Solving the TTC 2014 Movie Database Case with GrGen.NET

Edgar Jakumeit
eja@ipd.info.uni-karlsruhe.de

The task of the Movie Database Case [2] is to identify all couples of actors who performed together in at least three movies. The challenge of the task is to do this fast on a large database. We employ the general purpose graph rewrite system GRGEN.NET in developing and optimizing a solution.

1 What is GrGen.NET?

GRGEN.NET (`www.grgen.net`) is a programming language and development tool for graph structured data with pattern matching and rewriting at its core, which furthermore supports imperative and object-oriented programming, and features some traits of database-like query-result processing.

Founded on a rich data modeling language with multiple inheritance on node and edge types, it offers pattern-based rewrite rules of very high expressiveness, with subpatterns and nested alternative, iterated, negative, and independent patterns, as well as preset input parameters and output def variables yielded to. The rules are orchestrated with a concise control language, which offers constructs that are simplifying backtracking searches and state space enumerations.

Development is supported by graphical and stepwise debugging, as well as search plan explanation and profiling instrumentation for graph search steps – the former helps in first getting the programs correct, the latter in getting them fast thereafter. The tool was built for performance and scalability: its model data structures are designed for fast processing of typed graphs at modest memory consumption, while its optimizing compiler adapts the search process to the characteristics of the host graph at hand and removes overhead where it is not needed.

GRGEN.NET lifts the abstraction level of graph-representation based tasks to declarative and elegant pattern-based rules, that allow to program with structural recursion and structure directed transformation [3]. The mission of GRGEN.NET is to offer the highest combined speed of development and execution available for graph-based tasks.

2 Getting it right

As always, the first step is to get a correct solution specified in the cleanest way possible, and only later on to optimize it for performance as needed.

Task 1: Generating Test Data

The synthetic test set of task 1 is generated with the rules in `MovieDatabaseCreation.grg`, with the iteration `for{i:int in [0:n-1]; createPositive(i) ;> createNegative(i)}` in the sequence definition `createExample`, applying the rules `createPositive` and `createNegative` in succession.The rules are a direct encoding of the patterns in the specification, just in textual GRGEN syntax (as explained with the next task).

To appear in EPTCS.

Task 2: Finding Couples

The workhorse rule for finding all pairs of persons which played together in at least three movies is `findCouples`.

```
rule findCouples
{
  pn1:Person; pn2:Person;
  independent {
    pn1 -:personToMovie-> m1:Movie <-:personToMovie- pn2;
    pn1 -:personToMovie-> m2:Movie <-:personToMovie- pn2;
    pn1 -:personToMovie-> m3:Movie <-:personToMovie- pn2;
  }

  modify {
    c:Couple;
    c -:p1-> pn1;
    c -:p2-> pn2;

    exec(addCommonMoviesAndComputeAverageRanking(c, pn1, pn2));
  }
} \ auto
```

Figure 1: findCouples rule

It specifies a pattern of two nodes `pn1` and `pn2` of type `Person`, and an `independent` pattern which asks for 3 nodes `m1`, `m2`, `m3` of type `Movie`, each being the target of an edge of type `personToMovie` starting at `pn1`, and each being also the target of an edge starting at `pn2`. (Types are given after a colon, the optional name of the entity may be given before the colon, for edges they are inscribed into the edge notation `-->`).

An independent pattern means for one that its content only needs to be searched and is not available for rewriting, and for the other that for each `pn1` and `pn2` in the graph, it is sufficient to find a single instance of the independent, even if the rule is requested to be executed on all available matches – without the independent we would get all the permutations of `m1`, `m2`, and `m3` as different matches.

The rewriting is specified as nested pattern in `modify` mode, which means that newly declared entities will be created, and all entities from the matched pattern kept unless they are explicitly requested to be `deleted`. Here we create a new node `c` of type `Couple`, link it with edges of the types `p1` and `p2` to the nodes `pn1` and `pn2`, and then `exec`ute the helper rule `addCommonMoviesAndComputeAverageRanking` on `c`, `pn1`, and `pn2`. The helper rule is used to create the `commonMovies` edges to *all* the movies *both* played in (you find it in Figure 3 in A.1).

