An Algorithm Portfolio for the Sub-Graph Isomorphism Problem

Roberto Battiti and Franco Mascia

Università degli Studi di Trento, Italy
{battiti,mascia}@dit.unitn.it

Abstract. This work presents an algorithm for the sub-graph isomorphism problem based on a new pruning technique for directed graphs. During the tree search, the method checks if a new association between two vertices is compatible by considering the structure of their local neighborhoods, represented as the number of limited-length paths of different type originating from each vertex. In addition, randomized versions of the algorithms are studied experimentally by deriving their run-time distributions. Finally, algorithm portfolios consisting of multiple instances of the same randomized algorithm are proposed and analyzed. The experimental results on benchmark graphs demonstrate that the new pruning method is competitive w.r.t. recently proposed techniques. Significantly better results are obtained on sparse graphs. Furthermore, even better results are obtained by the portfolios, when both the average and standard deviation of solution times are considered.

1 Introduction

The sub-graph isomorphism problem, a.k.a. graph pattern matching, consists of determining if an isomorphic image of a graph is present in a second graph. The problem, or relaxed versions thereof, appears in significant applications, ranging from computer vision, structural pattern recognition, chemical documentation, computer-aided design, and visual languages, see for example [1–4] for references.

Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be two graphs, V and E being their vertices and edges, respectively. A sub-graph isomorphism is bijective function $M : V_1 \rightarrow V_2' \subseteq V_2$ having the following property: $(u, v) \in E_1 \Leftrightarrow (M(u), M(v)) \in E_2' \subseteq E_2$, where E_2' contains the edges induced by the vertices in V_2'. Let's note that another definition has been used in some papers, for example [3], where existence of an arc in G_1 implies existence of an arc $(M(u), M(v))$ in G_2, but not *vice versa*, there can be arcs in E_2' which do not correspond to arcs in G_1.

The original motivation for this work is double. First, we investigate whether the adoption of a portfolio approach produces better results by considering more instances of the same algorithm running in a time-sharing fashion. Second, we experiment with novel pruning techniques based on the local structure around the next node to be associated.

In the following sections, the existing state-of-the-art approaches are briefly reviewed in Section 2, then our new pruning technique based on paths compatibility is explained in Section 3. The computational experiments to assess the

efficacy and efficiency of the new pruning proposal are presented in Section 4 for the deterministic algorithms, and in Section 5 for the randomized versions. The motivation for using portfolios and the proposal is explained in Section 6, and the corresponding computational results are presented in Section 7.

2 Existing Approaches

The sub-graph isomorphism problem is NP-hard [5], and previous approaches for its solution include [1–4]. A recent algorithm appropriate for matching large graphs encountered in relevant applications is proposed in [4]. The proposed method VF2 is an exact algorithm for the sub-graph isomorphism problem, which explores the search graph by means of a depth-first-search and which uses new pruning techniques to reduce the size of the generated solution tree. The effectiveness of VF2 is assessed in the cited paper, which contains also experimental comparisons with Ullmann [1] and Nauty [6] algorithms.

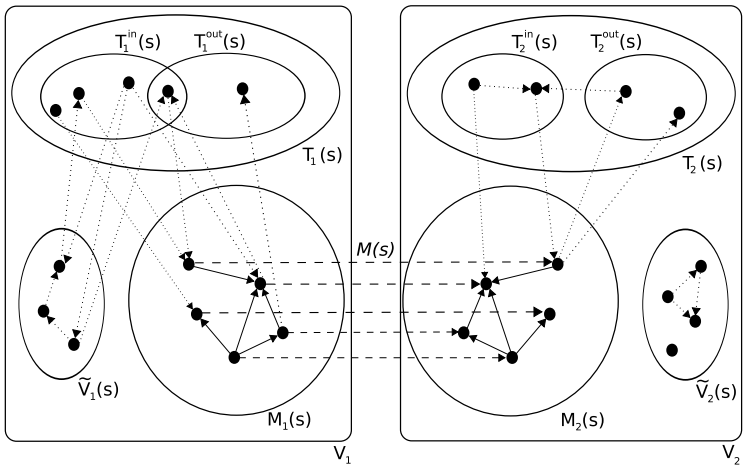

Fig. 1. Partial mapping M_s and sets in VF2.

