	3.45%
6	1.72%
7+	0.00%

(f) `le450_15b.col`, $k = 15$

Rank	% of Nodes
1	70.05%
2	0.32%
3	0.22%
4	0.39%
5	0.12%
6	0.05%
7	0.05%
8	0.06%
9	0.06%
10	0.06%
11+	28.62%

(g) `ran10_100_b.bliss`, $k = 10$

Rank	% of Nodes
1	61.30%
2	6.98%
3	2.19%
4	0.93%
5	0.79%
6	0.46%
7	0.95%
8	2.07%
9	2.68%
10	1.83%
11+	19.82%

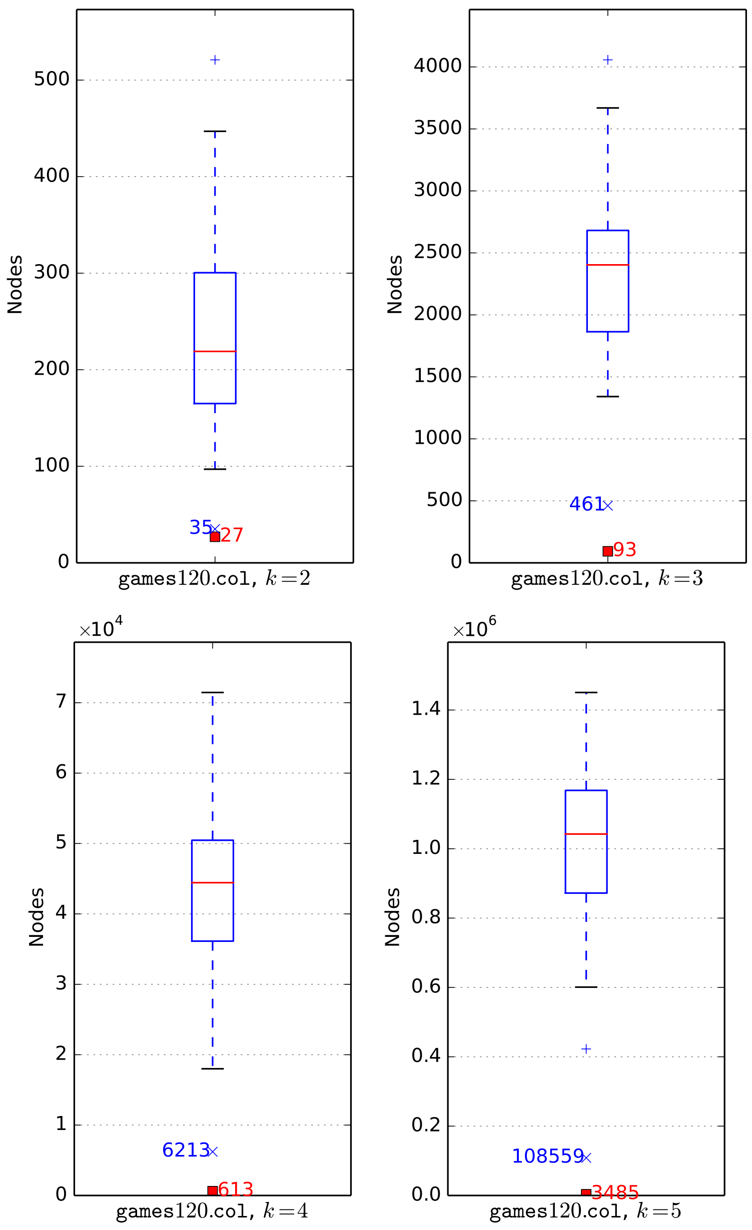
(h) `ran10_100_c.bliss`, $k = 10$

Rank	% of Nodes
1	60.92%
2	2.76%
3	1.32%
4	2.21%
5	0.60%
6	0.39%
7	0.14%
8	0.15%
9	0.17%
10	0.16%
11+	31.19%