The `auto` keyword after the rule requests GrGen.NET to generate a symmetry filter for matches from automorphic patterns. The pattern is automorphic (isomorphic to itself) because it may be matched with `pn2` for `pn1` and `pn1` for `pn2`.

To get *all* pairs of persons which played together in at least three movies we execute the rule with all-bracketing from a graph rewrite sequence in the GrShell script `MovieDatabase.grs`, filtering away automorphic matches, with the syntax: `exec [findCouples\auto]`.

The language constructs are explained in more detail in the extensive GrGen.NET user manual [1].

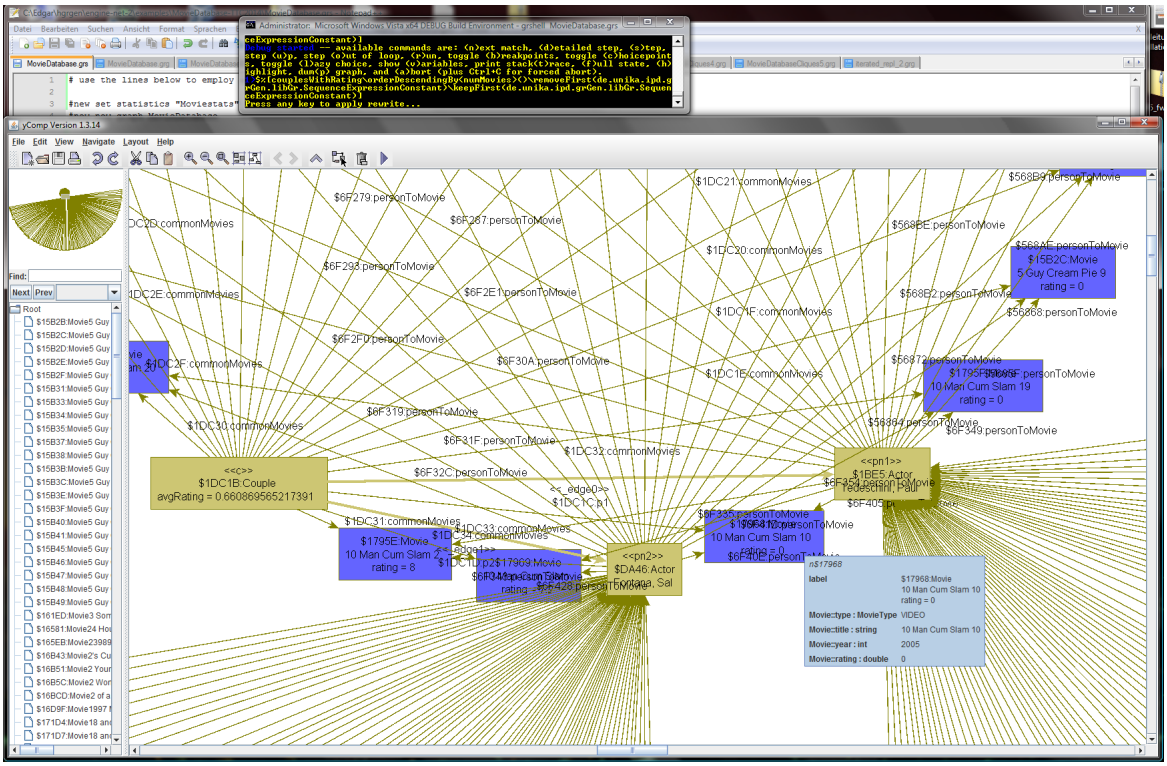

Figure 2: Debugging the top match from the 10000 movies file

Task 3: Computing Average Rankings

The helper rule `addCommonMoviesAndComputeAverageRanking` that was already used for creating the common movies edges is also used for computing the average rankings, you find it in Figure 3 in A.1.

Extension Task 1: Compute Top-15 Couples

The extension task 1 requires to compute the top couples according to the average rating of their common movies, and according to the number of common movies. It is solved with the rule `couplesWithRating` that you find in Figure 4 in A.1. We use the rule twice, ordering the matches differently. The sequence `[couplesWithRating\orderDescendingBy<avgRating>\keepFirst(15)]` executed from the GrShell asks for all matches of the rule `couplesWithRating`, then sorts them descendingly by the `avgRating` pattern variable, then throws away all but the first top 15 matches. The other value of interest is handled in exactly the same way.

