Menagerie and Optimization

1 Background

This section goes through a number of more advanced Prolog features, which tend to be used with less frequency, if at all. Many of these features have strange semantics and limitations, and so they have a tendency to make programs difficult to understand if misapplied. However, these can be crucial for solving certain kinds of problems, particularly if efficiency is a concern. We first introduce a number of these features, and then discuss some general points about optimizations in Prolog.

2 Negation-as-Failure

Prolog is all about inferring what is *true* given a clause database and a query. However, occasionally we are interested in what is *false* given the same information.

To provide an example, let's consider the problem of choosing flights so that we arrive at a given destination. First, let's create a series of facts corresponding to different available direct flights, shown below:

```
1  flight(lax, sea, 1).
2  flight(sea, lax, 2).
3  flight(pdx, sea, 3).
4  flight(sea, pdx, 4).
```

The first fact above (line 1) states that there is a flight from `lax` to `sea` (using airport codes), and that flight number is `1`. The rest of the facts follow the same format.

Given information about direct flights, we can derive whether or not it is possible to reach a given destination, perhaps using multiple flights. That is, while a direct flight might not be possible, it may be possible to use a series of flights to reach a given destination. This can be succinctly encoded using two rules, shown below:

```
1  flightPath(A, B, [Num]) :-
2      flight(A, B, Num).
3  flightPath(A, B, [FirstNum|Rest]) :-
4      flight(A, C, FirstNum),
5      flightPath(C, B, Rest).
```

The `flightPath` procedure above is intended to derive a path from the first parameter to the second parameter, putting the path in the third parameter. The path is an ordered list of flight numbers, and taking the flights in the given order will allow you to reach the desired destination from the desired source.

At a cursory glance, the above code works, as shown through the queries below:

```
?- flightPath(lax, sea, Path).
Path = [1].
?- flightPath(lax, pdx, Path).
Path = [1, 4].
```

However, both of the above queries have multiple solutions, and moreover and infinite number of solutions. To see this, the first three answers of both of the above queries are shown below:

```
?- flightPath(lax, sea, Path).
Path = [1] ;
Path = [1, 2, 1] ;
Path = [1, 2, 1, 2, 1] ;
...
```

```
?- flightPath(lax, pdx, Path).
Path = [1, 4] ;
Path = [1, 2, 1, 4] ;
Path = [1, 2, 1, 2, 1, 4] ;
...
```

As shown, instead of giving us the most direct route, the above paths contain cycles. From the perspective of traveling, the alternative solutions above make no sense, as these alternative paths tell us to go in circles.

There is a second, deeper problem in the above code: this can end up looping forever if no path exists. To illustrate, let's add two more flights between two imaginary destinations `foo` and `bar`:

```
flight(foo, bar, 5).
flight(bar, foo, 6).
```

If we then ask if it's possible to travel from `foo` to `sea`...

```
?- flightPath(foo, sea, Path).
ERROR: Out of global stack
```

...the above code appears to loop forever, and eventually runs out of memory!

Both of these issues are rooted in the same fundamental problem: nothing in this problem description disallows cycles. That is, we are free to keep visiting the same airport repeatedly during one trip. With the first set of queries, this leads to us going back and forth between the same airports. With the most immediate query above, this leads to an infinite loop, as the code above indefinitely goes between `foo` and `bar` without ever making any real progress.

Perhaps the most straightforward way of disallowing cycles is to keep track of the airports visited, and to disallow traveling to the same airport twice. In English, it's easy enough to phrase this as "I can only take a flight to a given destination if I have **not** already traveled to that destination". The use of "not" in the sentence above requires us to introduce a new Prolog feature: *negation-as-failure*. Negation-as-failure causes a query to fail if it succeeds, and succeed if it fails. This is similar to the sort of negation you're more used to from Boolean logic.

With negation-as-failure in hand, we reimplement `flightPath` below with a new procedure named `uniqueFlightPath`. The first, second, and fourth parameters to `uniqueFlightPath` behave like the first, second, and third parameters of `flightPath`, respectively. The new third parameter keeps track of the airports visited. The Prolog code is shown below, where `\+` is the negation-as-failure operator:

```
1  uniqueFlightPath(A, B, Traveled, [Num]) :-
2      flight(A, B, Num),
3      \+ member(B, Traveled).
4  uniqueFlightPath(A, B, Traveled, [Num|Rest]) :-
5      flight(A, C, Num),
6      \+ member(C, Traveled),
7      uniqueFlightPath(C, B, [C|Traveled], Rest).
8
9  % for the initial call to uniqueFlightPath
10 % states that we have already traveled to
11 % the source airport
12 uniqueFlightPath(A, B, Path) :-
13     uniqueFlightPath(A, B, [A], Path).
```

