Functional color coding applied to graph matching

DBMI Technical report Oct 2011

Staal A. Vinterbo and Jialan Que
Division of Biomedical informatics, UCSD
sav@ucsd.edu, jque@ucsd.edu

Abstract We present an inexact algorithm for finding a vertex assignment between general simple attributed graphs. While applicable to general graphs, the algorithm particularly targets complete graphs and is based on computing matches between random sub-trees of an *origin* graph to a *target* graph using a color coding algorithm. The computed sub-tree assignments together with their scores are combined to form a weighted bipartite graph between origin and target vertices. From this bipartite graph we compute a global assignment using the Hungarian algorithm, as well as derive beliefs in the correctness of the individual vertex correspondences. Computational experiments show that our algorithm significantly outperforms the SMAC graph matching algorithm by Cour et al. in terms of match quality on edge weighted complete graphs. Our implementation of color coding is based on the observation that color coding can be seen as a fold over trees. This results in a succinct declarative presentation of color coding for attributed trees that can easily be translated into efficient code with time complexity matching the best reported for this problem. Our color coding algorithm is also an example of graph matching as a catamorphism. Implementations in Python as well as OCaml are freely available under an open source licence.

Introduction

In the context of graph matching, the "assignment" or "correspondence" problem is the problem of finding a relation between the vertex sets of two graphs. Introducing directionality of the relation, we will say that the assignment assigns *origin* vertices to *target* vertices, and we will call the corresponding graphs the origin and target, respectively. The assignment problem is often characterized in terms of being "exact" or "inexact". Exact matching requires *edge preservation* from origin to target. It means that if two origin vertices are endpoints of an edge, corresponding origin vertices are connected by an edge as well. If this does not hold, the matching is said to be "inexact". If the assignment is injective, and the corresponding matching is exact, the assignment is a *monomorphism*. If the assignment is a monomorphism, and the induced subgraph has no extraneous edges, the assignment is called a *subgraph isomorphism*. If the assignment is a subgraph isomorphism and the target contains no extraneous vertices, the assignment is an *isomorphism*. Similarly to Cordella et al. [Cordella] we can further distinguish between *syntactic* and *semantic* properties of an assignment. Syntactic properties are essentially the properties we described so far, while semantic properties pertain to the attribute values of vertices and edges. The VF2 algorithm of Cordella et al. can be used to find (syntactic) subgraph isomorphisms for which the corresponding attributes are equal according to some defined criterion (the authors describe this in terms of rules for syntactic and semantic *feasibility*).

Whenever strutures can be described in terms of graphs, assignments can describe relationships between structures. In this sense, the assignment problem can be said to be a fundamental structural pattern recognition problem. Application ares of graph matching span image and video analysis, document processing (including hypertext documents), biometrics, biology and biomedicine. Conte et al [Conte] present an overview of many approaches and a taxonomy of algorithms.

Our approach is motivated by the need for an algorithm that deals with the case where both the source and target are simple, complete, and possibly undirected. Furthermore, graph attributes should be allowed to be points from an arbitrary space equipped with a similarity function. Unfortunately, this problem is equivalent to finding a minimum weight m-clique in a near to complete m,n-partite graph (the tensor product of the origin and target). This problem in turn is not polynomial-time approximable in a complete graph within 2^{n^k} for any fixed $k > 0$ unless P=NP [Mielikainen], and it is not clear that the near-completeness improves the situation. Furthermore, the lack of constraints with regards to attribute spaces makes many approaches that take attribute space properties into account less appropriate or even inapplicable. Examples are methods from fields such as computer vision that take the geometry of images into account [Torresani], and methods that require *semantic* compatibility between origin and target [Cordella]. For the latter, a problem is the requirement imposed by the propositional definition of feasibility. For example, the difference between edge weights of optimally corresponding edges in target and origin are not guaranteed to be smaller than an arbitrary number of non-corresponding ones.

Key to our approach is that while the general problem is hard, it is tractable for small bounded tree-width origins [Alon]. The hope is that optimal assignments computed from tractable origin sub-trees can be composed to produce a high quality assignment overall. A problem is that of *deception*, by which we mean that a sub-tree is deceptive if it has optimal assignments that are not (fully) contained in a globally optimal assignment. Our strategy for dealing with such deceptive sub-trees is to consider them "noise", and systematically sample random sub-trees from the origin for which we compute sub-assignments. Each individual correspondence (a vertex pair) in these sub-assignments is associated with the overall quality score of the sub-assignment and are collected to form a bipartite weighted graph B between the origin and target vertex sets. Finally, the Hungarian algorithm is applied to find a minimal cost matching in B, which is taken as the final overall assignment. This approach is *inexact* in that while it produces an injective assignment, it is not guaranteed that it is a monomorphism. We present experimental results showing that our approach presents a viable option.

While the combination of sub-problem solutions is not novel in the graph matching context, our contribution lies in

- a general graph matching algorithm applicable to simple vertex and edge annotated graphs
- a description of color coding in terms of a fold, allowing almost direct translation of the description into code, resulting in
- a functional algorithm that has time complexity matching the best reported, implementable in less than hundred lines of Python or OCaml code, and
- the use of the bipartite graph to compute a "belief" measure for each vertex correspondence,

- publicly available open source implementations of the algorithm in Python and OCaml.

Methods

If G is a graph, we let $V(G)$ denote the vertex set of G, and let $E(G)$ denote the edge set of V. Let G_1 and G_2 be two graphs. Let $w : E(G_1) \times V(G_2)^2 \to [0,1]$ be a similarity measure on edges, and let $v : V(G_1) \times V(G_2) \to [0,1]$ be a similarity measure on vertices. We can now formally state what we mean by the assignment problem as follows.

The Assignment Problem: Given G_1, G_2, w, and v, find injective $f : V(G_1) \to V(G_2)$ such that (1):

$$\sum_{a \in V(G_1)} v(a, f(a)) + \sum_{(a,b) \in E(G_1)} w((a,b),(f(a),f(b)))$$

is maximized.