You can inspect the found results with the debugger of GrGen.NET, see Figure 2. There, a visualization of the match with most connections is displayed, in layout `Circular`; take a look at `MovieDatabaseLayout.grs` to find out about the visualization configuration applied to reach this. Notably we configured it so it shows only the matched entities plus the one-step connected elements as context, as large graphs are costly (often too costly) to visualize.

Extension Tasks 2, 3, 4

The other extension tasks asking to find cliques of actors are solved with manually coded rules for the sizes 3, 4, and 5 as a straightforward generalization of the couples-based task; higher-order or meta-programming is not supported (and won't be, you have to employ an external code generator if needed).

3 Getting it fast

The clean pattern-based solution presented in the previous chapter unfortunately scales badly with increasing model sizes. Utilizing the built-in search plan explanation and profiling instrumentation, we were able to find out that the most expensive part is the check that there are at least 3 common movies existing, for two persons that are already connected by a common movie. So we replaced it with an imperatively formulated helper function, utilizing hash sets – they scale considerably better for connectedness checking of a large number of adjacent(/incident) elements ($O(n)$ instead of $O(n*n)$).

In addition to the built-in optimizations of independent inlining to ensure that pattern matching follows connectedness, and search planning, adapting the pattern matching process to the characteristics of the host graph at hand, did we apply some further optimizations of the code.

- We used imperatively formulated hash-set lookup instead of the nested-loops implementation of the pattern matcher for the at-least-3-common-movies-check, to cope well with actors with high numbers of movies.
- We even inspect the number of incident edges in this check (yielding an adaptive algorithm), so the smaller hash set is built; in addition, we utilize an incident edges count index maintained by the engine that gives us direct access to the number of incident edges without having to iterate and count them.
- We applied some search space pruning by early filtering during the matching based on the unique id order of the actors, instead of filtering of permuted matches of automorphic patterns afterwards; and by checking for actors with fewer than 3 movies early on, to remove the "long tail" (of actors performing in few movies).
- We separated the computation of the average ranking from the addition of the common movies, this allowed us a reformulation with an all-bracketed rule instead of an iterated pattern, saving us the overhead of the graph parser employed in matching iterated, alternative, and subpatterns.
- We *parallelized* the pattern matcher with a `parallelize` annotation. This does not minimize the amount of work to be carried out for the specification as the other optimizations did, but maximizes the amount of workers thrown on the work. A task with an expensive search like ours – as revealed by profiling – benefits considerably from it.

The final optimized rule is shown in Figure 5, and its helper functions in Figure 6 and Figure 7 (the optimization increased the LOC from about 30 for the original version of task 2 to about 90). The optimization process is described in more detail in [4].

4 Calling from API and Performance Results

The task description asks for a standalone command line version for benchmarking. We supply a C#-Program that employs the GrGen.NET API towards this end, which can be found in `MovieDatabase-Benchmarker.cs` (everything else was coded entirely in the GrGen-languages, and is based on actual

GRGEN.NET-features). It expects as first parameter the name of the rule to apply (`findCouplesOpt`, `findCliquesOf3Opt`, `findCliqueOf4Opt`, `findCliqueOf5Opt`), and as second parameter either the graph to import (e.g. `imdb-0005000-50176.movies.xmi.grs`), or the number of iterations to use in generating the synthetic test graph. An additional sequence may be given in quotes, intended for emitting the sorted results.

The standalone version contains a further optimization that can be only applied on API level. After importing the IMDB graphs, it reduces the named graphs to unnamed graphs, throwing away the name information, which saves us a considerable amount of memory (and cache).

On a Core i7 920 with 24GiB, running Windows with MS .NET, the largest synthetic benchmark graph was processed in about 65sec for the couples, the 3 cliques in about 40sec, the 4 cliques in about 36sec, and the 5 cliques in about 20sec [1]. The entire IMDB was processed on average in about 500sec (searching for all couples performing in at least three movies, linking them to all of their common movies). The cliques (which were outside of the main focus of our optimization work) show a stunning computation time explosion, from 5 secs for cliques of 3, over something less than 2 minutes for cliques of 4, to more than 2 hours for cliques of 5, already on the smallest IMDB graph.