There are multiple key differences in `uniqueFlightPath` when compared to the original `flightPath`:

- The third parameter (`Traveled`) is a list of airports which we have traveled to. As a refresher, `member(A, Traveled)` will succeed only if `A` is a member of the list `Traveled`.
- (Line 2) If there is a flight between `A` and `B`, take the flight, but only if we haven't already visited `B` (line 3).
- (Line 5) We can take a flight between `A` and `C`, but only if we haven't already traveled to `C` (Line 6).
- (Line 7) Record that we have traveled to `C` before making the recursive call.

Running the above queries again with `uniqueFlightPath` now gives us answers we'd expect:

```
?- uniqueFlightPath(lax, sea, Path).
Path = [1]. % only solution
?- uniqueFlightPath(lax, pdx, Path).
Path = [1, 4]. % only solution
?- uniqueFlightPath(foo, sea, Path).
false. % answer derived almost instantly
```

As shown, negation-as-failure allows us to represent this acyclic property properly.

2.1 Limitations

Negation-as-failure works exactly as described: it runs the given query, and succeeds if and only if the query fails. It is not immediately obvious from this description, but this means that negation-as-failure cannot be used to construct or unify any values. For example, consider the following query:

```
?- \+ member(A, [1, 2, 3]).
false.
```

The above query returns `false`, which may seem strange. One might want to interpret the above query as "select an `A` such that `A` is not a member of the list `[1, 2, 3]`". However, this is not how the query gets executed. Instead, the Prolog engine first executes the non-negated query:

```
?- member(A, [1, 2, 3]).
A = 1 ;
...
```

...which has at least one answer, as shown. At this point, because the non-negated query succeeded, this causes the original negated query to fail.

With this in mind, we often need to be careful when negation-as-failure is issued, because it tends to work in a decidedly non-logical way. For example, consider the following two related queries:

```
?- A = 1, \+ A = 2.
A = 1.
?- \+ A = 2, A = 1.
false.
```

The first query succeeds, since by the time `\+ A = 2` is reached, `A` alredy has the value `1`, which does not unify with `2`. However, with the second query, since the uninstantiated variable `A` *can* unify with `2`, the first portion `\+ A = 2` fails, even though later we do unify `A` with a non-`2` value (namely `1`). This is a reflection of the fact that in Prolog, conjunction (`,`) works strictly from left-to-right. This defies typical logic, since from a purely logical standpoint the ordering of the parameters in the conjunction shouldn't matter.

Lastly, negation-as-failure will never instantiate any logical variables to any values. This is because negation-as-failure succeeds only if internally the given query fails. Upon query failure, no variables are ever instantiated with values, *even if* it might be possible to do so. For example, consider the following query:

```
?- A = 1, 2 = 3.
false.
```

The above query fails, and does not instantiate `A` with any values, *even though* the `A = 1` portion succeeded. This is reflected if we use negation-as-failure on the overall query, like so:

```
?- \+ (A = 1, 2 = 3).
true.
?- \+ (A = 1, 2 = 3), A = 2.
A = 2.
```

As shown in the first query, the engine does not output any value for `A`, implying that `A` was never given any specific value. The second query further demonstrates that `A` wasn't given a value in the negation, as we are able to unify it with an incompatible value (namely `2`) after the negation-as-failure succeeds.

3 Determining if a Variable has a Value

Occasionally, it is useful to know whether or not a variable has a value at a given point in time. For this purpose, there exist three related built-in procedures:

1. `var`
2. `nonvar`
3. `ground`

The `var` procedure succeeds if its parameter is uninstantiated, and fails otherwise. For example, consider the following queries:

```
?- var(X).
true.
?- X = 1, var(X).
false.
?- var(X), X = 1.
X = 1.
```

As shown, `var` succeeds if the given variable does not have a known value. With this in mind, the ordering can matter with conjunctions; the second query fails because `X` has a known value by the time `var(X)` is reached, whereas the third query succeeds because `X` is only given a value *after* the `var(X)` call.