If we create the *tensor product graph* W of G_1 and G_2, and assign weights to vertices in W using v and weights to edges in W using w, the assignment problem as defined above is equivalent to finding a maximum vertex and edge weight m-clique in W. As noted, this problem is not only NP-hard but also not polynomial-time approximable in a complete graph within 2^{n^k} for any fixed $k > 0$ unless P=NP [Mielikainen]. Whether near-completeness improves the situation, is not clear. Disregarding vertex weights for a moment and identifying W with its weighted adjacency matrix, the assignment problem can be formulated in terms of a quadratic optimization problem: find a binary mn length vector x such that:

$$x^T W x$$

is maximized subject to the condition that x represents an injection. Torresani et al. [Torresani] apply Pseudo-boolean optimization [Boros] to a formulation similar to the above one in order to address the assignment problem in the context of feature recognition in images. In general, applying algebraic machinery such as singular value decomposition and (semi-definite and quadratic) mathematical programming has been applied to the assignment problem. The main motivation for this is to create analytic approaches building on existing continuous optimization theory and methods as well as complementing the heuristics forced by the hardness of the problem. This is particularly true in image-based applications where the internal structure in an image is essentially Euclidian, and information stemming from this can be readily expressed in terms of mathematical programs.

Our motivating application however does not allow us to make such assumptions. The edge weights represent similarity, but no further structural assumptions are made. Furthermore, the graphs are also mainly complete, immediately disallowing any direct applications of standard methods for unweighted problem instances.

As stated, our approach is based on matching random origin subtrees by a color coding algorithm, and subsequently combining these subtree assignments into a global assignment. We now present the color coding algorithm, and the overall algorithm.

Color Coding By Folding

There are $\frac{n!}{(n-m)!} \in O(n^m)$ possible assignments that potentially need to be investigated in order to find the optimal one. Alon, Yuster, and Zwick [Alon] noted that if the vertices of G_2 are randomly colored using m colors, the probability p_c of a particular order m subgraph ending up colorful, i.e., all its vertices are uniquely colored, is $m!/m^m > e^{-m}$. Consequently, if the random coloring is repeated using multiple independent trials, for a particular m order subgraph to have been colorful in at least one trial with a probability at least $1-\varepsilon$, we must perform

$$q \geq \frac{\ln(\varepsilon)}{\ln(1-p_c)}$$

trials. Using that $\ln(1-x) < -x$ and $\ln(x) = -\ln(1/x)$ for $0 < x < 1$, we get that

$$\frac{\ln(\varepsilon)}{\ln(1-p_c)} < \frac{\ln(\varepsilon)}{-p_c} < \frac{\ln(\varepsilon)}{-e^{-m}} = e^m \ln(\frac{1}{\varepsilon}).$$

Hence, if we perform $q \geq e^m \ln(1/\varepsilon)$ trials, any particular order m subgraph of G_2 has been colorful at least once with probability at least $1-\varepsilon$.

Furthermore, Alon et al. showed that if G_1 is of bounded threewidth, an isomorphism (if it exists) with a colored subgraph in G_2 can be found in time $O(2^m p)$ where p is a polynomial in the size of the input. This means that if we consider m and ε constant, i.e., not a part of the input, we can find an assignment (if it exists) from G_1 to G_2 with probability $1-\varepsilon$ in polynomial time $O((2e)^m \ln(1/\varepsilon)p)$. Alon et al. also present a derandomization of the algorithm, showing that the subgraph isomorphism problem for bounded treewidth graphs G_1 is *fixed parameter tractable* [Downey].

However, the (constant) factor $(2e)^m$ makes this color coding approach practical only for relatively small m. As an example, Dost et al. [Dost] report experimental results for m of size 11 in their presentation a weighted (and inexact) version of the randomized color coding for matching tree structured pathways in protein interaction networks.

Details of our implementation Consider the graph G and the path T given here:

```
               0
               A
          _.-'
G:     1 B
         |'.         E 3
         |  `.      ,
       2 C.._ `.  ,'
             ``-D'
                2

T:       a -> b -> c -> d
```

Each vertex in G has a color in $\{0,1,2,3\}$ which is indicated next to the vertex. Note that we in the following associate with each vertex a value that is a vertex identifier, and hence unique. We model attributes of vertices (and edges) by accessor functions taking the identity as input and yielding the value as output. We associate each subgraph of G with its *colorset*, i.e., the set of colors assigned to its vertices. We say that subgraph of G is *colorful* if there are no two vertices in the subgraph that share a color, i.e., the cardinality of its colorset is equal to its order. We now want to find a colorful match for the directed path T of length $m = 4$ in G, in other words we want a vertex assignment from T to a colorful subgraph of G. By inspection we see that there are two possibilities, the colorful path ABDE in G, or the reverse of this, starting at E. These are the only possibilities because C and D have the same color 2. Now, consider the case where we have matched the k length path (a,b,c) and wish to extend this with a match for d. The following table shows the possibilities.

Table 1: Possible colorful match extensions of (a,b,c) to (a,b,c,d)

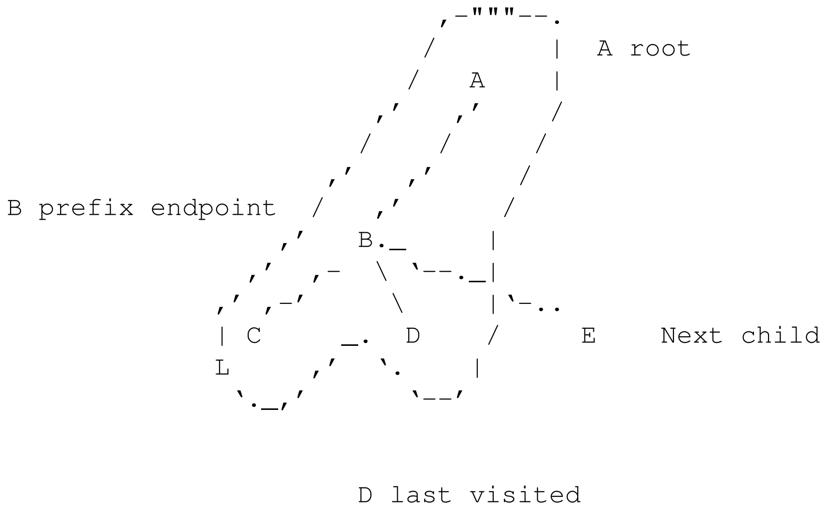
a	**b**	**c**	**d**
A	B	C	/
A	B	D	E
B	D	E	/
C	B	A	/
D	B	A	/
E	D	B	A

If we are only interested in answering whether there exists a match, we only have to store the colors and the prefix endpoint (the vertex to which the next vertex will be connected) of any matched k length subpath in G.

