GKC: A Reasoning System for Large Knowledge Bases

Tanel Tammet[✉]

Tallinn University of Technology, Tallinn, Estonia
tanel.tammet@taltech.ee

Abstract. This paper introduces GKC, a resolution prover optimized for search in large knowledge bases. The system is built upon a shared memory graph database Whitedb, enabling it to solve multiple different queries without a need to repeatedly parse or load the large parsed knowledge base from the disk. Due to the relatively shallow and simple structure of most of the literals in the knowledge base, the indexing methods used are mostly hash-based. While GKC performs well on large problems from the TPTP set, the system is built for use as a core system for developing a toolset of commonsense reasoning functionalities.

Keywords: Automated reasoning · Knowledge base

1 Introduction

We present the first release of GKC (acronym for "Graph Knowledge Core"), version 0.1: an automated theorem prover for first order logic, optimized for handling large knowledge bases. GKC is intended to become a building block for developing specialized methods and strategies for commonsense reasoning, including nonmonotonic reasoning, probabilistic reasoning and machine learning methods. We envision natural language question answering systems as the main potential application of rule-applying commonsense reasoning methods.

The immediate focus of GKC is implementing core technologies for efficient handling of large knowledge bases like DBpedia, YAGO, NELL and OpenCyc. The reason for this focus is the recognition that any useful commonsense reasoning system would necessarily contain a very large set of facts and rules, most of which have a relatively simple structure.

Due to the complexities of handling large knowledge bases, the most common current approach is to rely on specialized query tools like SPARQL or reasoning systems built for RDFs or some restricted subset of OWL, focusing mostly on taxonomies, while complex relations between different objects are rarely handled. On the other hand, the natural language question answering systems currently rely mainly on so-called shallow reasoning using methods based on word vectors.

Our hypothesis is that enabling efficient "deep" reasoning with more complex rules would complement and significantly enhance the capabilities of commonsense reasoning systems. This approach has been pursued in several papers and implementations. In particular, we note the papers [4,6,8,15,17].

GKC is undergoing active development and the current version 0.1 of GKC is the first public release of the system [18]. Hence the set of implemented search strategies and indexing technologies is limited, with heavy stress on implementing efficient core algorithms. The main algorithmic and technical features of GKC are:

1. The use of a shared memory database to enable fast startup of proof search and independent parallel searches on the same knowledge base.
2. Using hash indexes instead of tree indexes.
3. Efficient clause-to-clause subsumption algorithm using hashes of ground literal structure.
4. Simplification of derived clauses by hash-based search of contradicting units from all derived clauses.
5. Using several separate queues for picking clauses for the set of support strategy.

We expect GKC to become significantly stronger in near future. We have already started experiments with building nonmonotonic and probabilistic reasoning on a separate layer on top of GKC. However, the GKC system itself will be developed as a conventional first order reasoner without including the experimental features.

2 Architecture

GKC is a conventional first order theorem prover with the architecture and algorithms tuned for large problems. It is implemented in C on top of the data structures and functionality of the shared memory database WhiteDB [14]. GKC is available for both Linux and Windows under GNU AGPLv3 and is expected to be highly portable, see [18].

The prover is run from the command line, indicating the command, the input file, the strategy selection file using JSON syntax and optionally the parsed and prepared knowledge base as a shared memory handle number. Both a detailed configurable running log and a detailed proof is printed for each solution found.

The shared memory database makes it possible to start solving a new problem in the context of a given knowledge base very quickly: parsing and preprocessing the large knowledge base can be performed before the GKC is called to do proof search: the preprocessed database can be assumed to be already present in the memory. Since GKC does not write into the shared database during search, several GKC-calling processes can run simultaneously without locking while using the single copy of a knowledge base in memory. The memory database can be dumped and read from the disk and there could be multiple memory databases present in shared memory simultaneously.

