EarSearch User Guide

Version 1.0

Derrick Stolee
University of Nebraska-Lincoln
`s-dstolee1@math.unl.edu`

April 14, 2011

Abstract

The Ear Search program implements isomorph-free generation of 2-connected graphs by ear augmentations. This document describes the interfaces used for customized searches, as well as describes three example searches: unique saturation, edge reconstruction, and extremal graphs with a fixed number of perfect matchings.

1 Introduction

The EarSearch library implements the generation algorithm of [7] to generate families of 2-connected graphs. It is based on the TreeSearch library [9]. The class `EarSearchManager` extends the class `SearchManager` and manages the search tree, using ear augmentations to generate children. It automates the canonical deletion selection in order to remove isomorphs.

2 Data Management

2.1 Graphs

Graphs are stored using the `sparsegraph` structure from the `nauty` library.

During the course of computation, these graphs are modified using edge and vertex deletions. To delete the ith vertex, set the `v` array to -1 in the ith position. To delete the edge between the i and j vertices, set the `e` array to -1 in two places: in the list of neighbors for i where j was listed and in the list of neighbors for j where i was listed. To place the vertices or edges back, place the previous values into those places.

2.2 Augmentations and Labels

The labels for each augmentation use two 32-bit integers. The first is the order of the augmented ear. The second is the index of the pair orbit which is used for the endpoints of the ear.

2.3 `EarNode`

Each level of the search tree is stored in a stack, where all data is stored in an `EarNode` object. All of the members of `EarNode` are public, in order to easily add data structures and flags that are necessary for each application. All pointers are initialized to 0 in the constructor and are checked to be non-zero before freeing up any memory in the destructor.

The core data necessary for `EarSearchManager` is stored in the following members:

- ear_length – the length of the augmented ear.
- ear – the byte-array description of the augmented ear.
- num_ears – the number of ears in the graph.
- ear_list – the list of ears in the graph (-1 terminated).
- graph – the graph at this node.
- max_verts – the maximum number of vertices in all supergraphs. Default to max_n from `EarSearchManager`.
- reconstructible – TRUE if detectably reconstructible
- numPairOrbits – the number of pair orbits for this graph.
- orbitList – the list of orbits, in a an array of arrays. Each array `orbitList[i]` contains pair-indices for pairs in orbit and is terminated by `-1`.
- canonicalLabels – the canonical labeling of the graph, stored as an integer array of values for each vertex
- solution_data – the data of a solution on this node.
- violatingPairs – A set of pair indices which cannot be endpoints of an ear.

3 Pruning

The interface `PruningAlgorithm` has an abstract method for pruning nodes of the search tree. The method `checkPrune` takes two `EarNode` objects: one for the parent and another for the child. Using this data, the method decides if no solution exists by augmenting beyond the child node. Since the pruning algorithm is called before the canonical deletion algorithm, this can also remove nodes which cannot possibly be canonical augmentations.

4 Canonical Deletion

The interface `EarDeletionAlgorithm` has an abstract method for finding a canonical ear deletion. The method `getCanonical` takes two `EarNode` objects for the parent and child and returns the array corresponding to the canonical ear. The `EarSearchManager` will determine if this canonical ear is in orbit with the augmented ear.

5 Solutions

The interface `SolutionChecker` is an abstract class which contains methods for finding solutions given a search node, storing the solution data, reporting on these solutions, and reporting application-specific statistics.

The method `isSolution` takes the parent, child, and depth and reports if there is a solution at the child node. It returns a non-null string if and only if there is a solution, and that string is a buffer containing the solution data. This buffer will be deallocated with `free()` by the `EarSearchManager`.

The method `writeStatisticsData()` returns a string of statistics (using the TreeSearch format) to be reported at the end of a job.

6 Example 0: 2-Connected Graphs

To enumerate all 2-connected graphs, the interfaces were implemented to only prune by number of vertices and possibly by number of edges. The search space is defined by three inputs: $N, e_{\min}$, and $e_{\max}$. These implementations are give by the following classes:

- `EnumeratePruner` will prune a graph if it has more than N vertices or more than $e_{\max}$ edges. Also, if $e(G) + (N - n(G) + 1) > e_{\max}$, it will prune since we cannot add the remaining $N - n(G)$ edges without surpassing $e_{\max}$ edges.
- `EnumerateDeleter` implements the default deletion algorithm: over all ears e in G so that $G - e$ is 2-connected, find one of minimum length, then use the canonical labels to select the canonical ear.
- `EnumerateChecker` detects "solutions" as any graph with exactly N vertices and between $e_{\min}$ and $e_{\max}$ edges.

7 Example 1: Unique Saturation

The input consists of two numbers r and N, and we are searching for uniquely K_r-saturated graphs of order N. The unique saturation problem utilizes the deletion algorithm in `EnumerateDeleter`, but adds some data to `EarNode` in order to track the constraints. The `SaturationAlgorithm` class implements both the `PruningAlgorithm` and `SolutionChecker` interfaces.