Now consider each extension of a match having a cost depending on which edge in G is used (independently from previous edges used), and instead of wanting to determine whether a match exists, we want the cheapest match. Again it is enough to store the colorset and the prefix endpoint, but with the caveat that we also need to store information that lets us recover a cheapest path corresponding with a given colorset and endpoint. One way to do this is to also store at each step the value of the cheapest prefix as well as the prefix itself. Alternatively, Huffner [Huffner] proposes storing the order of the colors instead of the prefix itself, and then recover the sequence at the end by backtracking. Storing each entry in the prefix costs $\log(n)$ bits, while storing each entry in the color order costs $\log(m)$ bits, representing savings in storage space at the cost of the recovery.

The above approach of path extension can be thought of as processing of the ordered list of vertices in T much like we would compute the sum of integers in a list using a binary operation. At each step a vertex from T is combined with a table to produce a table. The result is then read out of the final table. Considering that a list is a recursive data structure, this pattern of recursion is called a *fold* or "reduction" of the list.

In the path matching approach above we extended a prefix of length i with endpoint u matched to T vertex q_i by considering all edges (u,v) in G as a match for edge (q_i, q_{i+1}) in T. If we think in terms of edges instead of vertices, the fold can equivalently be expressed as a fold over the edges of T in pre-order. For the path problem, we know that each edge to be added originates (is incident to) at the last vertex added. This is not so in general for trees. The next edge origin might be any already visited vertex. Consider the tree G_1 with vertices A,B,C,D,E below:



Assume we have already visited vertices A, B, C, and D in pre-order, indicated by the drawn envelope of this subtree. The vertex D is the last visited vertex, A is the root, while the next parent is B representing the prefix endpoint. All the candidate colorful trees in G_2 are extended at the point corresponding to B adding an edge to a point which will be put in correspondence with E. Consequently, all trees sharing the same colorset and prefix endpoint also share the same possible extensions, which means that we only need to store information on one tree for each possible (colorset, endpoint) pair in order to determine whether there exists a tree match or compute the minimum cost match. In the following we will use a mapping data structure "table" that maps a (colorset, vertex) pair to stored information.

In the following we turn to details of how we implement the above ideas. We write function applications and definitions much like it is done in the functional programming languages OCaml and Haskell. Let `(h::t)` denote a list that has `h` as a first element and the list `t` containing the rest of the elements. The infix list construction operator `::` is often called "cons" and we could have equivalently defined `(h::t)` in terms of a prefix application like `(::) h t` when we use the convention that we can convert an infix operator into a prefix function by enclosing it in parentheses. Also let `[]` denote the empty list. The list containing elements 1,2, and 3 in that order can then be written as `(1::2::3::[])` or `(::) 1 ((::) 2 ((::) 3 []))`. Now recall that `(+) x y = x + y`, and let us substitute `(::)` in the above list with `(+)`, and `[]` with `z = 0`. Then we get `(+) 1 ((+) 2 ((+) 3 z)) = (1 + (2 + (3 + 0)))`. This substitution of function calls (and `z`) for type constructors in recursively defined data types is called a *fold*. For lists as we defined them, we can define a `fold` function as follows:

```
fold g z []     = z
fold g z (h::t) = g h (fold g z t)
```

We can get the above fold example by letting `g = (+)` and `z = 0`, i.e., `fold (+) 0 (1::2::3::[])`.

Using a fold, we can define standard list processing functions in term of fold as:

```
map f list = fold (x t -> (f x)::t) [] list
filter p list = fold (x t -> if p x then x::t else t) [] list
concat l1 l2 = fold (::) l2 l1
```

where (arguments -> expression) means the unnamed function that when given arguments arguments returns the value of expression. The function map applies the function f to all elements in list in turn and returns the list of results, filter takes a predicate p and a list list and returns the sub-list of elements in list for which p is true, while concat concatenates two lists. Concatenation of lists is for conveniance often denoted by an infix operator. We will use ++, i.e., a ++ b = concat a b.

Of particular interest for us later is the foldl function:

```
foldl f z []     = z
foldl f z (h::t) = foldl (f z h) t
```

This function is called a *left* fold in that it instead of processing a list in a right to left manner, processes it in a left to right manner instead. So foldl (+) 0 (1::2::3::[]) is ((0 + 1) + 2) + 3. We can define foldl in terms of fold as:

```
foldl f z l = fold (x g -> (a -> g (f a x))) (x -> x) l z
```

As Hutton [Hutton] points out, while programs written using fold can be less readable, they can be constructed systematically and can more easily be analyzed and transformed. This, and the expressive power of using folds, are reasons why we in the following express color coding in terms of folds.

Returning to the path matching example, we now see that if we have an operation g that takes as input a table and a vertex and returns a table, we can compute the final table t as:

```
finaltable vertices = foldl g emptytable vertices
```

For path matching, we know that the prefix endpoint is the latest seen vertex, hence vertices can be a simple ordered list of vertices. In the tree case, as we have discussed above, this assumption does not hold and vertices must contain information about which vertex is the endpoint for each prefix.