(i) `ran10_100_d.bliss`, $k = 10$

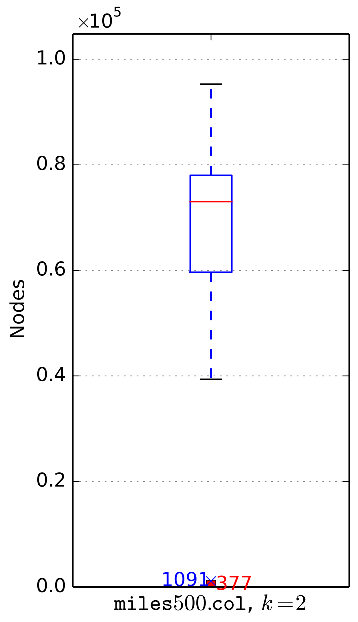
Rank	% of Nodes
1	79.29%
2	1.33%
3	0.52%
4	0.15%
5	0.60%
6	4.66%
7	0.60%
8	1.69%
9	2.92%
10	0.09%
11+	8.15%

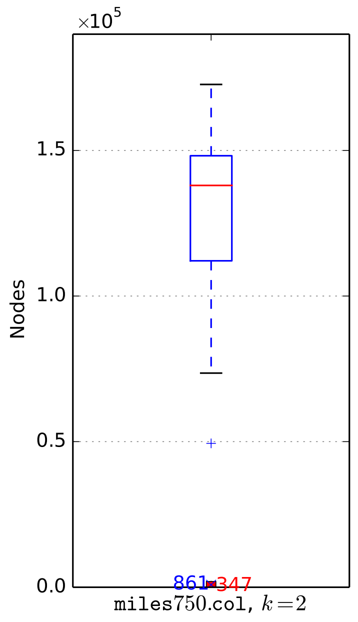
Figure 7: Random branching (box plot) vs. *edgeUse* branching (🟥, number of nodes on right) vs. local branching ($\times$, number of nodes on left)

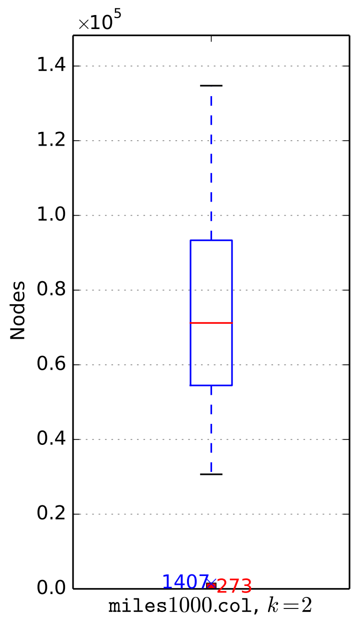


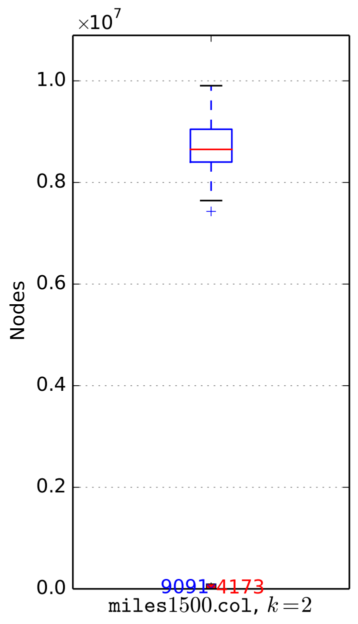
(continued)

Figure 7: Random branching (box plot) vs. *edgeUse* branching (🟥, number of nodes on right) vs. local branching (×, number of nodes on left), continued