WhiteDB is a lightweight NoSQL database library written jointly by the author of this paper and Priit Järv as a separate project. It is available under the GPL licence and can be compiled from C source or installed as a Debian package. All the data is kept in main memory and can be dumped and read back to and from the disk as a whole. There is no server process, data is read and written directly from/to shared memory, no sockets are used between WhiteDB and the application program. Since data is kept in shared memory, it is accessible to separate processes.

GKC implements a parser for the [1] first order formula syntax. The parser is implemented using GNU tools Flex and Bison. There is also a parser for the Otter clause normal form syntax and a simplified Prolog syntax. The parsed formula is converted to a clause normal form (CNF) using both ordinary CNF conversion rules and replacement of subformulas with introduced predicates in cases there is a danger of the CNF size exploding, using a simple top-down procedure: any time we see that the distribution should be applied to $(a\&b) \vee (c\&d)$, renaming is performed.

3 Algorithms, Strategies and Optimizations

The derivation rules currently implemented in GKC are very basic. The preference stems from our focus of optimizing for the set of support strategy (see [9] for core terminology).

1. Binary resolution with optionally the set of support strategy, negative or positive ordered resolution or unit restriction.
2. Factorization.
3. Paramodulation with the Knuth-Bendix ordering.

In particular, we note that neither the demodulation, hyperresolution, unit-resulting resolution nor purely propositional methods like AVATAR have been implemented so far. We plan to add these rules, but they have not been a priority for GKC development.

The overall iteration algorithm of GKC is based on the common *given-clause algorithm* where newly derived clauses are pushed into the *passive list* and then selected (based on the combination of creation order and clause weight) as a *given clause* into an *active list*. The derivation rules are applied only to the given clause and active clauses. In the following we explain how we perform clause simplification with all the derived clauses present in the active list. We will also explain the use of different clause selection queues used by GKC.

The *set of support strategy* we rely upon for large problems basically means that the large knowledge base is immediately put into the active list and no direct derivations between the clauses in the knowledge base are performed. As an additional limitation we do not perform subsumption of given clauses with non-unit clauses in the knowledge base: during our experiments the time spent for this did not give sufficient gains for the efficiency of proof search.

3.1 Hash Indexes and Their Use

Perhaps the most interesting innovation in GKC is the pervasive use of hash indexes instead of tree indexes. In contrast, all state-of-the-art provers implementing resolution rely on tree indexes of various kinds. Research into suitable tree indexes has been an important subfield of automated reasoning: see [12] for an early overview of common indexing techniques.

Our experiments demonstrate that hash indexes are a viable alternative and possibly a superior choice in the context of large knowledge bases. The latter are expected to consist mostly of ground clauses representing known "facts" and a significant percentage of derived literals are ground as well.

Hash indexes are particularly well suited for ground literals. The motivation for using hash indexes can be summed up as:

1. Hash indexes take up much less space than tree indexes.
2. Hash indexes are possibly faster for our primary scenario of large knowledge bases with a shallow term structure, although confirming this hypothesis would need further research.
3. Hash indexes are simpler to implement than tree indexes.

A hash index of a term or an atom is an integer. We compute the hash of a term by sequentially adding the hashes of constants and variables (`hash` in the formula) to the previous value with a popular addition function *sdbm* developed for strings: `value + (hash << 6) + (hash << 16) - hash`. The same iterative function is used for calculating hashes of constants interpreted as strings, to be used in the hash calculation of the term. These hashes are cached in the data structure of the constant to avoid regular recomputation. A hash of a variable is based on the order of a first variable occurrence in the clause: i.e. if hashes of two literals in two separate clauses are equal, then the literals are equal modulo renaming the variables in the whole clause.

As a hash index we use a simple integer array, elements of which point to hash chains. We currently use the arrays with a length of one million. A minor drawback of this approach is the fact that the hash arrays have to be zeroed during the initialization phase of proof search, which takes time proportional to the array size.