Now consider a recursively defined data type Tree that consists of vertices Vertex v s where v is the vertex label and s is a list of subtree root vertices. Note that this data type has two constructors, Vertex for creating the (label, subtree) tuple, and "cons" or (::) for creating the subtree list. Similarly to fold for lists, we can define a fold on trees by substituting functions for the type constructors (and again an initial element z for [] in the subtree list signifying that Vertex label [] is a leaf):

```
treefold  f g z (Vertex label subtrees) =
  f label (treefolds f g z subtrees)
treefolds f g z []     = z
treefolds f g z (h::t) = g (treefold f g z h) (treefolds f g z t)
```

The function f takes the place of the Vertex constructor, the function g takes the place of "cons" in the subtree list, and z the place of []. The auxiliary function subtrees is essentially a fold over the subtree list. We can define the following tree processing functions using treefold:

```
preorder tree = treefold (::) (++) [] tree
order tree = treefold (y x -> x + 1) (+) 0 tree
```

Here, preorder returns the list of vertex labels in preorder, while order returns the number of vertices in the tree.

An ordered tree, which our Tree data type represents, is isomorphic to its pre-order edge list. We can obtain this list using treefold as:

```
f label s = (label, fold ((i, l) res -> (label, i)::(l ++ res)) [] s)
preedges tree = snd (treefold f (::) [] tree)
```

where snd (a,b) = b. As a path is isomorphic to its preorder vertex list, and we could fold over this in the path matching example above, we can now analogously fold over the pre-order edge list when matching trees:

```
finaltable tree = foldl g emptytable (preedges tree)
```

We now turn to the question of how to implement the binary operation g we want to use in the fold over pre-order edge list. To this end let a (sub-)tree in G be stored in a structure that has the information retrieval functions colorset, endpoint, score defined on it. Also, let addvertex tree edge v add the vertex v to the structure instance tree and update information stored in it accordingly using that the current edge in the origin tree is edge. Also let init v produce a structure containing only the root v. Of the above functions, the addvertex function needs to have access to information from which it can compute the new colorset and endpoint of the tree, as well as the score. We will not address this function in further detail. Also let empty denote an empty tree structure.

As discussed above, we need to be able to store trees indexed by their colorsets and endpoints. Let *table* be a generic map type from (colorset, endpoint) keys to trees. Associated with the map type are functions items which yields a sequence (e.g., a list) of elements (trees) stored in the map instance, update which, given a tree, stores the tree at its (colorset, endpoint) entry either if there is no tree there already, or if the already stored tree has a lower score, and new () which creates a new empty store. Also, let the function max extract a maximum score tree from the store.

We can now define a function that takes a tree and an augmented edge and produces all colorful extensions of this tree as:

```
extend edge tree =
  [addvertex tree edge v | v <- successors G (endpoint tree) and
                          (color v) not in (colorset tree) ]
```

The notation [expr | v <- gen and pred] is a *list comprehension* and means the list of all values of expression expr for all elements v generated by expression gen for which expression pred evaluates to true. Functions successor G u yields all v such that there is an edge (u, v) in graph G, and color v yields the color associated with vertex v in G. We can now define the operation g that takes as input a *table* instance and an augmented edge and returns a *table* instance as:

```
g table edge =
  let op table' tree = foldl update table' (extend edge tree) in
  foldl op (new ()) (items table)
```

The function g works by iteratively extending each tree in the given table into a list of trees from which an initially empty table is populated. We can then now encode a trial of color coding as:

```
trial tree =
  let table = foldl update (new ()) [init u | u <- (vertices G)] in
  max (foldl g table (preedges tree))
```

The trial function first creates an initial table containing only single vertex trees, one for each vertex in G. Then it folds g over the augmented edge list computed from tree creating a final table from which a maximum score tree is extracted and returned. Once enough trials have been performed, the best matching tree can be translated into an assignment.

In fact, any subtree produced during the execution of trial can be translated into a sub-assignment. In fact, we can consider sub-trees as sub-assignments and vice versa. Now, given a sub-assignment f' of with value σ' we could decide if f' could be extended to an optimum assignment f with value σ^* if we knew the largest possible value σ'' associated with this extension. If we knew this, we could at any stage in the algorithm, and in the computation of g in particular, prune away subassignments that would never be extendable to an optimum assignment, i.e., prune f' if $\sigma' + \sigma'' < \sigma^*$. The time and space savings of this would be significant! Note that while σ'' and σ^* are in general not available, any approximation $\sigma''_\sim$ and $\sigma^*_\sim$ such that $\sigma''_\sim \geq \sigma''$ and $\sigma^*_\sim \leq \sigma^*$ can be used. For $\sigma^*_\sim$ we can at any time use the best encountered value so far. For the subassignment f' of size $k > 0$ we know that the extension to f involves $m-k$ edges and $m-k$ vertices. We can thus find an upper bound $\eta_i + \alpha_i$ for σ'' by computing $w(e, e'')$ for $(e, e'') \in E(G_1) \times E(G_2)$ and computing η_i as the sum of the i largest of these, and computing $v(a, b)$ for $(a, b) \in V(G_1) \times V(G_2)$ and letting α_i be the sum of the i largest of these. We then perform pruning by setting $\sigma''_\sim = \eta_{m-k} + \alpha_{m-k}$ when considering f' of size k. Alternatively, as Huffner et al. [Huffner] suggest for their their path matching algorithm, we can exhaustively compute σ_i for $i \leq l$ for small l, and compute σ_i for $i > l$ as the smallest sum of σ_j such that $j \leq l$ and $\sum_j j = i$. Given a function keep tree that returns false if the sub-assignment associated with tree cannot be extended to full assignment with a score that exceeds the currently best known $\sigma''_\sim$, we can perform the filtering by inserting a call to keep into extend as:

```
extend edge tree =
  filter keep [addvertex tree edge v
               | v <- successors G (endpoint tree)
                 and ((color v) not in (colorset tree)]
```

Trial Time Complexity Analysis In the above algorithm, when processing edge i (1-indexed) we are processing the trees in a map data structure that maps colorsets of length i and a prefix endpoint to trees. Consequently, we are processing at most $n\binom{m}{i}$ trees for edge i. Now consider a particular tree with colorset C and endpoint u. This particular prefix can only be extended with vertices that are adjacent to u in G. This means that we for all trees in the map with colorset C, we only need to consider edges in $(u, v) \in E(G)$ for which u is an actually occurring prefix endpoint. In short, for each colorset C we need to consider at most $|E(G)|$ possible extensions, making the total number of extensions we need to process $|E(G)|\binom{m}{i}$ for edge i. Since we are extending as many times as there are edges in the tree, each time with a colorset size that has increased by one, we get that we are processing at most

$$\sum_{i=1}^{m-1} |E(G)| \binom{m}{i} \in O(|E(G)| 2^m)$$

extensions. We now assume that computing the score of an extension takes constant time, as does accessing the list of successors of a vertex in G. The two remaining steps of processing an extension is updating the structure (addvertex), and the update of the map data structure. Updating the tree structure entails adding a color to a colorset, storing the new score, and adding a vertex to a list. For colorsets smaller than the length of machine registers, this update can be considered constant time.