The official results (in seconds) measured on an Opteron 8387 with 32GiB, running LINUX with mono are given in Table 1, for the time needed to synthesize the models and the time needed for processing the couples on the synthesized models, and in Table 2, for the time needed for processing the couples and 3-cliques on the IMDB models.

5 Conclusion

We first specified a clean and simple solution of the movie database task in the GRGEN languages, then we optimized it for performance. The tool supported us in validating the solution with its graphical debugger, and in optimizing it with its search plan explanation and profiling instrumentation for search step counting.

GRGEN.NET search planning can be compared to searching straw stars on a freshly harvested field, by looking at the places where the ground is only slightly covered, only reaching into the haystacks when they can't be circumvented at all. A pattern matcher is generated based on the assumption that search planning worked well in circumventing those haystacks. Here the task consists solely of diving within a hay stack, a particularly large and interwoven one. So we had to supplement the declarative patterns (implemented with loop-nesting behind the scenes, which is optimal for sparse graphs) by imperative hash set based helper functions – at least this highlights that for about any task built on a graph-based representation there are the language constructs available that are needed for solving it.

Once again, GRGEN.NET is among the top performing tools.

References

[1] Jakob Blomer, Rubino Geiß & Edgar Jakumeit (2014): *The GrGen.NET User Manual*. `http://www.grgen.net/GrGenNET-Manual.pdf`.

[2] Tassilo Horn, Christian Krause & Matthias Tichy (2014): *The TTC 2014 Movie Database Case*.

[3] Edgar Jakumeit (2011): *EBNF and SDT for GrGen.NET*. Technical Report. Available at `http://www.info.uni-karlsruhe.de/software/grgen/EBNFandSDT.pdf`. Presented at AGTIVE 2011.

[1] picking the fastest of 3 runs, with considerable variations in between

```
rule addCommonMoviesAndComputeAverageRanking(c:Couple, pn1:Person, pn2:Person)
{
  iterated it {
    pn1 -:personToMovie-> m:Movie <-:personToMovie- pn2;

    modify {
      c -:commonMovies-> m;
      eval { yield sum = sum + m.rating; }
    }
  }

  modify {
    def var sum:double = 0.0;
    eval { c.avgRating = sum / count(it); }
  }
}
```

Figure 3: addCommonMoviesAndComputeAverageRanking rule

[4] Edgar Jakumeit (2014): *Optimizing a Solution for the TTC 2014 Movie Database Case*. Technical Report. Available at `http://www.info.uni-karlsruhe.de/software/grgen/OptimizingMovie.pdf`.

[5] Edgar Jakumeit (2014): *SHARE demo related to the paper Solving the TTC 2014 Movie Database Case with GrGen.NET*. `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_TTC14_64bit_grgen_v2.vdi`.

A Appendix

A.1 Getting it right

Task 2 and 3

The helper rule `addCommonMoviesAndComputeAverageRanking` shown in Figure 3 eats all common movies with an `iterated` pattern which is matched as often as possible; in the rewrite part it links the `Couple` node to each such movie with a `commonMovies` edge. In the `eval` part used for attribute evaluation, the `avgRating` is computed as the sum of the ratings of the movies munched, divided by the `count` of the `iterated`s matched. The `def var` is used to define a variable whose content is computed *after matching* from the matched entities, the `yield` is used to assign to such variables, from nested patterns up to their containing patterns (normally variables are passed the other way round, from nesting to nested patterns, following the flow of matching).

