Optimizing a Solution for the TTC 2014 Movie Database Case

Edgar Jakumeit

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 esp. optimizing a solution. You are interested in this paper if you want to learn how to optimize GRGEN.NET for performance.

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 query-result processing.

Founded on an 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 enumeration.

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]. Employing GRGEN.NET one is able to achieve the highest combined speed of development and execution available for graph-based tasks.

2 Data transformation

The input supplied for the case is in XMI format, following the EMF modeling. This means esp. that in between `Movie` and `Person` nodes we find `movies` and `persons` references resp. edges, always a pair of them, one in the opposite direction of the other. These twins offer no addition information, they just allow to find the opposite node either way. This would be wasteful for GRGEN whose edges are always navigable in both directions.

So as a preparatory step we apply a model-to-text transformation, mapping the XMI models supplied to GRS files, the native serialization format of GRGEN.NET. The central rule `create_personToMovie` in `GrgenifyMovieDatabase.grg` emits a new edge command for a `personToMovie` edge upon visiting a pair of `movies` and `persons`. The transformation is applied on all input files with a shell script `GrgenifyMovieDatabase.sh` containing a call to the GRSHELL with a here-document specifying the workflow. In the following steps we then import the GRS files with the single edges.

A further benefit from this model mapping are the cleaner names of the manually specified model, compared to the model that is auto-generated when the ecore/XMI is imported. The latter mechanism applies name mangling – which is needed to prevent name clashes of global edge types stemming from multiple equally named references that are contained in the type of their node, and for preventing collisions with GRGEN.NET keywords.

Removing the edges reduces memory consumption and improves cache utilization, the gains are considerable when working with millions of objects [1], since GRGEN always implements directed, typed, multi-graphs with/for each of its entities, storing them in a system of ringlists, which allows for fast pattern matching and rewriting, with lookups by type in constant time, accesses of incident elements in constant time, and graph additions and removals in constant time.

3 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.

Main rule (Tasks 2 and 3)

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,

[1] The memory consumption of a GRGEN.NET edge without attributes is 52 bytes using a 32bit CLR (9 pointers a 4 bytes, a 4 bytes flags field/bitvector, a 4 bytes unique id field, plus 8 bytes .NET object overhead), it is 96 bytes for a 64bit CLR (9 pointers a 8 bytes, a 4 bytes flags field, a 4 bytes unique id field, plus 16 bytes .NET object overhead). A node without attributes uses up 36 bytes, or 64 bytes on 64bit. Attributes only add the .NET overhead of their implementation type.

```
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 2: addCommonMoviesAndComputeAverageRanking rule

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.

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 helper rule `addCommonMoviesAndComputeAverageRanking` shown in Figure 2 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).

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

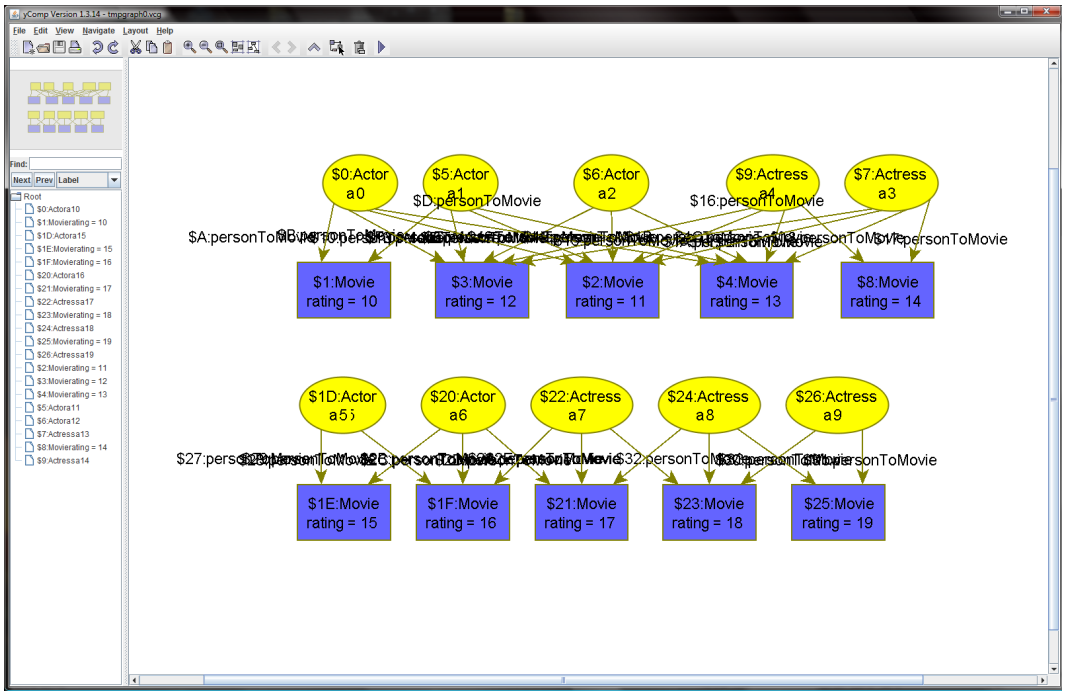

Figure 3: The smallest synthetic model in the graph viewer

Test data and Top Couples (Task 1 and Extension Task 1)

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.

You can check they are correctly generated with the graph viewer supplied with GRGEN.NET. In Figure 3 you see the smallest generated synthetic model, with actors colored yellow and movies colored blue.

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. 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.

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) 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>`.

You can inspect the found results with the debugger of GRGEN.NET. It highlights the matched