This is not a real limitation as the processing of 2^k trees becomes impractical for k significantly smaller than modern machine register lengths. There are two subtle points associated with the implementation of the tree structure. It would be easy for `addvertex` to return a complete copy of the original tree together with updates making up the extension. This would make the running time of `addvertex` a function of the input. This can be avoided by having all extensions share the common part of the structure, for example by storing the vertices encountered in reverse order in a linked list. The other subtle point is keeping track of the next endpoint. If we keep track of the index of this endpoint and keep the tree structure in a list as proposed, the `endpoint` function has to traverse a list of $O(i)$ every time for origin edge number i. However, since the structure of the tree is known a priori, we can let the tree representation have a stack that can in constant time be used to compute the next endpoint. Like the tree vertex list, extensions of a tree share the common part of the stack. Consequently, the extensions of a tree can each be done in constant time with this scheme at the cost of storing an additional $O(log(m))$ size stack for each tree. If we allocate a $n2^m$ table of empty tree references up front, `new` does nothing and `update` takes constant time. Using combinatorial number systems [Knuth] and color sets encoded in register size bitvectors, we can produce all colorsets of a given size i in $\binom{m}{i}$ time. This means that `items` for the lookup-table takes at most $n\binom{m}{i}$ time (it can also disregard empty entries in the table). Since items is called once for each edge, we get that the resulting running time is

$$\sum_{i=1}^{m-1} |E(G)|\binom{m}{i} + n\binom{m}{i} \in O(|E(G)|2^m)$$

for any connected graph. This approach also has a best case running time in $\Omega(|E(G)|2^m)$, and takes $O(n2^m)$ and $\Omega(n2^m)$ space. At the cost of an additional $O(n2^m)$ in processing time, we can instead of allocating the table up front, allocate a table of size $n\binom{m}{i}$ at step i. Space consumption is then only $O(n\binom{m}{\lceil m/2 \rceil})$., but the asymptotic running times stay the same. Another option is to use a dynamic map structure that stores only what is needed. Allocating a new structure at step i with `new` then becomes constant, but `update` becomes $O(\log(n\binom{m}{i}))$ resulting in a running time

$$O(\sum_{i=1}^{m-1} |E(G)|\binom{m}{i} \log(n\binom{m}{i})) \subseteq O(m|E(G)|2^m).$$

This bound is no longer tight, and using a dynamic table can in practice be faster than the $O(|E(G)|2^m)$ approach when pruning is effective. Also the $O(n\binom{m}{\lceil m/2 \rceil})$ space bound is not tight, as only actually constructed colorful extensions for each origin edge are stored and processed.

The Overall Algorithm

The color coding algorithm *match* can be applied only in cases where G_1 is a small tree. Then, it is optimal with a specified probability as well as reasonably efficient, particularly when we can obtain a good bound for $\sigma^*_\sim$ early on.

As stated previously, our overall approach is to

1. sample subtrees of G_1,
2. compute assignments for these, and
3. assemble a global assignment from the subtree assignments.

We randomly sample an order k subtree of the general graph G_1 by randomly sampling k vertices V' from $V(G_1)$, and then computing a random spanning subtree from the subgraph of G_1 induced by V'. The spanning subtree is computed by temporarily assigning random weights and subsequently computing a minimum spanning tree. Finally, the original edge values in this tree are restored. A second subtree sampling option implemented is to sample random stars from G_1. For a given root vertex, we create a random star by randomly sampling $k-1$ neighbors and connecting these with the edges from G_1. This is done for the vertices in G_1 in turn.

Once a subtree has been sampled, a set of assignments with corresponding value σ is computed using *match*. Every time an assignment f with value σ_f is computed, σ_f is added to the weight of the initially 0 weighted edge (a,b) for all $(a,b) \in f$ in the complete bipartite graph $B = V(G_1) \times V(G_2)$. The procedure of subsampling a tree, computing f and σ_f, and updating B is repeated until each vertex a in $V(G_1)$ has been assigned to at least a predetermined number *coverage* times. Once this has been achieved, a value $strength(a)$ is computed for each $a \in V(G_1)$ by

$$1 - 2(n(n-1))^{-1} \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} s_i * s_j,$$

where $(s_0, s_1, \ldots, s_{n-1})$ are the n weights of edges incident on a in B. Let f be any function defined on a set S. Then, let $\{S_i\}_i$ be the partition induced by f where x and y are in the same equivalence class if $f(x) = f(y)$. If we let s_i be the cardinality of S_i, then

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} s_i * s_j$$

is the number of unordered pairs of elements in S that f discerns between. Hence, the value $strength(a)$ can be interpreted as the lack of variability in the computed assignments for a. Finally, all edge weights in B are negated and the Hungarian algorithm is applied to compute a minimum cost assignment from B. This assignment and the values *strength* are the output of the overall algorithm.

