CISC-481/681 Project 1: Inference Algorithms Applied to Poisonous Fungus Identification

Dylan Chapp, Jake Moritz, E.J. Murphy
Department of Computer and Information Sciences
University of Delaware - Newark, DE 19711
Email: {dchapp},{jmoritz},{murphyej}@udel.edu

I. Introduction

Due to the wide variety of human ailments and the limited capacity of human medical expertise, artificial intelligence knowledge systems (KS) are increasingly finding use in real diagnostic situations. [1] One particular application of KS to diagnostics is in the identification of poisonous fungi, which we explore in this work.

Some stuff goes here [2] [3] [4]

II. Related Work

III. Knowledge Base

IV. Implementation

We chose to implement the knowledge system as a command-line tool written primarily in Python [3] for two primary reasons. The first is that as Python is a dynamic, interpreted language with a sufficiently powerful standard library. As a consequence, it was possible to get a prototype working within a week of development and from there onward it was possible to rapidly iterate upon and test the design. The second is that Python has a rich ecosystem of open source third-party modules. Among them, the SymPy symbolic logic module [4] was instrumental to our implementation. Our knowledge system makes extensive use of the types provided by SymPy and the module's ability to efficiently convert sentences in propositional logic to conjunctive normal form (CNF).

A. Establishing the Knowledge Base

The knowledge system operates by first ingesting a file containing *rules* which it internally represents as SymPy inferences. Then, it offers two methods for determining a set of *facts*. Either the user can provide a file containing those facts which can be parsed similarly to the rule file, or the user can invoke the interactive mode of the knowledge system which prompts them with questions about the ingested mushroom and the symptoms of the patient.

Once a list of rules and a list of facts are established, the system concatenates them, formats their contents as necessary for agreement with SymPy's API. The list can now be converted into a conjunction of SymPy symbols using the *sympify* function, then converted to CNF using the *to_cnf* function. Once a CNF representation of the knowledge base is generated, the user is prompted to enter a query, which is subsequently formatted and sympy-fied. The result is a pair of inputs that can be operated upon by any of the implemented inference algorithms to determine entailment of the query.

B. Inference Algorithms

We implement three inference algorithms in this knowledge system: resolution, forward chaining, and backward chaining. Our implementations of resolution and forward chaining are based upon the pseudocode in Russell and Norvig [2], while our implementation of backward chaining is based primarily on the notes and slides provided in class.

V. Evaluation

A. Correctness

B. Performance

In order for a knowledge system to be useful in real diagnostic situations, it must deliver correct results quickly. For two of the algorithms considered–forward chaining and backward chaining–there is a suitable performance guarantee. Namely, one can prove that these algorithms run in linear time with respect to the size of the knowledge base, provided that the the knowledge base consists only of i*Horn Clauses*. [2]. Our implementation guarantees this property of the knowledge base by ensuring that the knowledge base is in CNF before it is passed into the inference algorithms. Moreover, linear time is the worst case scenario for backward chaining. In many cases, the backward chaining algorithm will terminate much faster, though this is strongly dependent on rules and facts that exist within the knowledge base.

However, the performance of the resolution algorithm is much worse. (continued discussion goes here)

Run time is not the only performance concern to be addressed. The memory requirements of the inference algorithms also vary which, were this knowledge system to be deployed on embedded device in a clinincal environment, would need to be taken into account. In particular, we provide two versions of the backward chaining algorithm–a recursive implementation and a purely iterative implementation. Both provide correct performance, but the recursive implementation's many function calls can introduces intolerable overhead in a resource constrained environment. During development, we encountered a curious problem while testing the recursive implementation's correctness. The entire correctness test suite would pass on a MacBook Air with 4 GB RAM, but only some of the tests would pass when the code was run in an online IDE in the Chrome web browser. We conjectured that due to the need to potentially host many users simultaneously on fixed hardware, the online IDE imposes strict memory use constraints on any individual user. Discovery of this limitation inspired us to implement an iterative backward chaining algorithm, which is able to pass the entire test suite even in the online IDE environment.

References

[1] D. E. Smith, "Expert systems for medical diagnosis: A study in technology transfer," Ph.D. dissertation, San Diego, CA, USA, 1989, aAI9015675.

[2] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.

[3] `Python Team`, "Python language reference v. 2.7.11," https://docs.python.org/2/reference/.

[4] `SymPy Team`, "Sympy 1.0 documentation," http://docs.sympy.org/latest/index.html.