Let's introduce the notation used to explain VF2 and our novel proposal. Let $M \subset V_1 \times V_2$ be the isomorphism, and M_s the mapping at state s in the state space representation. A mapping is developed by adding a new pair of nodes (v_1, v_2) at each step, and the state s is given by the current set of associations between nodes of G_1 and nodes of G_2. $M_1(s)$ and $M_2(s)$ are the set of vertices $v_1 \in V_1, v_2 \in V_2$ such that $(v_1, v_2) \in M_s$ and $G_1(s)$ and $G_2(s)$ the sub-graphs induced by these sets. Let $T_1^{in}(s)$ and $T_1^{out}(s)$ be the set of vertices adjacent from and to the vertices in $M_1(s)$, but not yet in the partial mapping $M_1(s)$, and $T_1(s) = T_1^{in}(s) \cup T_1^{out}(s)$. The set of the vertices $\tilde{V_1} = V_1 - M_1(s) - T_1(s)$ is the set of vertices $u \in V_1$ not connected to vertices belonging to the mapping.

Fig. 1 shows the sets described above, highlighting the connections among the induced sub-graphs $G_1(s)$ and $G_2(s)$ with solid arcs, the partial mapping with dashed arcs, and connections with the terminal sets with dotted arcs.

```
1.  MATCH (G1, G2, s)
2.  if Ms covers all the vertices of G1 then
3.      return Ms
4.  else
5.      foreach (v1, v2) ∈ P(s) do
6.          if COMPATIBLE(v1, v2) then
7.              s' ← s ∪ (v1, v2)
8.              MATCH (G1, G2, s')
9.      done
10. return no match found
```

Fig. 2. A generic back-tracking scheme for the sub-graph isomorphism problem. The function COMPATIBLE determines the pruning and it depends on the specific algorithm.

Fig. 2 shows the back-tracking algorithm which implements the depth-first-search. If the partial mapping M_s covers all the vertices of G_1, the goal is reached, otherwise the depth-first-search goes deeper in the search tree and tries to add a new pair to the current state s.

To reduce as much as possible the CPU time, by an appropriate ordering the algorithm never visits the same state twice. The search space reduction w.r.t. the complete search tree is determined by the pruning technique. Pruning acts by controlling that a candidate pair $p = (v_1, v_2)$ selected from the set of candidate pairs $P(s)$ survives the test executed by the COMPATIBLE routine. If COMPATIBLE returns false, the addition of the new pair is doomed to failure and the sub-tree is pruned.

In detail, the COMPATIBLE routine for VF2 works as follows. The candidate pairs $(v_1, v_2) \in P(s)$ are selected with priority to the nodes adjacent *from* the vertices already in the mapping, i.e., $v_1 \in T_1^{out}(s)$ and $v_2 \in T_2^{out}(s)$. If there are no such vertices, the pairs of the ones adjacent *to* the vertices already in the mapping are selected. If a graph has more than one connected component such couples could not exists, and in this case the "less constrained" vertices belonging to $\tilde{V}_1(s)$ and $\tilde{V}_2(s)$ are considered.

Let us now introduce the sets of predecessors and successors of the current node: $\mathsf{Pred}(G, v) = \{u \in V | (u, v) \in E\}$ and $\mathsf{Succ}(G, v) = \{u \in V | (v, u) \in E\}$. The COMPATIBLE routine performs each of the following tests in order, stopping early if at least one test fails. The first test checks if the partial mapping extended with the additional association $(M_s \cup (v_1, v_2),$ where $(v_1, v_2) \in P(s))$ is still a valid isomorphism: for all nodes already in the partial mapping, edges to (from) the last nodes considered for addition must be preserved by the extended mapping: if an edge is present in the graph induced by $M_1(s) \cup v_1$ the

corresponding edge must be present in the graph induced by $M_2(s) \cup v_2$, and *vice versa*.

$$\forall v_1' \in M_1(s) \quad (v_1', v_1) \in E_1 \Rightarrow (M_s(v_1'), v_2) \in E_2 \tag{1}$$

$$\forall v_1' \in M_1(s) \quad (v_1, v_1') \in E_1 \Rightarrow (v_2, M_s(v_1')) \in E_2 \tag{2}$$

$$\forall v_2' \in M_2(s) \quad (v_2', v_2) \in E_2 \Rightarrow ({M_s}^{-1}(v_2'), v_1) \in E_1 \tag{3}$$

$$\forall v_2' \in M_2(s) \quad (v_2, v_2') \in E_2 \Rightarrow (v_1, {M_s}^{-1}(v_2')) \in E_1 \tag{4}$$