```
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

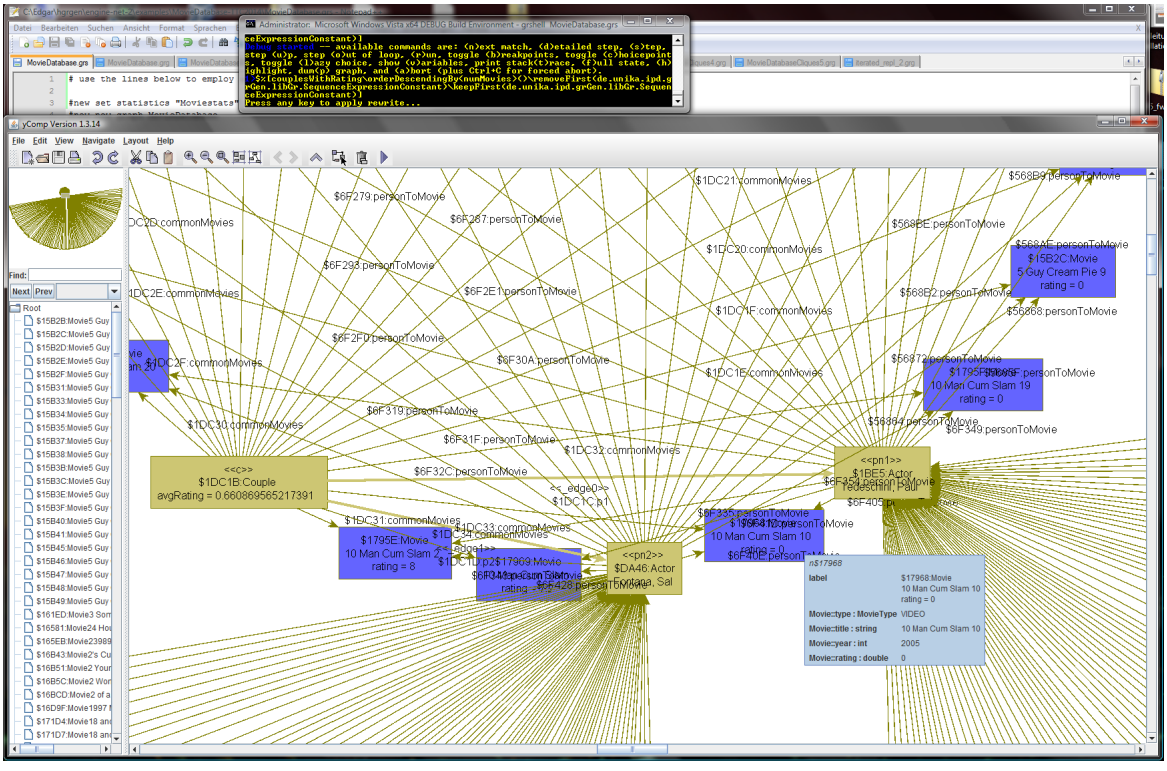

Figure 5: Debugging the top match from the 10000 movies file

pattern of the (currently executed) rule in the current graph, and displays the effects of the rewriting step before it leads to the follow-up graph. This only works with the full graph for the smaller auto-generated and the smallest IMDB-derived graph, though. The larger graphs are too costly to visualize (which simply poses a general problem of large graphs irrespective of the chosen viewer – besides you would get lost in a haystack). But we can configure the debugger with `dump add graph exclude` to show only the matched entities, plus the one-step connected elements as context. This way we can still inspect what GRGEN found. In Figure 5 can you see a visualization of the match with most connections, in layout `Circular` (take a look at `MovieDatabaseLayout.grs` to find out about the visualization configuration applied to reach this).

We achieved Figure 5 by executing the sequence
`$%[couplesWithRating\orderDescendingBy<numMovies>\removeFirst(0)\keepFirst(1)]`
in `debug` mode. First all matches are sought, then they are ordered descendingly alongside the number of common movies, then the first 0 matches (i.e. nothing) are removed – increase this value to k to inspect the kth match from the matches found. Afterwards,all but the first 1 remaining match is removed – increase this value to k to inspect the first k remaining matches at once. The `$` chooses one match from all matches found that remain after sorting and filtering, the `%` allows you to cycle through the matches in the debugger.

Cliques (Extension Tasks 2 and 3 and 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).

4 Getting it fast

The most important point to understand when optimizing for speed is that the expensive task is the search carried out during pattern matching.

Searching is carried out with a backtracking algorithm following a search plan in a fixed order, binding one pattern element after another to a graph element, checking if it fits to the already bound parts. If it does fit search continues trying to bind the next pattern element (or succeeds building the match object from all the elements bound if the last check succeeds), if it does not fit search continues with the next graph element; if all graph element candidates for this pattern element are exhausted, search backtracks to the previous decision point and continues there with the next element.