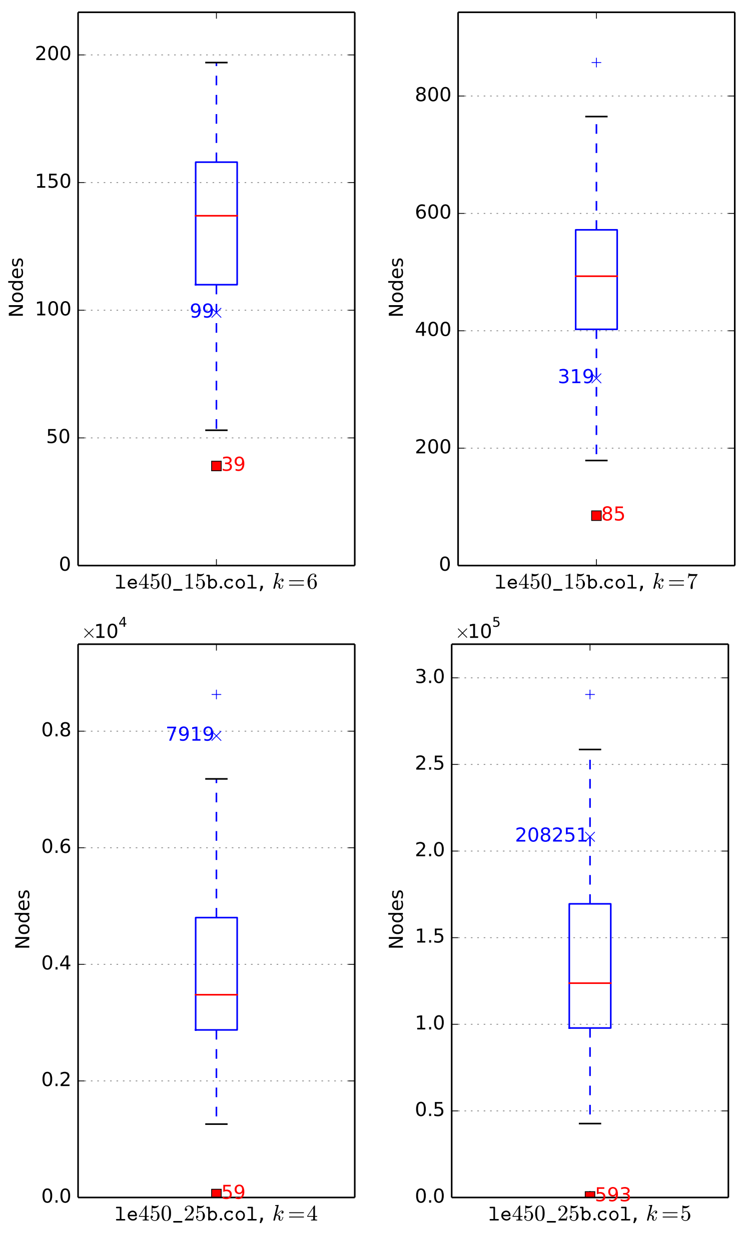






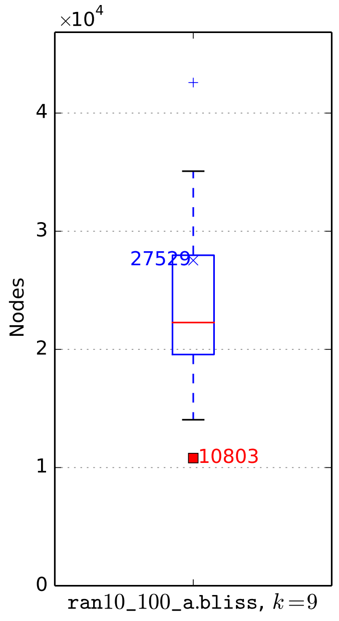
(continued)

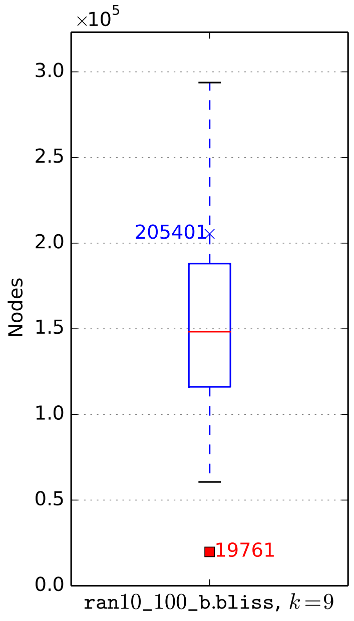
Figure 7: Random branching (box plot) vs. *edgeUse* branching (🟥, number of nodes on right) vs. local branching (×, number of nodes on left), continued

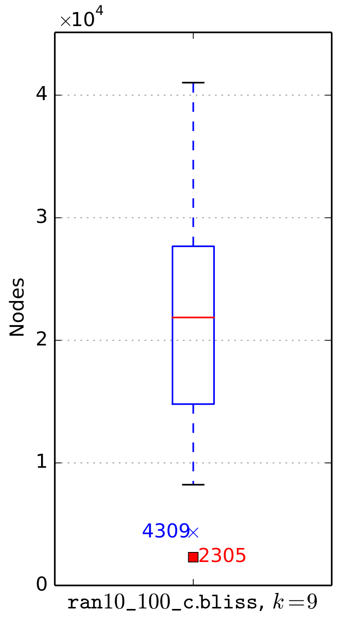


(continued)