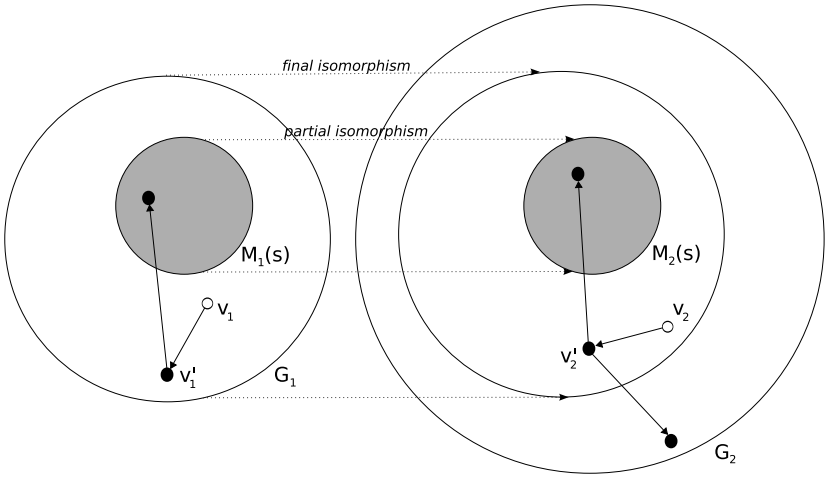

Fig. 3. Example of the additional checks executed by VF2. A node in G_1 will have to be mapped to a compatible node in G_2 in the future steps. If no compatible node in G_2 is available the partial mapping is doomed.

If the previous checks give the green light to extend the mapping, the following additional checks are performed to prune the search tree, trying to find incompatibilities between branches of the two graphs that could arise in the future steps. The tests count number of nodes with different connectivity structure w.r.t. M_1 in G_1 and make sure that *at least* the same number of nodes with compatible connectivity structure is available in G_2. Otherwise, for sure the mapping cannot be completed in the future steps. To follow the different cases it may be useful to consider the example in Fig. 3, related to the check in eqn. 5. The node external to M_1 is a successor of v_1 and has at least one incoming arc to M_1. If the mapping is to be completed, at least one node in G_2 external to M_2 with compatible edges has to be present. Again, given a number of nodes with a certain connectivity in $G_1 \setminus M_1$, at least the same number of

nodes with compatible connectivity has to be present in $G_2 \setminus M_2$. Let's note that, in addition to the required edges, some additional edges may be present in G_2 because only a subset of its nodes will be covered by the final mapping. The different cases consider all possible directions for the edges (in-in, in-out, out-out, out-in) and finally the case of successors and predecessors without edges to or from the current M_1.

$$|T_1^{in}(s) \cap \mathsf{Succ}(G_1, v_1)| \leq |T_2^{in}(s) \cap \mathsf{Succ}(G_2, v_2)| \tag{5}$$

$$|T_1^{in}(s) \cap \mathsf{Pred}(G_1, v_1)| \leq |T_2^{in}(s) \cap \mathsf{Pred}(G_2, v_2)| \tag{6}$$

$$|T_1^{out}(s) \cap \mathsf{Succ}(G_1, v_1)| \leq |T_2^{out}(s) \cap \mathsf{Succ}(G_2, v_2)| \tag{7}$$

$$|T_1^{out}(s) \cap \mathsf{Pred}(G_1, v_1)| \leq |T_2^{out}(s) \cap \mathsf{Pred}(G_2, v_2)| \tag{8}$$

$$|\tilde{V}_1(s) \cap \mathsf{Succ}(G_1, v_1)| \leq |\tilde{V}_2(s) \cap \mathsf{Succ}(G_2, v_2)| \tag{9}$$

$$|\tilde{V}_1(s) \cap \mathsf{Pred}(G_1, v_1)| \leq |\tilde{V}_2(s) \cap \mathsf{Pred}(G_2, v_2)| \tag{10}$$

3 Pruning by Considering Paths Compatibility

The motivation for the cited pruning technique and for the new one is as follows. Let's assume that we are checking for an addition of the pair (v_1, v_2) to the current mapping. Now, if the mapping is going to be completed, the local structure of connections around $v_1 \in G_1$ will have to be mapped to a similar local structure around $v_2 \in G_2$. The tests in VF2 considered counts of successor or predecessor *nodes* with different connectivity, we decided to explore checks dedicated to counting *paths* of different kinds. In particular, if there is a path in G_1 of length d starting from vertex v_1, the same path has to be found in G_2 starting from vertex v_2. If such path in G_2 does not exist we can safely omit considering (v_1, v_2) and therefore we can prune the part of the search tree arising from this novel association.

Let us call this general principle "local-paths-based pruning". The realization considered in the present work is based on counting paths in the *underlying* (*undirected*) graph UG corresponding to the original graph. Edge (u, v) is present in the undirected graph if and only if arc (u, v), arc (v, u) or both are present in the original graph. Given a path in UG, it is labeled according to the direction of the arcs in the original graph G. For example, see Fig. 4 for the illustration of a path of kind "out-in-out" arising from v_1. Let us note that we consider *all* paths, including also non-simple ones, with cycles and repeated vertices.