Typically, the first pattern element is determined with a lookup operation reaching into the graph, binding the pattern element to a graph element of the required type (the less elements of that type exists, the better); then neighbouring elements are traversed following the graph structure (the less neighbouring elements exists, the better), until a match of the entire pattern is found.

Initial and Host-Graph-Adapted Search Plans

For each pattern, GRGEN.NET creates a static un-optimized search plan (automatically, behind the scenes). You can inspect it for each rule with the `custom actions explain` command. For `findCouples` it delivers the output displayed in Figure 6.

The search plan in Figure 6 shows us that the graph is entered with a lookup of a `personToMovie` edge, then the candidates for the source node `pn1` and the target node `m1` are extracted from it. Afterwards

```
static search plans
findCouples:
    lookup -_edge0_inlined_idpt_0:personToMovie-> in graph
    from <-_edge0_inlined_idpt_0- get source pn1:Person
    from -_edge0_inlined_idpt_0-> get target m1_inlined_idpt_0:Movie
    from m1_inlined_idpt_0 incoming <-_edge1_inlined_idpt_0:personToMovie-
    from <-_edge1_inlined_idpt_0- get source pn2:Person
    independent {
        (preset: pn1)
        (preset: m1 after independent inlining)
        (preset: pn2)
        (preset: _edge0 after independent inlining)
        (preset: _edge1 after independent inlining)
        from pn1 outgoing -_edge2:personToMovie->
        from -_edge2-> get target m2:Movie
        from pn1 outgoing -_edge4:personToMovie->
        from -_edge4-> get target m3:Movie
        from pn2 outgoing -_edge3:personToMovie->
        from pn2 outgoing -_edge5:personToMovie->
    }
```

Figure 6: Initial search plan

the other `personToMovie` edge is matched in reverse from the movie to `pn2`. Several elements in the independent are handed in as already matched presets from the outer pattern, then the `personToMovie` edges are taken from `pn1` to the movies `m2` and `m3`, and finally the `personToMovie` edges from `pn2` are matched, with an implicit check that the target movie is the same as the one already matched.