Figure 7: Random branching (box plot) vs. *edgeUse* branching (🟥, number of nodes on right) vs. local branching (×, number of nodes on left), continued







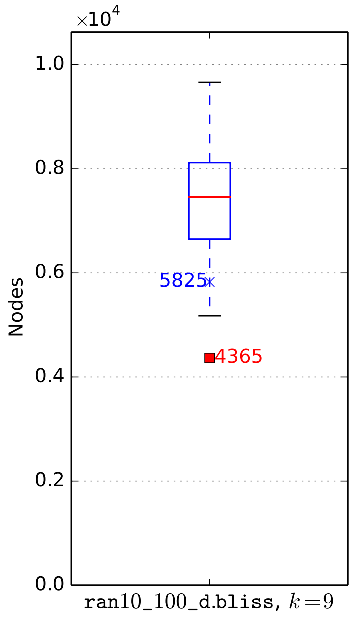


Table 8: Computational Results – three children per node

games120.col $n = 120$ $e = 638$

k	0	1	2	3	4	5	6	7	8	9
γ_k^G	119	118	117	114	113	112	112	111	111	112*
seconds	0.0	0.0	0.1	0.2	0.3	0.4	2.4	28.5	938.7	†

miles250.col $n = 128$ $e = 387$

k	0	1	2	3	4	5	6	7
γ_k^G	108	106	104	102	100	99	97	98*
seconds	0.0	0.1	0.2	0.7	3.1	55.8	852.7	†

miles500.col $n = 128$ $e = 1170$

k	0	1	2	3	4	5	6	7	8
γ_k^G	114	113	111	110	109	108	107	108*	107*
seconds	0.0	0.1	0.1	0.3	1.6	16.3	212.0	†	†

miles750.col $n = 128$ $e = 2113$

k	0	1	2	3	4	5	6	7	8
γ_k^G	122	121	120	119	118	117	116	115	115*
seconds	0.0	0.1	0.2	0.4	1.5	9.3	99.4	872.3	†

miles1000.col $n = 128$ $e = 3216$

k	0	1	2	3	4	5	6	7
γ_k^G	123	122	121	120	119	118	118	117*
seconds	0.0	0.1	0.2	0.5	2.0	29.9	422.6	†

miles1500.col $n = 128$ $e = 5198$

k	0	1	2	3	4	5
γ_k^G	102	101	100	99	98	97*
seconds	0.0	0.7	1.7	24.5	586.6	†

le450_5b.col $n = 450$ $e = 5734$

k	0	13	14	15	16	17	18	19
γ_k^G	450	450	450	450	450	450	450	449
seconds	0.0	1.1	1.3	1.6	1.8	5.1	387.6	2847

le450_15b.col $n = 450$ $e = 8169$

k	0	1	2	4	5	7	8	10	11	14	15	16	17
γ_k^G	450	450	449	449	448	448	447	447	446	446	445	446*	447*
seconds	0.0	0.1	0.2	0.4	0.6	0.7	0.8	1.1	1.3	47.2	176.3	†	†

le450_25b.col $n = 450$ $e = 8263$

k	0	1	2	3	4	5	6	7	8	9	10
γ_k^G	450	450	449	449	449	448	448	447	447	447	448*
seconds	0.0	0.1	0.2	0.4	0.6	0.6	1.1	2.0	13.9	129.4	†

ran10_100_a.bliss $n = 100$ $e = 502$

k	0	6	7	8	9	10	11	12
γ_k^G	100	100	99	99	99	99	99	100*
seconds	0.0	0.0	0.1	0.4	1.9	14.0	342.9	†

ran10_100_b.bliss $n = 100$ $e = 464$

k	0	3	4	7	8	9	10	11	12
γ_k^G	100	100	99	99	99	99	98	98	100*
seconds	0.0	0.0	0.1	0.2	2.0	6.8	96.4	1644	†

ran10_100_c.bliss $n = 100$ $e = 525$

k	0	5	6	7	8	9	10	11	12	13
γ_k^G	100	100	99	99	99	99	99	98	98	100*
seconds	0.0	0.0	0.1	0.1	0.4	1.9	7.2	101.6	1561	†

ran10_100_d.bliss $n = 100$ $e = 514$

k	0	1	7	8	9	10	11	12
γ_k^G	100	100	100	99	99	99	99	100*
seconds	0.0	0.0	0.1	0.6	4.5	22.1	466.0	†