Before starting the algorithm, a pre-processing phase counts the number of paths of length up to d of the different kinds explained above originating at the different vertices of the two graphs G_1 and G_2. When the algorithm encounters a pair of vertices (v_1, v_2) to be tested for possible inclusion in the mapping, one tests whether the number of paths originating at v_1 and v_2 are compatible. In detail, for each length from 1 to d, if the number of paths of at least one kind originating from v_1 is bigger that the number of paths of the same kind

originating from v_2, the test is immediately terminated in a negative way. No possible isomorphism can be found by adding (v_1, v_2) to the current mapping.

The new pruning technique presented in this work, hereinafter referred as *BM1* (BattitiMascia-1), is a compatibility check applied before the VF2 check with the aim of further reducing the size of the search tree. Of course the reduction in the number of states visited comes at the cost of an increased complexity of the extra check, and the length d of the paths impacts the precision as well as the cost of the check.

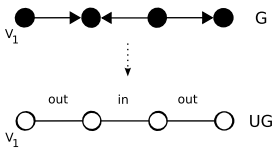

Fig. 4. Path originating from vertex v_1 as considered in BM1.

```
1. CompatiblePaths (v1, v2, d)
2.   foreach r in 1,...,d do
3.     foreach i in 1,...,2^d do
4.       if (PathsDS[v1][r][i] > PathsDS[v2][r][i]) then
5.         return false;
6.     done
7.   done
8.   return true;
```

Fig. 5. Pseudo-code for path-counting pruning used in the BM1.

3.1 Data Structures and Computational Complexity

The BM1 algorithm requires an appropriate value of the d parameter. Larger values of d will prune more but at the cost of an increasing computation and memory requirement. It is therefore of interest to evaluate the effectiveness in the reduction of tree size, and the space and time costs as a function of the parameter d. Fig. 5 shows the CompatiblePaths pseudo-code. The numbers of paths of different length and different kinds are compared and the test returns immediately as soon as, for a specific length and kind, the number of paths in G_2 is less than the number of equivalent paths in G_1.

Fig. 6 shows how the neighborhood information is stored in an ad-hoc data-structure which is computed statically before the actual search takes place. The

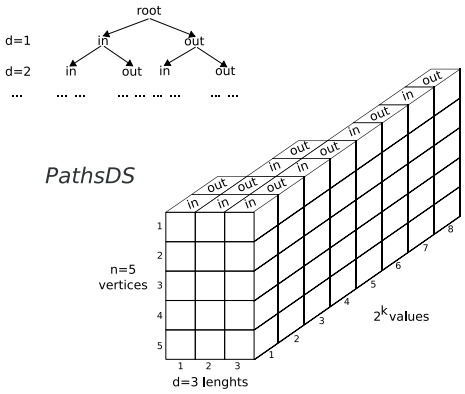

Fig. 6. PathsDS: data structure storing the number of paths of different kinds originating from a node.

tree rooted at the vertex whose neighborhood has to be checked shows the information stored in the data structure, i.e., the number of paths of length d labeled with the corresponding "in", "out" arc labels on the edges of the graph.

During the run, each of the d-length paths checks are performed by making a number of comparisons equal to the leaves in the binary tree representing the neighborhood at the given length. The time complexity of a check for a single pair of vertices is in the worst case equal to:

$$\sum_{i=1}^{d} 2^i = 2^{d+1} - 2$$

The data structure is constructed in a recursive way: the nodes and all their neighbors are visited until all paths of depth d are reached, and during the tail of the recursion all degrees are summed up to fill in the elements.

The time complexity for building the data structure is bounded by $O(n^d)$ and the space occupied by the table is $n * (2^{d+1} - 2)$, see Fig. 6.

4 Computational Experiments for VF2 and BM1

The technique has been tested against chosen instances of the AMALFI Graph DataBase[7]. In order to study the effectiveness of BM1, ten random graphs classes have been selected, having different number of nodes, density, and sub-graph sizes.

Each class, which contains 100 instances of the problem, is identified by the size of the sub-graph (`si2` means that the number of vertices of the sub-graph is 20% of the graph), the number of vertices of the graph, and the probability η of connection between the vertices. More in detail, the graph is constructed by

connecting the vertices with a number of arcs equal to $\eta \cdot |V| \cdot (|V| - 1)$ and by successively adding arcs until the graph is connected [7].

The sub-graph isomorphism between each pair has been tested by means of the original VF2 method, and of the BM1 proposal with values of the parameter d ranging from 1 to 3. For these values of d the initialization time to build the data structure used by the CompatiblePaths routine is hardly measurable and not significant w.r.t. the CPU time spent during the tree search. In any case, the total CPU time including initialization is measured in the experiments.