We see here an automatically applied optimization, the movie `m1` was inlined from the independent pattern to its containing pattern. Without this optimization, `pn1` and `pn2` would have to be enumerated in the main pattern *unconnected*, resulting in the unfolding of the cartesian product of all `Person` nodes, before handing it in to the matcher of the nested independent pattern to purge the actors without a connecting movie. This is a crucial optimization that saves us from the execution time explosion of the $O(n*n)$ cartesian product.

We can do better with search plans adapted to the host graph, towards this end we issue a `custom graph analyze` followed by a `custom actions gen_searchplan` command. The first computes some information about the multiplicities of the types and esp. the multiplicities of the connections in between the types from the host graph, the latter re-generates the matchers at runtime based on those graph characteristics gathered, replacing the auto-generated ones. For the small IMDB graphs, this results in a lookup of `m1`, extending on the one side to `pn1`, and on the other to `pn2`. For the large IMDB graph this results in a lookup of `pn1`, extending to `m1`, then extending to `pn2`. This leads to a small runtime improvement, esp. together with a later attribute condition (for many tasks, the gains are higher).

Profiling, Big-O, and Pruning

Things are getting really slow with increasing graph size and esp. connectedness of the IMDB graphs. We must go into greater details – let's start profiling. We do so with the `new set profile on` command at the begin of the shell file before the graph is created, and a `show profile findCouples` command thereafter. Profiling shows us a massive increase in the number of search steps with the graph sizes, hinting at an $O(n*n)$ algorithm: the 207,876 nodes graph requires 1,259,006,408 search steps, executed in a bit below 9 seconds; the 405,592 nodes graph requires 9,541,314,556 search steps, executed in a bit below 46 seconds.

Inspecting the top connected matches visually and aligning them with the search plan we understand why: the pattern matcher has to follow all edges from `pn1` on and all edges from `pn2` on, to find and bind the common movies, giving us an $O(n*n)$ iteration (with n being the number of adjacent movies). The iteration is not much of an issue for actors with few connections (i.e. sparse graphs), but gets very expensive for massively connected actors. Here we simply have to revert to a non-pattern based approach to compute the set of common nodes. You see it displayed in Figure 11 in section 6 with an intersection of two sets (delivered by the built-in `adjacentOutgoing` function), which is $O(n)$ because the hash set lookup employed by the intersection operator `&` is $O(1)$.

We apply a further optimization that prunes the search space earlier. Instead of finding and materializing the automorphic matches, just to remove them later on with an automorphic matches filter, we reject them straight ahead. To this end, we require that the unique id of the actor bound to `pn1` is smaller than the unique id of the actor bound to `pn2`. Unique ids are unique integer numbers assigned by GrGen.NET after they have been requested with a `node edge unique` declaration in the model (they render deletion a bit more expensive, which is why they are not active by default).

Furthermore, we optimize `addCommonMoviesAndComputeAverageRanking`. Instead of iterating all common movies from within one rule application, computing the averages from the iterated patterns, do we only add the common movies in this step without averaging (more is not requested), with an all-bracketed application of `addCommonMovies` (cf. Figure 12 in section 6) which saves us the overhead of the graph parser employed in matching iterated, alternative, and subpatterns. Whose costs are not huge, but they can be felt.

After the optimization, the 207,876 nodes graph requires 42,195,046 search steps, executed in a bit above 10 seconds. The 405,592 nodes graph requires 110,809,701 search steps, executed in a bit above 22 seconds. We get a considerable decrease in search steps, and esp. a considerably slowed down increase with increasing graph size and connectedness. The larger graph is processed twice as fast, but the smaller one is in fact processed a bit slower – hash set based connectedness checking only pays off for high connectedness degrees.

Further Pruning and Strength Reduction

The set intersection already helped regarding the big-O, we still can do better: only 3 common movies are needed, so we replace it with a dedicated algorithm that iterates one set and checks whether the movies are available in the other, leaving early after 3 such movies were found. In excess, this algorithm only needs to allocate and fill one single set. You can find this helper `function atLeastThreeCommonMoviesIntermediateOpt2` in Figure 14 in section 6.