The first important use of hash indexes in GKC is simplifying the newly derived clauses. Each derived clause immediately undergoes simplification and subsumption attempts by looking for existing unit clauses in the passive or active clause list, either deleting some of its literals or subsuming the clause. For this we search for exactly equal literals with the same or negative polarity by looking up the atoms in the hash index. Each match is followed by an actual equality check with atoms in the hash chain. Every unit clause derived is pushed to this hash index. We note that the simplification algorithm can also cut off literals containing variables.

GKC uses an analogous hash index for forward subsumption of newly selected given clauses with unit clauses from the list of active clauses.

Finally, we use hashes of head predicate and function symbols while processing a given clause for the purpose of looking for literals to resolve and paramodulate upon and terms to paramodulate into.

3.2 Hash Features for Clause-to-clause Subsumption

Efficient clause-to-clause subsumption is important for forward subsumption of long clauses. Deriving and using long clauses is very common for the set of support strategy crucial for large knowledge bases. In our experiments the forward subsumption of newly given clause by a given-clause algorithm dominated the time spent on search until we implemented hash features. We note that GKC performs full subsumption only for given clauses, and not for newly derived clauses: for the latter we only perform fast hash-based subsumption with units for memory conservation. Also, we do not perform full self-subsumption of the axiom set for the set-of-support strategy.

It is well known that the tree indexing methods do not perform well for subsumption of long clauses. Earlier work has presented algorithms for efficiently filtering out impossible subsumption cases by feature vectors, see [3] and [13].

The well-known core idea for checking whether a clause A could subsume a clause B is to look at a short vector of meta-information – *features* – stored along with the clause A and compare it to the corresponding meta-information of B. For example, a longer clause cannot subsume a shorter clause, a deeper clause cannot subsume a shallower clause, etc.

The basic features GKC uses for comparison are ground/non-ground, clause length, size, depth, length of a negative subset, length of a ground subset.

Additionally – what turned out to have a significant effect – we compute and compare *hash features* as the hashes of ground prefixes. A *ground prefix* denotes the part of the sequential atom representation until the first occurrence of a variable. As one of the non-hash features we use the longest ground prefix.

We look at ground hashes of several short lengths (currently 1, 2 and 3) of ground prefixes: length 1 corresponds to (signed) predicate symbols, length 2 to a predicate symbol plus the first the argument, assuming it is ground, etc. Each integer hash of the ground prefix is again hashed to a small integer $0 \ldots 29$ corresponding to a single set bit position in an integer.

Hence, for 30-bit integers as used in the WhiteDb encoding we essentially have 30 different representations for ground prefixes. Finally, the one-bit representations of ground prefix hashes of all literals in a clause are put together by using the bit-wise logical *or*. As a consequence we can use the following observation during clause-to-clause subsumption: a clause A cannot subsume a clause B if the bit-wise representation of ground prefixes of the same length of A are not a bit-wise subset of the same bit-wise representation for B.

We will bring examples of the performance of hash features in the later section.

3.3 Clause Selection Queues

Clause selection is, in our understanding, the most crucial choice point of resolution provers, see [11]. Hence we plan to carry out more research and experimentation with GKC for this direction.

Currently we perform the selection of a given clause by using several queues in order to spread the selection relatively uniformly over different important categories of derived clauses. The queues are organized in two layers.

As a first layer we use the common ratio-based algorithm [11] of alternating between selecting N clauses from a weight-ordered queue and one clause from the FIFO queue with the derivation order. This *pick-given ratio* N is set to 4 by default.

As a second layer we use four separate queues based on the derivation history of a clause. Each queue in the second layer contains the two sub-queues of the first layer.

We note that formulas in TPTP are normally annotated as either being axioms or conjectures/goals to be proved or assumptions/hypothesis posed and relevant for the goal. This annotation can be seen to arise naturally in question answering tasks from a large knowledge base: the latter consists of axioms and the question can be often split into the goal and the assumptions part.

We split all the input and derived clauses into four classes based on their history according to the annotations:

1. Clauses having both the goal and assumption in the derivation history.
2. Clauses having some goal clauses in the derivation history.
3. Clauses having some assumption clauses in the derivation history.
4. Clauses having only axioms in the derivation history.

