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.

II. Knowledge Base

The precise identification of a mushroom can be a difficult task requiring careful and complete inspection of the mushroom and in some cases inspection of its spores under a microscope or application of reagents to its tissue. [2] In the context of identifying a poisonous mushroom after ingestion, this complex task is further confounded by the possibility that only an incomplete specimen remains, that an identifying component such as a volva fell off during harvesting, or more simply that the symptoms of mushroom poisoning render self-diagnosis impossible. Nevertheless, even if the exact species–e.g. *Amanita Phalloides* the "Destroying Angel"–of poisonous mushroom cannot be identified, determination of the genus coupled with information about the patient's symptoms can be sufficient to determine the type of poisoning–e.g. *amatoxin*–and recommend a treatment. With these considerations in mind, we collected sets of rules for mushroom identification and for linking sets of symptoms–e.g. jaundice–to syndromes–e.g. liver failure–associated with specific toxins.

A. Mushroom Identification Rules

With the approximately 20,000 documented species of mushrooms found throughout the Earth it is to be expected that some would be toxic to humans. But, of the sixty or so poisonous species that we know to exist, twenty-nine are known to be deadly poisonous. Focusing on these twenty-nine species, we evaluated a number of attributes that could be used to identify one species over another. These attributes are cap shape, cap color, gill color, gill type, cap texture, hymenium type, stem color, stem texture, whether a stem has any rings, whether a stem has a volva, the spore print color, odor, location it was found, season of fruition, host organism, and climate. Knowing all of these features, or in most cases a subset of these features, one can classify a mushroom by genus and species. Once a mushroom has been identified it is easy to determine the types of toxins it harbors and if eaten, what antidotes or treatments to apply. For instance, if we were to know that for a particular mushroom the spore print color is white, the gill type is unattached, the gill color is white, and the stem color is white then we can assume that the mushroom belongs to the genus *Amanita*. By narrowing our search to mushrooms in the *Amanita* genus and to mushrooms found in the Guangdong province of China we can conclude that the mushroom is *Amanita Exitialis*.

B. Symptom-to-Syndrome Rules

In a similar manner, we developed a hierarchy of identifying what specific toxin someone is being afflicted by. By first identifying observable symptoms we can make inferences about what unobservable syndromes the individual is experiencing. From those syndromes we can identify the specific poisoning someone is suffering from. As an example, if an individual is reported to have a resting heart rate of greater than 100 they are experiencing

tachycardia. If the individual has blueish skin and shortness of breath along with tachycardia they could be suffering from respiratory failure. Finally, if someone is suffering from respiratory failure, liver failure, kidney failure, and diarrhea they may have been poisoned by high levels of *Alpha-Amanitin*, a toxin found in lethal amounts in mushrooms of the *Amanita* genus.

III. 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 [5], while our implementation of backward chaining is based primarily on the notes and slides provided in class.

The inference algorithms we implement are able to decide entailment of queries based on the relationship between entailment and boolean satisfiability. Specifically, suppose kb is a knowledge base represented as a propositional logic sentence in CNF, and that q is a query represented as a propositional logic literal. Then $kb \models q$ is equivalent to the unsatisfiability of $kb \wedge \neg q$.

C. Modes of Use

IV. Evaluation

A. Correctness

Despite the fact that proofs of soundness exist for forward chaining, backward chaining, and resolution [5], a proof that we correctly implemented these algorithms does not. To establish confidence in the correctness of our implementations, we use the negation of the result of applying the SymPy `satisfiable` function to the conjunction of the knowledge base and the negation of the query as a benchmark of 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 *Horn Clauses*. [5].

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.

V. Conclusions and Future Work

We were able to implement a knowledge system composed of two major code components: 1). an ingestion engine for aggregating a knowledge base of facts and rules into a propositional logic sentence, and 2). a suite of inference algorithms which take as arguments the knowledge base and a user's query and decide whether or not the query follows. By embedding these components in a command line user interface designed with diagnosis in mind, we provide clinicians with the ability to rapidly and correctly identify cases of mushroom poisoning.

The most current version of the knowledge system code and the `.tex` and `.bib` files necessary to generate this document are available at https://github.com/dchapp/cisc681_kb_project. Though we feel that we have delivered an effective and useful knowledge system, there are architectural elements of the project that warrant future improvement.

Namely, we make only the most basic usage of the SymPy API. Aside from the SymPy types, the `to_cnf` function, and the `sympify` function, all other functionality is hand written. (We use the `satisfiable` function for correctness testing, and as such do not consider it strictly part of the implementation.) In future development, we would seek to replace our hand written routines for–e.g. extracing symbols from a conjunction of clauses–with optimized SymPy functions if they exist.

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] B. H. Rumack and D. G. Spoerke, *Handbook of Mushroom Poisoning: Diagnosis and Treatment*. CRC Press, 1994.

[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.

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