It takes two `Persons` `pn1` and `pn2` as arguments and returns `true` if they are linked by at least three common movies, and returns `false` otherwise. For each movie, it checks with a hash set lookup in $O(1)$ whether it is common to the persons. We even use an adaptive helper function here; we first find out

```
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 7: findCouplesOpt rule

what is the node with the smaller amount of `adjacent Movie` nodes by `counting` them, before we build the hash set from that smaller neighbourhood. The extra effort of counting pays off, hash set allocating and building is expensive (which is why we don't use hash sets in implementing the pattern matchers, in most cases they lead to a considerable slowdown compared to the nested loops employed there).

We prune the search space still further by exploiting the "long tail", the fact that there are many actors existing that performed in less than 3 movies (in contrast to the massively connected actors that cause the major pattern matching issues). So we check in the resulting rule `findCouplesIntermediateOpt2` given in Figure 13 in section 6 for actors with at least 3 adjacent movies.

After this optimization round, we're at about 17 seconds and 146,941,859 search steps for the 405,592 nodes graph and at about 6.5 seconds for the smaller one.

Index usage and Parallelization

In the previous section we introduced quite some adjacent node counting for its adaptability (equaling incident edge counting; causing an increase of the search steps). We can do better here with an incident edges counting `index`. GrGen supports attribute indices for quick access to graph elements based on their attribute values, and incidence count indices for quick access to nodes based on their number of incident edges, maintained in the background by the engine. Here we declare an incident edge index

`countPersonToMovie` in the model file, and use it in the rules with `countPersonToMovie[element]` to get $O(1)$ access to the needed counts instead of having to iterate the list of incident edges, counting them.

Until now we were concerned with minimizing the amount of work to be carried out for the specification. But with our-days multicore machines there is an alternative strategy available: maximize the amount of workers thrown on the work. Applying both as possible will of course yield the best results.

Parallelization brings locking and work distribution overhead with it, so it is not for free performance-wise. For tasks with only little search work to be carried out it makes things run *slower*. But a task with an expensive search like ours – remember the large number of search steps shown by profiling – benefits considerably from it.

You apply parallelization in GRGEN.NET with a `parallelize` annotation at a rule, specifying the maximum amount of worker threads to use. Only the matcher is parallelized [2] , and only the first search operation which causes a splitting in the search space, i.e. a lookup that splits work alongside all elements of the type requested, or an extension with a edge that splits work alongside all incident edges.

Figure 7 shows the rule `findCouplesOpt` we get after the final optimization step, Figure 8 shows the helper function `atLeastThreeCommonMovies` we employ to find out if there are 3 common movies, and Figure 9 shows the helper function `getCommonMovies` we employ to compute the common movies. The search plan of the final rule is shown in Figure 10.

You see a further minor optimization in the resulting rule with application of `hom`-declarations, they allow the pattern elements to match the same graph elements. The default is isomorophy, which needs to be checked. This is done efficiently, but here we want to squeeze out as much as possible, so we remove the checks – we can do so because the unique id checking already ensures that the pattern elements are not matched to the same host graph elements.

After this final optimization round, we're at 42,271,513 search steps, executed in a bit above 5.5 seconds on a Core i7 920, for the 405,592 nodes graph.

5 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`. 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. The GRSHELL always employs named graphs, they supply the names that can be seen in the debugger, and the names that are written in `grs`-serialization and read in again in de-serialization. The name information consists of a hash map from names to graph elements, and one hash map from graph

[2] The full transformation including rewriting won't be parallelized as it was done in the prototype employed in [5] because of the in our eyes too bad cost-benefit-ratio: the task where most benefits can be gained is searching for a match, a parallelization of rewriting would render the programming model to the user much more complicated, and would require graph partitioning (locking would eat up all performance gains), which is a difficult problem on its own, and esp. so for a general-purpose tool that must work for many different graph representations.