Classification of problem instances Borrowing the term "deceptive problem instances" from the evolutionary algorithms literature we define deceptive matching problem instances as follows. First, for a problem instance (G_1, G_2), and an assignment $f : V(G_1) \to V(G_2)$, let $\sigma_{(G_1,G_2)}(f)$ be the score of the assignment for this problem instance. Now, let $F_{(G_1,G_2)}$ be the set of optimal assignments all sharing the same optimal score $\sigma^*_{(G_1,G_2)} = \sigma(f)_{(G_1,G_2)}$ for $f \in F_{(G_1,G_2)}$. Also, let $\mathscr{S}_k(G)$ denote the set of order k subgraphs of G. For an assignment $f : V(G_1) \to V(G_2)$ we define the set of *k-deceptive* subgraphs of G_1 as the set $D^k_{(G_1,G_2)}(f)$ of k order subgraphs for which the restriction of f is not optimal. Formally,

$$D^k_{(G_1,G_2)}(f) = \{G \in \mathscr{S}_k(G_1) | \sigma^*_{(G,G_2)} < \sigma(f|_{V(G)})_{(G,G_2)}\}.$$

For the purpose of this discussion, we will call a problem instance (G_1, G_2) *k-deceptive* if there exists $f \in F_{(G_1,G_2)}$ such that $D^k_{(G_1,G_2)}(f) \neq \emptyset$. Such an f will also be called k-deceptive. We could use $D^k_{(G_1,G_2)}$ to build hierarchies of deceptiveness, levels could be defined by requiring that all $f \in F_{(G_1,G_2)}$ be k-deceptive, then by intersections of $D^k_{(G_1,G_2)}(f)$ for $f \in F_{(G_1,G_2)}$, being deceptive for $l \leq k$, and so forth. However, an investigation of such lies outside the scope of this discussion. However, we can say that the degree of k-deceptiveness of a problem instance (G_1, G_2) is related to the minimum size of $D^k_{(G_1,G_2)}(f)$ taken over all $f \in F_{(G_1,G_2)}$.

We now define *quasi-deceptiveness*. Formally, let

$$Q^k_{(G_1,G_2)}(f) = \{G \in \mathscr{S}_k(G_1) | \sigma^*_{(G,G_2)} \leq \sigma(f|_{V(G)})_{(G,G_2)}\}.$$

We now have that $D^k_{(G_1,G_2)}(f) \subseteq Q^k_{(G_1,G_2)}(f)$. Analogous to k-deceptiveness we define (G_1, G_2) *quasi k-deceptive* if there exists $f \in F_{(G_1,G_2)}$ such that $Q^k_{(G_1,G_2)}(f) \neq \emptyset$. This f will also be referred to as being quasi k-deceptive. If an instance or assignment is quasi k-deceptive but not k-deceptive we call it *strictly quasi k-deceptive*. Again, we say that the degree of (strictly) quasi k-deceptiveness of (G_1, G_2) is related to the minimum size of $(Q^k_{(G_1,G_2)}(f) - D^k_{(G_1,G_2)}(f))$ $Q^k_{(G_1,G_2)}(f)$ taken over all $f \in F_{(G_1,G_2)}$.

Our subsampling strategy rests on the assumption that we can treat deceptiveness as noise that can be addressed by sub-sampling. We also note that any non-weighted subgraph monomorphism problem is at most strictly quasi deceptive when a solution exists.

Computational Experiments

Experiments were run on a Dell Precision T5400 with a Intel(R) Xeon(R) CPU E5430 running at 2.66GHz and 8GB ram. The algorithm was implemented in Ocaml and in Python, the latter an interpreted language. No multi-processing was used, meaning that each experiment ran on one core only.

For all experiments, each individual test problem instance was instantiated as follows. A complete graph G_2 of a given order n was created. Each edge was assigned a random weight from the unit interval. Finally, each edge weight w_{old} was rounded using the following procedure:

$$w_{\text{new}} = [w_{\text{old}} * 20]/20$$

where $[x]$ denotes the rounding of x to the nearest integer value. This means that weights are uniformly chosen among a set of 21 possible values. From this graph G_2 a random size m set of vertices from G_1 and G_2 is then chosen as the subgraph induced by this vertex set. If a tree is desired, a random spanning tree of G_1 is computed. Consequently, the optimal assignment has score $m(m-1)/2 + m$ if G_1 is complete, and $m * 2 - 1$ if G_1 is a tree.

If we were not to round the weights in the random graph, the problem would with high probability be solvable in polynomial time in the number of edges as for each edge in G_1 there would be just one edge in G_2 having the same weight.

The probability of solution uniqueness when G_1 is tree can be expressed as function of m and n and bounded by considering the probability sampling two length m sequences of edges that exhibit identical corresponding sequences of weights.

Color Coding Efficiency

In order to assess the efficiency of our color coding algorithm, we performed the following experiments. Due to the size of the trees, the OCaml implementation of the algorithms was used. For each $(m,n) \in$

$\{5,8,10\} \times \{50,100\}$ we computed 10 instances (G_1, G_2) of orders m and n, respectively. Also, the G_1 graphs were all trees, while the G_2 graphs were all complete, undirected and simple. The color coding error probability was constant at $\varepsilon = 0.2$. A summary of the results from this experiment can be seen in the table below. The columns contain for each (m,n) pair the mean of the runs for the computation time in seconds for color coding `time`. Also summarized are the solution scores for each as `score`. Note that for all color coding runs, the score was optimal.

Table 2: Color coding summary

m	n	time	score
5	50	0.027	9
5	100	0.094	9
8	50	0.297	15
8	100	0.731	15
10	50	2.219	19
10	100	4.845	19
12	100	36.834	23
13	100	215	25
14	100	1028	27

In order to test the possible speedup we could achieve if we had a perfect bound, we supplied the algorithm for the $m = 14$ $n = 100$ computation with a bound of $27 - 9.9 * 10^{-11}$. The computation time was 978 seconds, yielding a gain ratio of 1.051.

To assess the efficency for non-complete graphs, we used the method of Holme et al. [Holme] as it is implemented in the Python package "networkx", to create a randomg graph. This graph had 5000 vertices and 14991 edges, and had similar vertex degree (min 3, average 5.99, and max 227) and clustering coefficient (0.055) to the yeast protein interaction network [Xenarios] (4389 vertices, 14318 edges, average degree 6.5, maximum degree 237, clustering coefficient 0.067) used by Huffner [Huffner], Dost et al. [Dost], and Scott et al. [Scott] in their experiments. A logarithmic plot of running times for $\varepsilon = 0.01$ without precomputing for tree orders 5-12 can be seen here

in Figure 1.