The CPU time spent by the algorithms is measured on our reference machine, having one Xeon processor at 3.4 GHz and 6 GB RAM. The operating system is a Debian GNU/Linux 3.0 with kernel 2.6.15-26-686-smp. All the algorithms are compiled with the g++ compiler with "-O3 -mcpu=pentium4".

To compare the performance we consider both the number of visited states (number of tree nodes) and the CPU time of the different alternatives. For convenience we also report ratios of the above values.

Table 1 summarizes the average number of visited states of the three different BM1(d) compared with VF2. It can be observed that the new pruning technique delivers comparable results on the denser random graphs with $\eta = 0.01$, while it delivers a significantly smaller number of visited states for the sparse graphs with $\eta = 0.001$. For example, for the less dense classes (e.g. `si6_200_001`) the reduction of visited states reaches 92%. While the additional cut in visited states is negligible for the denser graphs, the cut tends to increase as a function of the d parameter for the sparser graphs. Nonetheless, the speed of reduction in the number of states decreases rapidly as soon as d reaches large values. This can be observed from the Table by considering the reduction when passing from $d = 1$ to $d = 2$ and the much smaller reduction when passing from $d = 2$ to $d = 3$.

Both results, larger effectiveness for sparser graphs and diminishing additional cuts for large d values, are not unexpected. One has to consider that the number of possible paths increases very rapidly as a function of d, in particular if the graph is dense. Because more nodes and edges are available in G_2 to build possible paths one has to expect that the number of paths in G_2 when d increases will become so large that the inequalities in the tests in the CompatiblePaths routine will be easily satisfied. In practice this means that the bigger cuts are for very small values of d, a positive note when one consider the CPU time spent during the checks.

Table 2 compares the average time spent by the algorithms for finding the mapping between the instances of the different graph classes. For denser graphs ($\eta = 0.01$) the reduction in the number of visited states is too small to see a reduction in the CPU time. For example, the time needed for solving `si2_200_001` instances increases with the parameter d because the additional cost of the path compatibility check is not balanced by the reduction in the number of visited states. In the case of sparser graphs ($\eta = 0.001$) BM1 is able to prune the search space more effectively, and the average time for solving the instances decreases with the length of the paths checked, growing again when the increased length does not result in further pruning.

Instances	VF2	BM1 (d=1)	BM1 (d=2)	BM1 (d=3)
si2_200_01	44 822.60	44 814.14	44 814.14	44 814.14
si2_400_01	505 473.15	505 473.15	505 473.15	505 473.15
si6_200_01	2 524.05	2 100.32	2 100.32	2 100.32
si6_400_01	52 714.16	48 045.39	48 045.39	48 045.39
si6_800_01	5 012 505.53	4 931 432.31	4 931 432.31	4 931 432.31
si2_200_001	428 800.24	155 074.65	118 939.67	115 336.62
si2_400_001	2 315 104.50	838 758.49	818 768.34	818 290.73
si6_200_001	8 756.76	1 292.44	813.93	741.97
si6_400_001	1 303 904.85	82 158.56	47 317.74	46 621.02
si6_800_001	6 006 050.24	1 088 435.49	1 066 752.11	1 066 752.11

Table 1. Average states visited by each algorithm for selected graph classes.

Instances	VF2	BM1 (d=1)	BM1 (d=2)	BM1 (d=3)
si2_200_01	447 500.00	474 900.00	515 200.00	559 600.00
si2_400_01	11 838 800.00	12 612 900.00	13 855 200.00	14 822 800.00
si6_200_01	27 400.00	22 800.00	25 100.00	24 800.00
si6_400_01	1 709 500.00	1 613 200.00	1 733 300.00	1 847 100.00
si6_800_01	375 137 400.00	410 477 100.00	386 953 600.00	461 466 400.00
si2_200_001	2 403 500.00	983 000.00	836 500.00	903 800.00
si2_400_001	18 645 500.00	7 783 200.00	8 501 600.00	9 212 800.00
si6_200_001	45 800.00	5 700.00	3 400.00	4 100.00
si6_400_001	11 379 700.00	660 200.00	401 600.00	411 200.00
si6_800_001	79 632 000.00	13 441 000.00	14 065 400.00	13 944 800.00

Table 2. Average time in μ-seconds spent by each algorithm for selected graph classes.

Finally Table 3 summarizes the ratios between the average number of steps and times spent by the algorithms for solving the problem instances. The ratio is between BM1 and VF2, therefore values smaller than 1 implies that BM1 is the winning algorithm.