These four queues are disjoint and if conditions are not mutually exclusive, the higher (earlier) one has priority.

Our initial experiments with different ratios for these queues indicate that giving more preference to the first two seems to be a better strategy for large problems than a uniform approach. Obviously, an optimal ratio is highly dependent on the type of the problem.

All in all, the use of these four queue classes has been highly beneficial for the performance of GKC, both for the set of support strategy and all the other resolution strategies.

4 Term Representation

GKC uses WhiteDB data structures for term representation: each term or clause is represented as a WhiteDB database record containing meta-information followed by term elements encoded as integers. Meta-information for literals in the clause is kept on the clause level to avoid the need to follow a pointer to access the literal meta-information.

WhiteDB database records – and hence also the main data structures in GKC – are tuples of N elements, each element encoded as an integer in the WhiteDB-s data encoding scheme. Since the WhiteDB data structures can be kept in the shared memory where absolute pointers do not work (processes map memory areas to different address spaces), conventional pointers are not used in the main data structures: this role is given to integers indicating offset to the current memory area, thus enabling the data in memory to be independent of its exact location.

Low bits of an integer in a record indicate the type of data stored in high bits. Pointers (offsets) have always zeros in low bits. Predicate and function symbols are represented as URI-s. Small integers, floats, boolean constants and variables fit into one integer directly, while large integers, doubles, strings and URI-s are represented as an offset to a separate data structure. In particular, URI-s are stored uniquely and contain various statistical and cached information in addition to the namespace and main strings.

WhiteDB uses our own implementation of a malloc-like allocator for shared memory and simple continuous allocation from memory pools for terms, literals and clauses. In particular, memory space for subterms and literals in a clause is continuous, which improves cache locality, important for walking through an atom or a term.

5 Performance

We describe the performance of GKC by comparing it to the results from the first order proof search category FOF of the latest CASC competition CASC-J9 held in 2018, see [16]. The detailed logs of all proof attempts as well as a working system for experimentation can be found in the GKC repository release v0.1–alpha [18].

The long-term winner of CASC – since 2002 – is the prover Vampire [10] with a clearly superior performance to all the other competitors. Incidentally, an early prover Gandalf [2] of the current author won an analogous division of CASC in 1997 and 1998. GKC has no direct relations to Gandalf nor shares any code.

The FOF division presents 500 randomly chosen problems from TPTP (see [1] and [7]) to the competitors with the goal to solve as many as possible under the given time limit. The competition was run on a quad-core Intel(R) Xeon(R) E5-2609 chip, with each problem solution attempt having access to all the cores for a maximum of 300 s.

We note that the problems were selected from a wide variety of different problem classes and for the most part are not very large. GKC is not designed for the majority of these classes.

For comparison purposes we ran GKC on a laptop with Ubuntu Linux 16.04 and the Intel(R) Core(TM) i7-5500U chip, using just one core. Shared memory was not used, i.e. each proof attempt started from scratch and included parsing and problem preparation.

A fixed sequence of simple strategies was run inside the limit of 200 s. The core strategies were binary ordered resolution, unit resolution and the set of support resolution. These were combined with none or small static limits (1, 2, and 4) for term depth and either using four beforementioned clause selection queue classes for giving preference to goal- and assumption-derived clauses or only one, common queue class, giving preference to smaller and older clauses regardless of history. For the set of support strategy stronger preference was given to the clauses containing goals and assumptions in their history. For unit and ordered binary resolution we used either no history-based preference at all or equal preference to the queues of clauses derived from goals, assumptions, combination of these or axioms only.

It is worth noting that on average the binary ordered resolution with several equally preferred history-based queues performed better than having no history-based preference. On the other hand, strong preference to queues formed from goal- and assumption-derived clauses performed on average worse than equal preference among the queues. The same cannot be said of the set of support strategy, where stronger goal- and assumption preference of said queues was better on average.