Note: The `SaturationAlgorithm` class is implemented only for $r \in \{4, 5, 6\}$ in order to use compiler optimizations for the nested loop structure.

7.1 Application-Specific Data

The following fields were added to `EarNode` for tracking constraints during the search. Most information is tracked in `adj_matrix_data`, which stores information as an adjacency matrix. The others are boolean flags which mark different properties of the current graph. These flags are set during the `checkPrune` method, and are accessed by the `isSolution` method.

- `adj_matrix_data` – Data on the (directed) edges. For unique saturation, this gives -1 for edges, and for non-edges counts the number of copies of H given by adding that edge. Values are in $\{0, 1, 2\}$, since when 2 is listed, then there are too many copies of H.
- `any_adj_zero` – A boolean flag: are any of the cells in adj_matrix_data zero?
- `any_adj_two` – A boolean flag: are any of the cells in adj_matrix_data at least two?
- `dom_vert` – A boolean flag: is there a dominating vertex?
- `copy_of_H` – A boolean flag: is there a copy of H?

8 Example 2: Edge Reconstruction

The Edge Reconstruction application takes an integer N and searches over all 2-connected graphs of order up to N and up to $1 + \log_2 N!$ edges. The deletion is built to make graphs with the same deck be siblings. Then, all siblings which are not detectably edge reconstructible are checked to have different edge decks.

The following three classes implement the interfaces:

- `ReconstructionPruner` implements the `PruningAlgorithm` interface and prunes any graph with more than N vertices or more than $1 + \log_2 N!$ edges.
- `ReconstructionDeleter` implements the `EarDeletionAlgorithm` interface and performs two different deletions:
    1. If the graph is detectably edge reconstructible, the deletion can be application-ignorant and utilizes the standard deletion algorithm from `EnumerateDeleter`.
    2. If the graph is NOT detectably edge reconstructible, the canonical ear is selected by using only the edge deck. Further, if the deletion is canonical, the graph is stored in the parent `EarNode` for later comparison of edge decks. The `GraphData` class was implemented specifically for storing these children within the parent `EarNode`.
- `ReconstructionChecker` implements the `SolutionChecker` interface and compares the current graph's edge deck against all previous siblings. This is done using three levels of comparison, which are implemented in the `GraphData` class.

8.1 Application-Specific Data

The `GraphData` class stores all information for a child graph. It implements three levels of comparison, which are checked in order within the `compare` method.

1. `computeDegSeq` computes and stores the standard degree sequence for the current graph.
2. `computeInvariant` calculates and stores a more complicated function based on the degree sequence and the degrees of the neighborhood for each vertex.
3. `computeCanonStrings` computes canonical strings for every edge-deleted subgraph and sorts the list. These are then compared, card-for-card.

In order to store these `GraphData` objects, the following members were added to the `EarNode` class:

- child_data – the GraphData objects for immediate children, used for pairwise comparison.
- num_child_data – the number of GraphData objects currently filling the data.
- size_child_data – the number of pointers currently allocated.

9 Example 3: p-Extremal Graphs

This problem is investigated in [8] and is the most involved of all applications. See [1] and [3] for background on this problem. The input is given as $P_{\min}$, $P_{\max}$, C, and N. The search is for elementary graphs with p perfect matchings (for $P_{\min} \leq p \leq P_{\max}$) with excess at least C and at most N vertices. The search actually runs over 1-extendable and almost 1-extendable graphs, which are the graphs reachable by the ear augmentations. A second stage adds forbidden edges to maximize excess without increasing the number of perfect matchings.

The following classes implement the EarSearch interfaces:

- `MatchingPruner` implements the `PruningAlgorithm` interface. Graphs are pruned for three reasons:

1. There are an odd number of vertices. By the Lovász Two Ear Theorem, we know that every ear augmentation has an even number of internal vertices.
2. There are more than $P_{\max}$ perfect matchings.
3. The parent graph was not 1-extendable, and neither is the current graph. By the Lovász Two Ear Theorem, we can always go from 1-extendable to 1-extendable using at most two ear augmentations.
4. Let c be the maximum excess of an elementary supergraph of the current graph, which is of order n, and let p be the current number of perfect matchings. If $c + 2(P_{\max} - p) - \frac{1}{4}(n' - n)(n - 2) < C$ for all $n \leq n' \leq N$, then prune. Otherwise, maximize the n' so that the inequality $c + 2(P_{\max} - p) - \frac{1}{4}(n' - n)(n - 2) \geq C$ holds. That value of n' is then used to bound the length of future ear augmentations, since no graph reachable from the current graph can have excess at least C and more than n' vertices.

In addition to pruning, the pruning algorithm also performs the on-line algorithm for updating the list of barriers by using the current ear augmentation.