Extension Task 1: Compute Top-15 Couples

The rule `couplesWithRating` in Figure 4 matches a `Couple` and its linked `Persons`. Two `def` variables `avgRating` and `numMovies` for the values of interest are created and filled with the average rating stored in the couple nodes, and the number of movies as computed from the size of the set of `commonMovies` edges outgoing from the couple node. We employ a `yield` block to assign the variables (bottom-up)

```
rule couplesWithRating
{
  c:Couple;
  c -:p1-> pn1:Person;
  c -:p2-> pn2:Person;

  def var avgRating:double;
  def var numMovies:int;
  yield {
    yield avgRating = c.avgRating;
    yield numMovies = outgoing(c, commonMovies).size();
  }

  modify {
    emit("avgRating:␣" + avgRating + "␣numMovies:␣" + numMovies
      + "␣by␣" + pn1.name + "␣and␣" + pn2.name + "\n");
  }
} \ orderDescendingBy<avgRating>, orderDescendingBy<numMovies>
```

Figure 4: couplesWithRating rule

after the (top-down) pattern matching completed. In the rewrite part specified in `modify` mode we just `emit` the values of interest. Furthermore, we request GRGEN to generate sorting code for the def pattern entities `avgRating` and `numMovies` with the declaration of the auto-generated matches accumulation filters `orderDescendingBy<avgRating>` and `orderDescendingBy<numMovies>`.

A.2 Getting it fast

Figure 5 shows the rule `findCouplesOpt` we get after the final optimization step. Figure 6 shows the helper function `atLeastThreeCommonMovies` we employ to find out if there are 3 common movies, and Figure 7 shows the helper function `getCommonMovies` we employ to compute the common movies. Note that the `adjacentOutgoing` function is built-in.

A.3 Calling from API and Performance Results

In Table 1 and Table 2 you find the official measurements.

```
rule findCouplesOpt[parallelize=16]
{
  pn1:Person; pn2:Person;
  hom(pn1,pn2);
  independent {
    pn1 -p2m1:personToMovie-> m1:Movie <-p2m2:personToMovie- pn2;
    hom(pn1,pn2); hom(p2m1,p2m2);
    if{ atLeastThreeCommonMovies(pn1, pn2); }
  }
  if{ uniqueof(pn1) < uniqueof(pn2); }
  if{ countPersonToMovie[pn1] >= 3; }
  if{ countPersonToMovie[pn2] >= 3; }

  def ref common:set<Node>;
  yield {
    yield common = getCommonMovies(pn1, pn2);
  }

  modify {
    c:Couple;
    c -:p1-> pn1;
    c -:p2-> pn2;

    eval {
      for(movie:Node in common) {
        add(commonMovies, c, movie);
      }
    }
  }
}
```

Figure 5: findCouplesOpt rule

```
function atLeastThreeCommonMovies(pn1:Person, pn2:Person) : boolean
{
  if(countPersonToMovie[pn1] <= countPersonToMovie[pn2]) {
    def var common:int = 0;
    def ref movies:set<Node> = adjacentOutgoing(pn1, personToMovie);
    for(movie:Node in adjacentOutgoing(pn2, personToMovie))
    {
      if(movie in movies) {
        common = common + 1;
        if(common >= 3) {
          return(true);
        }
      }
    }
  } else { /* pn1 and pn2 reversed */ }
  return(false);
}
```

Figure 6: atLeastThreeCommonMovies helper function

```
function getCommonMovies(pn1:Person, pn2:Person) : set<Node>
{
  def ref common:set<Node> = set<Node>{};
  if(countPersonToMovie[pn1] >= countPersonToMovie[pn2]) {
    def ref movies:set<Node> = adjacentOutgoing(pn2, personToMovie);
    for(movie:Node in adjacentOutgoing(pn1, personToMovie))
    {
      if(movie in movies) {
        common.add(movie);
      }
    }
  } else { /* pn1 and pn2 reversed */ }
  return(common);
}
```

Figure 7: getCommonMovies helper function

Table 1: Synthetic model generation and matching couples

N	synthesizing	matching couples
1000	0.13	0.25
2000	0.21	0.42
3000	0.30	0.59
4000	0.55	0.58
5000	0.62	0.96
10000	1.22	1.82
50000	7.29	9.58
100000	15.88	22.09
200000	34.20	51.13

Table 2: Matching couples and 3-cliques

nodes	matching couples	matching 3-cliques
49930	0.52	5.011
98168	0.99	13.634
207420	1.47	22.794
299504	4.14	33.480
404920	3.51	53.659
499995	9.27	69.194
709551	11.10	129.696
1004463	22.45	222.112
1505143	62.68	797.266
2000900	190.83	1930.327
2501893	318.72	4480.044
3257145	683.66	15616.83