Equality was always handled by paramodulation with the Knuth-Bendix ordering. No demodulation was used. In short, the strategies selected were basic, not specially tuned or dependent upon a problem given.

In this setting GKC showed satisfactory performance, landing in the first half of the result list.

The following table inserts the GKC result into the official list of CASC-J9 results for comparison purposes. GKC did not take part of this competition (Table 1).

Table 1. CASC-J9 FOF results with GKC inserted.

System	Proofs
Vampire 4.3	461
Vampire 4.2	454
CSE_E 1.0	363
E 2.2pre	350
CVC4 1.6pre	298
GKC 0.1	**260**
Leo-III 1.3	256
iProver 2.8	248
leanCoP 2.2	143
nanoCoP 1.1	126
CSE 1.1	123
CSE 1.0	122
Prover9 1109a	122
Twee 2.2	74
Geo-III 2018c	50

Next we will have a look at the performance of hash features for clause-to-clause subsumption described earlier. We will consider all clause-to-clause subsumption attempts of non-ground-unit clauses for the two hardest problems for GKC from the seven largest problems from CASC-J9.

The subsumption pre-filter runs in stages, each stage detecting that subsumption of A by B is impossible due to some features stored as meta-information. As described earlier, the first, top features stage, considers ordinary features like clause length, depth etc. The hash prefix stages introduced in GKC check the bit-wise inclusion of encoded hashes of ground prefixes of length 1, 2 and 3. Only the subsumption attempts passing all these filters will continue to the stage where the subsumption of literals in respective clauses is considered (Table 2).

Table 2. Subsumption prefilter performance example.

Filter stage	CSR056+6	CSR033+6
All subsumptions attempted	360280255	669782763
Passed top features stage	13818803	6288334
Passed hash prefix length 1	772059	437145
Passed hash prefix length 2	448353	78176
Passed hash prefix length 3	435218	54881

5.1 Performance on the Largest Problems in TPTP

The experiments in this section are conducted using a newer release 0.1-epsilon of GKC, having better command-line support for using shared memory databases and being roughly twice faster for large problems than the release 0.1-alpha used in the previously described experiments.

The largest problems in TPTP, CSR025 ... CSR074, ask questions from the axioms built from the OpenCyc database: 50 problems for the axiom set CSR002+5.ax with over three million formulae and 50 problems for the subset of the latter, the axiom set CSR002+4.ax with over half a million formulae. Importantly, the larger set CSR002+5.ax is itself unsatisfiable, while the smaller CSR002+4.ax is satisfiable. We note that CASC-J9 contained seven problems based on the set CSR002+5.ax and only Vampire and iProver could solve any of these problems.

The default strategy of GKC for large formulas is binary resolution with the set-of-support strategy and earliest-derived vs. lightest-clause picking ratio four, with no special limits or preferences.

GKC parses and indexes the CSR002+5.ax set into shared memory in ca 23 s. After that the proof searches are run as new independent command-line commands which do not need to parse or do initial indexing and could be run in parallel. In this setting GKC proves 44 of the problems with the default strategy, most of them under 1 s and the longest time being 31 s. The remaining six of the problems are proved with a few variations of the clause picking ratio and set of support preferences, with the longest time being 3 min.

Since CSR002+5.ax is itself unsatisfiable, the problems based upon it are not well suited for comparison with other provers. Although the GKC strategy appears to use the given question in the derivation, there is no strict obligation to do so.

Next we look at the same 50 problems asked about the smaller, satisfiable axiom set CSR002+4.ax. GKC parses and indexes the CSR002+4.ax set into shared memory in ca 3.7 s. Again, using the shared memory database, GKC proves 45 of the problems with the default strategy and the remaining five hard problems either with a unit strategy limit (one of the clauses resolved upon must be a unit clause) or a derived clause size limit of 2. 36 of the problems are solved under 0.1 s, 12 between 0.1 and 1 s and the slowest two under 3 s.