The constant slope in this plot indicates that our analysis of the algorithms running time complexity is correct.

A log plot of the times reported for "standard color coding" in Dost et al. minus the times plotted above for given tree orders can be seen here

in Figure 2.

The running times reported by Dost et al. grow exponentially faster than the running times in our experiments. However, note that care should be taken when drawing conclusions from this plot as the graphs were different, and the algorithm of Dost et al. also searches for inexact matches.

```
    +---------+---------------+--------------+--------------+--+
  8 +                                                       *  +
l   |                                                *         |
o   |                                                          |
g 6 +                                        *                 +
    |                                                          |
s   |                                *                         |
e 4 +                         *                                +
c   |                                                          |
o   |                 *                                        |
n 2 +                                                          +
d   |         *                                                |
s   |  *                                                       |
    +---------+---------------+--------------+--------------+--+
              6               8             10             12
                             tree order
```

Figure 1: Color coding tree order vs. running time.

```
     +--+-------+--------+--------+--------+-------+--------+--+
  10 +                                                      *  +
     |                                                         |
l  8 +                                             *           +
o    |                                     *                   |
g  6 +                                                         +
     |                                                         |
d  4 +                                                         +
i    |                            *                            |
f  2 +                   *                                     +
f    |          *                                              |
   0 +                                                         +
     |  *                                                      |
     +--+-------+--------+--------+--------+-------+--------+--+
        6       7        8        9       10      11       12
                             tree order
```

Figure 2: Log plot of time difference to Dost et al.

Overall Algorithm Performance

In order to investigate the performance of the algorithm overall, we created 30 (G_1, G_2) instances for each possible value of $(m,n) \in \{10,20\} \times \{30,40\}$ where m and n as before are the orders of G_1 and G_2, respectively.

In order to control the computational effort expended, we chose to compute 150 random order 5 stars for each vertex in G_1 to be matched in G_2 with an $\varepsilon = 0.2$. This choice was done in an ad-hoc fashion.

For each of the (m,n) tuples, the mean values of running time in seconds `time`, correct vertex correspondences in an assignment `c`, the computed score `score`, as well as `rank` and `rank/c` are presented in the table below. The value `rank` is the smallest index in a sequence of G_1 vertices sorted on decreasing correspondence strength value for which the correspondence is incorrect. This value is never larger than the number of correct correspondences. The last column `rank/c` corresponds to the means of the rank/correct ratios.

For the computation we used an OCaml implementation of the algorithm to produce the bipartite graph B from which the final assignment is computed.

Table 3: Yagma performance summary

m	**n**	**time**	**rank**	**c**	**score**	**rank/c**
10	30	12.53270	9.592593	9.925926	54.79259	0.9629630
10	40	17.70185	4.555556	8.222222	50.75926	0.5186667
20	30	25.50262	20.000000	20.000000	210.00000	1.0000000
20	40	35.68068	19.037037	19.851852	208.92037	0.9571111

In order to compare these results with another independently developed algorithm, we applied the well known SMAC graph matching algorithm developed by Cour et al. [Cour] to the same test instances with the following results.

Table 4: SMAC performance summary

m	**n**	**c**	**score**
10	30	2.7666667	46.63000
10	40	0.5666667	44.87167
20	30	18.8666667	206.80667
20	40	2.3333333	164.11833

With respect to match quality, our algorithm consistently outperformed the algorithm of Coer et al., particularly in the more difficult $n = 40$ instances.

The implementation of SMAC we used was the one posted on the web by the authors. It is a matlab wrapper of C/C++ code. Computation times for SMAC were in the tenths of seconds for all cases, and hence are one to two orders of magnitude smaller than for our algorithm. However, to produce

```
    +-+----------+----------+----------+---------+----------+--+
    |                   *************************************  |
200 +                  *                                       +
    |              *****                                       |
    |            ***                                           |
    |    *********                                             |
150 +    *                                                     +
    |                                                          |
    |   *                                                      |
    |                                                          |
    |                                                          |
100 +  *                                                       +
    |                                                          |
    |  *                                                       |
    +-+----------+----------+----------+---------+----------+--+
      0          1          2          3         4          5
```

Figure 3: Coverage vs. score.

results comparable in quality to those produced by the SMAC algorithm, we used on average 1.25 seconds in 10 $m = 20$ and $n = 40$ cases to produce an average $c = 3.1$ by only requiring $k = 4$ and a coverage of 1. In this case, SMAC used on average 0.66 seconds, which means that in this instance SMAC is less than 2 times as fast with results a third worse. While our algorithm is slower, it allows a rudimentary balancing of match performance and computational effort. The progression of solution score as function of *coverage* in a $m = 20$, $n = 40$ test instance using random stars of order 5 can be seen here

in Figure 3.

For this particular run, using each vertex in G_1 as the root of a random star twice would have been sufficient.

Ranking performance The overall average rank/correct ratio is 0.85. This means that if we sort the individual correspondences decreasingly according to their strength, a prefix of length $0.85 * c$ consists of correct correspondences. As such, this ratio serves as an indicator of the usefulness of the strength measure computed for each individual correspondence in an assignment. Note that the normalization achieved by dividing by c makes this independent from the quality of the assignment.

Discussion and Conclusion

Summary of results We presented a graph matching algorithm in terms of a fold over trees that can be directly translated into efficient code, reflecting a view of pattern matching as a catamorphism. Our overall approach of composing sub-matchings obtained by matching randomly sampled trees worked well in our test cases in which it significantly outperformed a well known graph matching algorithm in terms of matching quality.

Limiting scope of experiments Our experiments were restricted to limited size, strictly quasi-deceptive subgraph-isomorphism problem instances. The reasons for this restriction is that this allows the determination of the optimal match score, and hence the ability to determine the optimality of computed solutions. It is worth noting that all non-weighted subgraph monomorphism problems are in this class. Within this restriction, we chose a problem class consisting of instances composed of complete graphs with rounded edge weights randomly sampled from the unit interval. The only structural information is contained in the edge weights as the graphs were undirected and complete and had no vertex annotations. Furthermore, we rounded the edges to be of maximally 21 different possible values, ensuring a high degree of ambiguity in sub-matchings. In summary, while the experimental problem instances were of a limited class, we deliberately chose hard instances within this class.