5 Cumulative Distribution Functions of Randomized Versions

The time spent by the exact algorithm depends on the particular instance of the class of random graphs, but, for each single instance, also on the order in which the vertices are visited. In the original algorithm [4] the choice of the candidate vertices is deterministic. All considered algorithms have been randomized by randomly permuting the vertices in the input graph G_1 before starting. Therefore, in case of ties when considering the next nodes to be mapped, different nodes will be selected in different runs, leading to different results.

After randomization, the information of interest about the performance is summarized in the empirical cumulative distribution functions (CDF for short).

Instances	BM1 (d=1)		BM1 (d=2)		BM1 (d=3)	
	steps	μ-sec	steps	μ-sec	steps	μ-sec
si2_200_01	1.00	1.06	1.00	1.15	1.00	1.25
si2_400_01	1.00	1.07	1.00	1.17	1.00	1.25
si6_200_01	**0.83**	**0.83**	0.83	0.92	0.83	0.91
si6_400_01	**0.91**	**0.94**	0.91	1.01	0.91	1.08
si6_800_01	0.98	1.09	0.98	1.03	0.98	1.23
si2_200_001	0.36	0.41	**0.28**	**0.35**	0.27	0.38
si2_400_001	**0.36**	**0.42**	0.35	0.46	0.35	0.49
si6_200_001	0.15	0.12	**0.09**	**0.07**	0.08	0.09
si6_400_001	0.06	0.06	**0.04**	**0.04**	**0.04**	**0.04**
si6_800_001	**0.18**	**0.17**	0.18	0.18	0.18	0.18

Table 3. Steps and time *ratio* between the BM1 algorithm with three different path lengths and VF2. The best length d of the check for the given instance is highlighted. If there is no such value in the row, then VF2 is a better choice.

Fig. 7 and 8 show the probability of terminating within a given amount of microseconds for the VF2 and BM1 algorithms on two selected instances from the sparse graphs. Both algorithms were tested 1000 times with different random seeds on a single representative instance of the si6_r001_m200 and si6_r001_m400 random classes.

6 Algorithm Portfolios

The algorithm portfolios method, first proposed in [8], follows the standard practice in economics to obtain different return-risk profiles in the stock market by combining stocks characterized by individual return-risk values. Risk is related to the standard deviation of return. An evaluation of the portfolio approach on distributions of hard combinatorial search problems is considered for example in [9].

The basic algorithm portfolio method consists of running more algorithms concurrently on a sequential computer, in a time-sharing manner, by allocating a fraction of the total CPU cycles to each of them. The first algorithm to finish determines the termination time of the portfolio, the other algorithms are stopped immediately after one reports the solution.

It is intuitive that the CPU time can be radically reduced in this manner for some statistical distributions of run-times. To clarify ideas, let us consider an extreme example where, depending on the initial random seed, the termination time can be of 1 second or of 1000 seconds, with the same probability. If we run a single process, the expected termination time is approximately of 500 seconds. If we run more copies, the probability that at least one of them is lucky (i.e., that it terminates in 1 second) increases very rapidly towards one. Even if termination is now longer than 1 second because more copies share the same CPU, it is intuitive that the expected time will be much shorter than 500.

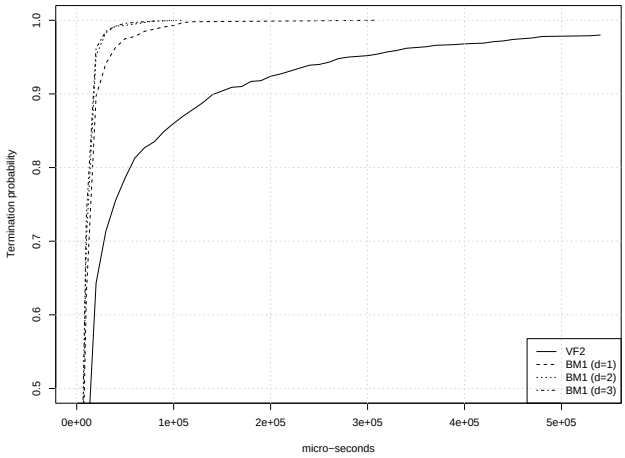

Fig. 7. Probability for randomized version of the algorithm to solve a `si6_r001_m200` instance within a fixed time.

The solution time of the portfolio t is related to the one of the individual instances of the algorithm. For a two instance-portfolio it corresponds to:

$$t = \min\{t_1 * 2, t_2 * 2\} \tag{11}$$

where t_1 and t_2 are the time spent by the to running instances to find the solution.

For a portfolio of N component instances, the probability that all instances terminate after t, because of the independence assumption and the slow-down effect, is equal to:

$$(1 - CDF(t/N))^N$$

The probability of the complementary event that at least one terminates before t, and therefore that the portfolio converges before t, is therefore:

$$CDF_{portfolio}(t) = 1 - (1 - CDF(t/N))^N$$

After taking differences one derives the distribution $p(t)$ of the portfolio finishing at time t, from which the expected value $E(t)$ and standard deviation $\sigma = \sqrt{\mathrm{Var}(t)}$ can estimated.