We have compared GKC performance on the same problems with Vampire 4.2.2 in the "casc" mode. Most of the problems are solved in ca 10 s and most of this time is spent on parsing and initial indexing. The default initial strategy of Vampire solves 41 problems, while the remaining nine hard problems are solved after sequentially trying several strategies. These sequential attempts take ca one minute and in one case two minutes until the proof is found. The five hard problems for GKC are a subset of the nine hard problems for Vampire.

6 Summary and Further Work

We have presented a new automated theorem prover GKC optimized for search in large knowledge bases. While the development of GKC is ongoing, it is already a usable and performant generic theorem prover. The main outstanding practical capabilities of GKC as it stands now are its ability to use prepared knowledge bases in shared memory and top of the line performance for handling large knowledge bases.

From a research perspective we note that our experiments with GKC indicate that it is feasible for a theorem prover to rely purely on the hash indexes and avoid tree indexes altogether. We have introduced hash prefix filters for clause-to-clause subsumption and demonstrated their good performance. This said, we acknowledge that tree indexes do have superior performance in scenarios with deep term structures.

We plan to pursue the following directions for future work:

1. Improving the general-purpose performance of GKC by implementing additional derivation rules, algorithms and strategies. We do plan to participate in the next CASC competition.
2. Implementing specialized methods for large knowledge bases, like precomputation and built-in handling of transitive properties.
3. Improving the functionality of knowledge base preparation and precomputation with methods like vector-based statistical analysis for likely interdependencies.
4. Measuring the performance of hash indexes when compared to tree indexes for different problem classes.

5. Developing an experimental toolset of commonsense reasoning functionalities on top of GKC, with the principal aim to make GKC usable as a component in natural language understanding systems.

References

1. TPTP homepage. http://tptp.cs.miami.edu/~tptp/
2. Tammet, T.: Gandalf. J. Autom. Reason. **18**(2), 199–204 (1997)
3. Tammet, T.: Towards efficient subsumption. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS, vol. 1421, pp. 427–441. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054276
4. Pease, A., Sutcliffe, G.: First order reasoning on a large ontology. In: Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories, vol. 257, pp. 61–70. CEUR Workshop Proceedings (2007)
5. Reagan, S.P., Sutcliffe, G., Goolsbey, K., Kahlert, R.C.: The Cyc TPTP Challenge Problem Set. Unpublished manuscript. http://www.opencyc.org/doc/tptp_challenge_problem_set
6. Suchanek, F., Kasneci, G.m Weikum, G.: YAGO: a core of semantic knowledge. In: Proceedings of the 16th International World Wide Web Conference, Banff, Canada, pp. 697–706
7. Sutcliffe, G.: The TPTP world – infrastructure for automated reasoning. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 1–12. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_1
8. Suda, M., Weidenbach, C., Wischnewski, P.: On the saturation of YAGO. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 441–456. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_38
9. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Handbook of Automated Reasoning, pp. 19–99. Elsevier (2001)
10. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
11. Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 330–345. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_23
12. Sekar, R., Ramakrishnan, I., Voronkov, A.: Term indexing. In: Handbook of Automated Reasoning, vol. II, chap. 26, pp. 1853–1964. Elsevier Science (2001)
13. Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics. LNCS (LNAI), vol. 7788, pp. 45–67. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36675-8_3
14. Tammet, T., Järv, P.: WhiteDB homepage. https://whitedb.org
15. Furbach, U., Schon, C.: Commonsense reasoning meets theorem proving. In: Proceedings of the Workshop on Bridging the Gap between Human and Automated Reasoning co-located with 25th International Joint Conference on Artificial Intelligence IJCAI 2016, pp. 74–85. CEUR (2016)
16. Sutcliffe, G.: The 9th IJCAR automated theorem proving system competition - CASC-J9. AI Commun. **31**(1), 1–13 (2018)

17. Lopez Hernandez, J.C., Korovin, K.: An abstraction-refinement framework for reasoning with large theories. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 663–679. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_43
18. Tammet, T.: Repository of the GKC system and experiment logs (2019). https://github.com/tammet/gkc