```
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 8: 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 9: getCommonMovies helper function

```
findCouplesOpt:
    def findCouplesOpt_var_common
    parallelized lookup m1_inlined_idpt_0:Movie in graph
    from m1_inlined_idpt_0 incoming <-p2m1_inlined_idpt_0:personToMovie-
    from <-p2m1_inlined_idpt_0- get source pn1:Person
    if { depending on findCouplesOpt_node_pn1, }
    from m1_inlined_idpt_0 incoming <-p2m2_inlined_idpt_0:personToMovie-
    from <-p2m2_inlined_idpt_0- get source pn2:Person
    if { depending on findCouplesOpt_node_pn2, }
    if { depending on findCouplesOpt_node_pn1,findCouplesOpt_node_pn2, }
    independent {
        (preset: pn1)
        (preset: m1 after independent inlining)
        (preset: pn2)
        if { depending on findCouplesOpt_node_pn1,findCouplesOpt_node_pn2, }
        (preset: p2m1 after independent inlining)
        (preset: p2m2 after independent inlining)
    }
```

Figure 10: Final search plan

elements to names. That sounds rather innocent, but requires about as much storage space as the plain GRGEN-graph, which accumulates to gigabytes for the large test graphs. Throwing this information away reduces the maximum memory requirements considerably, saves the engine from name maintenance on inserts, and allows for better cache utilization (depending on the allocation pattern of the .NET runtime).

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 [3]. 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.

6 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

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

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

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 its flexibility, that for about any task built on a graph-based representation there are the language constructs available that are needed for solving it.

Over the course of the stepwise optimizations, all features and language constructs of relevance for performance optimization were introduced, with the search plans and inlining used by the engine for implementing the patterns, with conditions inserted for early pruning and their effect on the profiling outcome, with imperative helper code and hash-sets employed for this special multipath connectedness checking task, with indices for quick access at the price of increased memory consumption, and with declarative matcher parallelization applied with a single annotation. Those features and constructs should allow you to optimize a solution you specified in GRGEN.NET for performance, too.

Once again, GRGEN.NET is among the top performing tools –
it keeps solutions affordable performance-wise, while lifting the level of abstraction to graph rewriting.

References

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

[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 `www.info.uni-karlsruhe.de/software/grgen/EBNFandSDT.pdf`. Presented at AGTIVE 2011.

[4] 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`.

[5] Jochen Schimmel, Tom Gelhausen & Christoph A. Schaefer (2009): *Gene Expression with General Purpose Graph Rewriting Systems*. In: *Proceedings of the 8th GT-VMT Workshop*, *Electronic Communications of the EASST* 18. Available at `journal.ub.tu-berlin.de/eceasst/article/view/276/259`.

A Appendix

```
rule findCouplesIntermediateOpt1
{
  pn1:Person; pn2:Person;
  independent {
    pn1 -:personToMovie-> m1:Movie <-:personToMovie- pn2;
    if{ (adjacentOutgoing(pn1, personToMovie) &
         adjacentOutgoing(pn2, personToMovie)).size() >= 3; }
  }
  if{ uniqueof(pn1) < uniqueof(pn2); }

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

    exec( [addCommonMoviesIntermediateOpt(c, pn1, pn2)] );
  }
}
```

Figure 11: 1st optmization step: findCouplesIntermediateOpt1

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

  modify {
    c -:commonMovies-> m;
  }
}
```

Figure 12: addCommonMoviesIntermediateOpt helper procedure

```
rule findCouplesIntermediateOpt2
{
  pn1:Person; pn2:Person;
  independent {
    pn1 -:personToMovie-> m1:Movie <-:personToMovie- pn2;
    if{ atLeastThreeCommonMoviesIntermediateOpt2(pn1, pn2); }
  }
  if{ uniqueof(pn1) < uniqueof(pn2); }
  if{ countAdjacentOutgoing(pn1, personToMovie) >= 3; }
  if{ countAdjacentOutgoing(pn2, personToMovie) >= 3; }

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

    exec( [addCommonMoviesIntermediateOpt(c, pn1, pn2)] );
  }
}
```

Figure 13: findCouplesIntermediateOpt2 helper function

```
function atLeastThreeCommonMoviesIntermediateOpt2(pn1:Person, pn2:Person) : boolean
{
  if(countAdjacentOutgoing(pn1, personToMovie) <= countAdjacentOutgoing(pn2, personToMovie))
  {
    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 14: atLeastThreeCommonMoviesIntermediateOpt2 helper function