7 Computational Experiments for Portfolios

Considering the `si6_r001_m400` instance, the average times for BM1 and VF2 are 6 and 83 seconds, respectively, and both cumulative time distribution functions

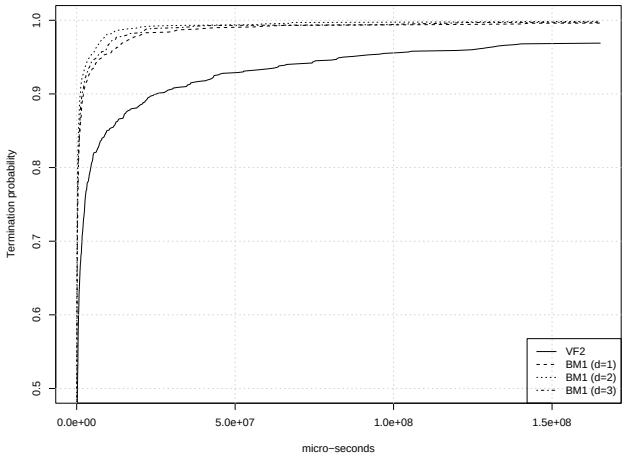

Fig. 8. Probability for randomized version of the algorithm to solve a `si6_r001_m400` instance within a fixed time.

are heavy tailed having a standard deviations of 89 seconds and 20 minutes respectively. After looking at the probability distributions, both algorithms are good candidates to be blended in a portfolio.

By combining several instances of BM1 in time-sharing, the probability to spend more than 20 seconds for finding a solution decreases from 0.12 to 0.02 with 2 instances and to 0.002 with only 4 instances. Fig. 9 and 10 show the new CDFs of the two algorithms.

The portfolio can be implemented by running one incremental step of each instance of the algorithm at a time, sharing the same process space as well as the path data structure PathsDS. In this way, the performance degradation is less for the lack of a "real" context switch, and the space as well as the cost of building the shared data structure is shared over the different instances. Fig. 11 and Fig. 12 show the mean solution time versus standard deviation (measured in micro-seconds) for two portfolios algorithms, using VF2 or BM1(1), on a single `si6_r001_m200` and `si6_r001_m400` instance, respectively. It can be noted how a portfolio consisting of a few copies of the same algorithm rapidly reduced the standard deviation of convergence times. As in the standard portfolio usage, the final choice among Pareto-optimal configurations is then up to the final user, depending on his level of risk-aversion.

8 Conclusions

The novel proposal of this paper consists of the definition of a new parametric pruning technique for the sub-graph isomorphism problem, the analysis of a ran-

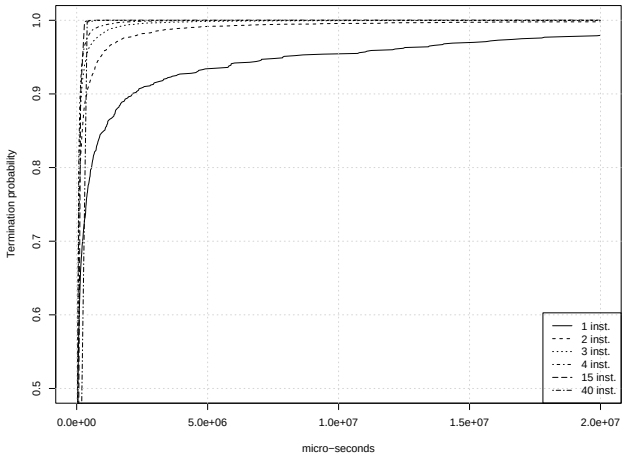

Fig. 9. Probability for a portfolio of several instances of the randomized version of the algorithm to solve a `si6_r001_m400` instance within a fixed time.

domized version of the BM1 and VF2 algorithms, and the study of an algorithm portfolio approach for the problem.

The experimental results on the considered benchmark graphs demonstrate that the proposed pruning technique is effective in reducing the average number of states visited by the BM1 algorithm for sparse random graphs. The reduction in the number of steps and also in the average time spent by the algorithms reaches 92% for some instances. On denser graph classes the reduction in the visited states is not sufficient in order to achieve also a reduction in the average CPU time.

When portfolios are considered, the heavy tails of the empirical run-time distributions of the algorithms can easily be cured by running more randomized instances concurrently on the same machine. Portfolios of algorithms using the proposed pruning technique dominate VF2 on sparse random instances.