- `MatchingChecker` implements the `SolutionChecker` interface. Given a 1-extendable graph with between $P_{\min}$ and $P_{\max}$ perfect matchings, forbidden edges are added in all possible ways and the elementary supergraphs with excess at least C are printed to output. If any are found, the `isSolution` method returns with success. The algorithm for enumerating all elementary supergraphs is implemented in the `BarrierSearch.cpp` file.

- `MatchingDeleter` implements the `EarDeletionAlgorithm` interface. The following sequence of choices describe the method for selecting a canonical ear to delete from a graph H:

  1. If H is almost 1-extendable, we need to delete an ear e' so that $H - e'$ is 1-extendable. By the definition of almost 1-extendable, there is a unique such choice.
  2. If H is 1-extendable, check if there exists an ear e' so that $H - e'$ is 1-extendable. If one exists, select one of minimum length and break ties using the canonical labels of the endpoints.
  3. If H is 1-extendable and no single ear e' makes $H - e'$ 1-extendable, then find an ear e so that there is a disjoint ear f with $H - e$ is almost 1-extendable and $H - e - f$ is 1-extendable. Out of these choices for e, choose one of minimum length and break ties using the canonical labels of the endpoints.

9.1 Application-Specific Data

The following members were added to `EarNode` to help the perfect matchings application.

- `extendable` – A boolean flag: is the graph 1-extendable?
- `numMatchings` – The number of perfect matchings for this graph.
- `barriers` – The list of barriers of the graph, given as an array of `Set` pointers. This barrier list is updated at each level by an on-line algorithm.
- `num_barriers` – the number of barriers in the graph.

9.2 Perfect Matching Algorithms

There are a few algorithms that are implemented in order to solve certain sub-problems, such as counting perfect matchings or enumerating independent sets. These are computationally complex problems, but the implementations are very fast for these small instances. The algorithms are mostly un-optimized and rely on simple instructions and low overhead in order to be run many many times during the course of the search.

- `countPM`(G, P) counts the number of perfect matchings in a graph G, with an upper bound of P. It operates recursively, selecting an edge e in G and attempts to extend the current matching using e and not using e. When a perfect matching is found, the counter increases. There are two shortcutting strategies:

  1. If there is ever a vertex with no available edges, the recursion is halted with a count of zero perfect matchings, since the current matching does not extend to a perfect matching.
  2. If the current count of perfect matchings ever surpasses P, then the current value is returned. During the search, we only care about graphs with at most $P_{\max}$ perfect matchings, so graphs with many more will only be pruned.

- `isExtendable`(G) tests if the given graph is 1-extendable. This is done by storing an array of boolean flags for each edge, marking each as they are found to be in perfect matchings. This algorithm is explicitly used in the deletion algorithm. During the pruning algorithm, where a specific augmentation is given, we can detect 1-extendability by asking if there is a perfect matching using the proper alternating path within the augmented ear.

- `enumerateAllBarrierExtensions`$(G, \mathcal{B}, C)$ and `searchAllBarrierExtensions`$(G, \mathcal{B})$ are two methods which take a 1-extendable graph G with barrier list $\mathcal{B}$ and attempts to add forbidden edges to G to attain the maximum excess. The difference is that `enumerateAllBarrierExtensions` will output any graphs with excess at least C, while `searchAllBarrierExtensions` will simply return the largest excess. The algorithm essentially enumerates all independent sets within the barrier conflict graph $\mathcal{B}$, where conflicts are computed on the fly. The enumeration is recursive, simply testing if the next available barrier should be added to the current independent set. As each set is added, it tests which barriers with larger index are in conflict with this graph. These barriers are then not considered in deeper recursive calls. Due to the low overhead for each independent set, this simple algorithm runs fast enough for the search to be feasible.

References

[1] A. Dudek and J. Schmidt. On extremal graphs with a constant number of 1-factors. submitted. 2010.

[2] S. G. Hartke and A. J. Radcliffe. Mckay's canonical graph labeling algorithm. In *Communicating Mathematics*, volume 479 of *Contemporary Mathematics*, 99111. 2009.

[3] S. G. Hartke, D. Stolee, D. B. West, and M. Yancey. On extremal graphs with a fixed number of perfect matchings. in preparation. 2011.

[4] B. D. McKay. Small graphs are reconstructible. *Australas. J. Combin.*, 15:123 126, 1997.

[5] B. D. McKay, Isomorph-free exhaustive generation *J. Algorithms*, 26(6):306-324. 1998.

[6] B. D. McKay, nauty user's guide (version 2.4) Dept. Computer Science, Astral. Nat. Univ., 2006.

[7] D. Stolee, Isomorph-free generation of 2-connected graphs with applications, in preparation, 2011.

[8] D. Stolee, Generating p-extremal graphs, in preparation, 2011.

[9] D. Stolee, TreeSearch user guide, available at http://www.github.com/derrickstolee/TreeSearch/ 2011.