Simplicity of the color coding algorithm implementation As noted previously, our definition of the color coding algorithm is essentially expressed in terms of a fold over the pre-order traversal list of tree vertices. Recursive patterns such as folds are routinely provided by standard libraries of languages that support functional programming idioms. An eloquent case for functional programming is given in the 1989 seminal paper "Why functional programming matters" by John Hughes [Hughes]. For example, once we have a list fold, the sum of a list of integers can be presented as a fold of addition over the list. The properties of sum can then be inferred from the properties of the fold and the properties of the binary addtion operation. Recursive patterns such as fold are not only useful for programmers, but have been studied from a theoretical perspective and the results provide a framework for reasoning about and proving properties of programs and algorithms expressed in terms of these patterns. Meijer [Meijer] discusses catamorphisms (fold), anamorphisms (unfold), hylomorphisms (fold-unfold), and paramorphisms over general recursive data structures from a theoretical standpoint. One of the results of this work relevant for us is the extension of fold properties to general (finite and infinite) recursive data structures. As we have seen the particularities of color coding are essentially contained in a binary operator g. Whether we are matching a path or a tree is now abstracted away in the recursive data structure the fold is over, essentially hiding the extension of path matching to tree-matching in the fold recursive pattern. As presented above, the parts of the code containing the essentials of the algorithm is only a few lines long, and follow the textual descriptive explanation closely. As an example, an efficient Python implementation of the algorithm for networkx module graph representations, including sub-tree pruning, is about 80 lines long. Our OCaml implementation of the algorithm is competitive compared to reported running times in the literature.

A measure of solution quality Considering the approximation hardness results presented by Mielikainen et al. [Mielikainen], offering guaranteed good overall solutions might be impossible. However, our experiments suggest that the measure *strength* computed from the bipartite graph constructed from the collection of sub-tree assignments can be used in a heuristic manner to assess the quality of individual vertex correspondences. This can be applied to identify origin sub-graphs for which it is easier to compute good solutions. How the identified concept of deceptive sub-graphs affects this and applicability in general is a topic for future research.

Inexact matching by local exact matching While the matching of sub-trees is performed in an (syntactically but not semantically) exact manner, the overall composition is inexact. Dost et al. [Dost] presented a syntactically inexact version of color coding where vertices were allowed to be missing in both origin and target (the former phrased as inserts in the target). How such extensions can be incorporated into the catamorphic presentation of color coding, is another topic for future research.

Acknowledgments

We thank Vineet Bafna for pointing out the color coding approach, and Jihoon Kim for proposing the normalization of the ranks. This work was funded in part by NIH grants 5 R01 LM007273-07 and U54 HL108460.

References

References

[Alon] Alon N., Yuster R., Zwick, U. Color-coding. Journal of the ACM (JACM), ACM, 1995, 42, 844-856

[Boros] Boros E., Hammer P. Pseudo-boolean optimization. Discrete Applied Mathematics, Elsevier, 2002, 123, 155-225

[Conte] D. Conte et al. Thirty Years of Graph Matching in Pattern Recognition. International Journal of Pattern Recognition and Artificial Intelligence Vol. 18, No. 3 (2004) 265--298

[Cordella] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence 26(10):1367-1372, 2004

[Cour] Cour, T., Srinivasan P., Shi J. Balanced graph matching. Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference, 2007, 19

[Dost] Dost B., Shlomi T., Gupta N., Ruppin E., Bafna V., Sharan R. QNet: A tool for querying protein interaction networks. Research in Computational Molecular Biology, 2007, 1-15

[Downey] Downey R.,Fellows M. Fixed-parameter intractability. Structure in Complexity Theory Conference, 1992., Proceedings of the Seventh Annual, 1991, 36-49

[Holme] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering, Phys. Rev. E, 65, 026107, 2002.

[Huffner] Huffner F., Wernicke S., Zichner T. Algorithm engineering for color-coding with applications to signaling pathway detection. Algorithmica, Springer, 2008, 52, 114-132

[Hughes] Hughes J. Why Functional Programming Matters. Computer Journal, 1989, 32, 98-107

[Hutton] Hutton G. A tutorial on the universality and expressiveness of fold. Journal of Functional Programming 9 (4): 355-372, July 1999.

[Knuth] Knuth DE, "Generating All Combinations and Partitions", The Art of Computer Programming Addison-Wesley, 2005. ISBN 0-201-85394-9.

[Meijer] Meijer E, Fokkinga M, Paterson R. Functional programming with bananas, lenses, envelopes and barbed wire. Proceedings of the 5th ACM conference on Functional Programming Languages and Computer Architecture. Cambridge, MA, USA, August 26-30, 1991.

[Mielikainen] Mielikainen T., Ravantti J., Ukkonen E. The Computational Complexity of Orientation Search Problems in Cryo-Electron Microscopy. CoRR, 2004, cs.DS/0406043

[Mu] Mu S, Bird R. Rebuilding a tree from its traversals: a case study of program inversion. Programming languages and systems: first Asian Symposium, APLAS 2003. Beijing, China, November 27-29, 2003: proceedings, 2003.

[Scott] Scott J, Ideker T, Karp RM, Sharan, R. Efficient algorithms for detecting signaling pathways in protein interaction networks. J Comput Biol, Computer Science Division, University of California, Berkeley, 94720, USA., 2006, 13, 133-144

[Torresani] Torresani L., Kolmogorov V., Rother C. Feature correspondence via graph matching: Models and global optimization. Computer Vision--ECCV 2008, Springer, 2008, 596-609

[Xenarios] Xenarios I, Salwinski L, Duan XJ, Higney P, Kim SM, Eisenberg D. DIP, the Database of Interacting Proteins: a research tool for studying cellular networks of protein interactions. Nucleic Acids Res, 2002, 30, 303-305.