Acknowledgement

We acknowledge useful discussions with Krishnam Raju during his internship at Trento about various graph isomorphism heuristics.

References

1. Ullmann, J.: An Algorithm for Subgraph Isomorphism. Journal of the ACM (JACM) **23**(1) (1976) 31–42

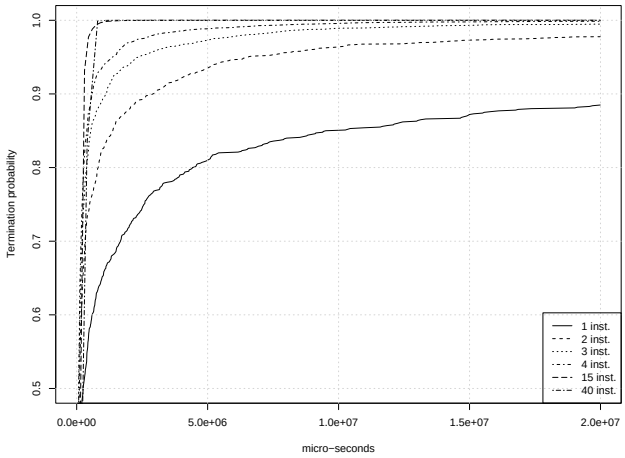

Fig. 10. Probability for a portfolio of several instances of the randomized version of the algorithm to solve a `si6_r001_m400` instance within a fixed time.

2. Bunke, H., Messmer, B.T.: Recent advances in graph matching. IJPRAI **11**(1) (1997) 169–203
3. Larrosa, J., Valiente, G.: Constraint satisfaction algorithms for graph pattern matching. Mathematical Structures in Computer Science **12**(4) (2002) 403–422
4. Luigi P. Cordella, Pasquale Foggia, C.S., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence **16**(10) (October 2004) 1367–1372
5. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
6. McKay, B.: Practical graph isomorphism. In: Numerical mathematics and computing, Proc. 10th Manitoba Conf., Winnipeg/Manitoba. (1980) 45–87
7. http://amalfi.dis.unina.it/
8. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275** (January 3 1997) 51–54
9. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1-2) (2001) 43–62

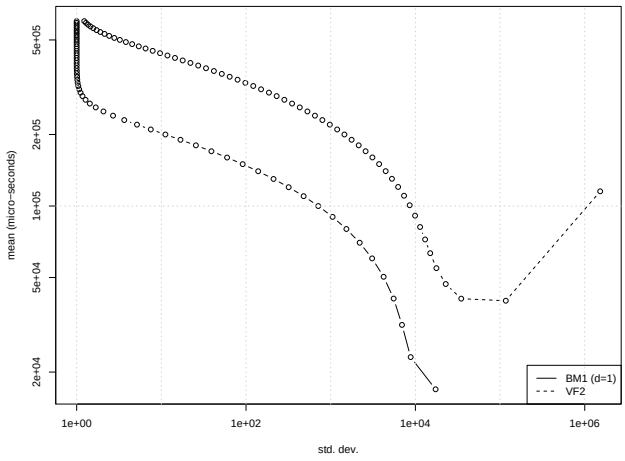

Fig. 11. Portfolios of several instances of the randomized VF2 and BM1 algorithms. The rightmost point in both curves corresponds to a single instance mean and standard deviation, the 2nd point to two instances, the 3rd three and so on. Each point is computed on 1000 runs on `si6_r001_m200`. Plot is in log-log scale.

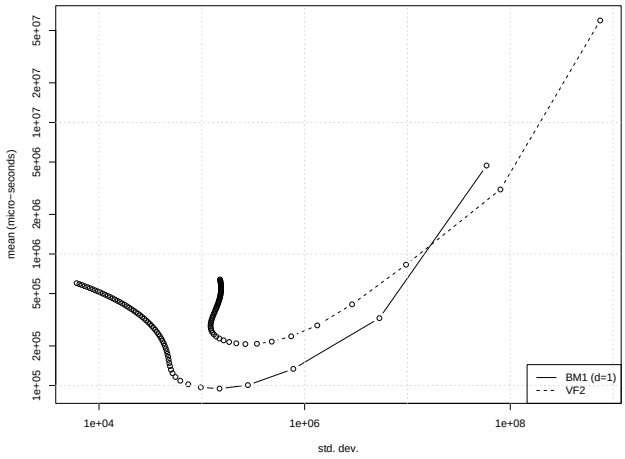

Fig. 12. Portfolios of several instances of the randomized VF2 and BM1 algorithms. The rightmost point in both curves corresponds to a single instance mean and standard deviation, the 2nd point to two instances, the 3rd three and so on. Each point is computed on 1000 runs on `si6_r001_m400`. Plot is in log-log scale.