The `nonvar` procedure works like `var`, except the result is inverted. That is, `nonvar` succeeds if the given variable has a known value, and fails if it doesn't have a known value. For example, consider the following queries:

```
?- nonvar(X).
false.
?- X = 1, nonvar(X).
X = 1.
?- nonvar(X), X = 1.
false.
```

Note that `nonvar` only works one level deep. For example, consider the following query:

```
?- X = foo(Y), nonvar(X).
X = foo(Y).
```

As shown above, `nonvar` succeeds on `foo(Y)`, even though `Y` does not have a known value. This is because `X` holds `foo(X)`, which has a *partially instantiated* value (that is, a structure with known and unknown parts).

If you want to explicitly check that the given variable is fully instantiated (that is, it contains no unknown values), you can use the `ground` procedure, like so:

```
?- ground(foo(X)).
false.
?- ground(foo(1)).
true.
?- X = 1, ground(foo(X)).
X = 1.
```

As shown above, the last two queries succeed because the parameter to `ground` is fully instantiated/known, whereas the first query fails because `X` is not instantiated to any known values.

3.1 Practical Uses

The aforementioned procedures can come up in places where we want different behavior, depending on whether or not some input has a known value. One such example is the built-in `length` procedure, which is intended to determine the length of a list. The built-in `length` procedure can work with any combination of known and unknown parameters, as shown with the queries below:

```
?- length([foo, bar, baz], Len).
Len = 3.
?- length(List, 2).
List = [_G1581, _G1584]. % different identifiers are possible
?- length(List, Len).
List = [],
Len = 0 ;
List = [_G1593] ;
Len = 1 ;
List = [_G1593, _G1596] ;
% infinitely more answers omitted
```

The fact that `length` can work in this way should surprise you, since the version we implemented in class cannot properly handle the second query. This version is shown below for convenience:

```
1  myLength([], 0).
2  myLength([_|T], Len) :-
3      myLength(T, TLen),
4      Len is TLen + 1.
```

The above implementation will correctly produce the correct answer for the second query, but will appear to have more answers. Asking for more answers will lead to an infinite loop (well, infinite recursion). We can fix this with some strategic uses of `var` and `nonvar`, shown below:

```
1  newMyLength(List, Len) :-
2      var(List),
3      nonvar(Len),
4      buildList(List, Len).
5  newMyLength(List, Len) :-
6      \+ (var(List), nonvar(Len)),
7      myLength(List, Len).
8
9  buildList([], 0).
10 buildList([_|T], Len) :-
11     Len > 0,
12     NewLen is Len - 1,
13     buildList(T, NewLen).
```

Note that `newMyLength` calls the original `myLength` procedure above. As shown, the `newMyLength` procedure will handle the case where the `List` is unknown but `Len` (short for length) is known differently. Specifically, in this case it will instead call the `buildList` procedure, which builds a list given a list length. With this variant in place, running the following query:

```
?- length(List, 2).
List = [_G1581, _G1584]. % different identifiers are possible
```

...will produce only the above answer. While more answers seem possible, it immediately fails if we ask for more, fixing the original infinite loop problem.

From a more abstract perspective, `var`, `nonvar`, and `ground` often come up when doing heavy optimizations, particularly in situations where we want to build up a given value incrementally. While we are building a value, we may need to check a lot of different things to ensure the value's validity. However, once a value is constructed, we don't need to keep re-checking validity. Concretely, this sort of scenario happens with Sudoku when dealing with input numbers which were originally present on the board. Assuming there is a solution to the problem, any constraints on these numbers can be skipped over, since we know they should be valid given the assumption that there is a solution. This means that potentially computationally expensive checks on the validity of such numbers (i.e., is the number unique within its row, column, and square?) can be safely skipped.

3.2 Caveats

While `var`, `nonvar`, and `ground` are incredibly powerful, with great power comes great responsibility. As with negation-as-failure, the aforementioned procedures break the purely logical view of programs. This leads to code that can be difficult

to reason about. Ideally, code should work consistently no matter what the query is, but the above procedures quite intentionally allow us to treat different queries differently. These should be used sparingly, and only after other options have been exhausted.

4 Implication

Occasionally we need to write a procedure that operates over mutually exclusive rules. A toy example illustrating this problem is the recurrence relation for fibonacci numbers, shown below:

$$\texttt{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \texttt{fib}(n-1) + \texttt{fib}(n-2) & \text{otherwise} \end{cases}$$

The Prolog-based implementation of the above relation we derived in class was as follows:

```
1  fib(0, 0).
2  fib(1, 1).
3  fib(N, Result) :-
4      N > 1,
5      MinOne is N - 1,
6      MinTwo is N - 2,
7      fib(MinOne, R1),
8      fib(MinTwo, R2),
9      Result is R1 + R2.
```

Of particular interest is line 4, which was necessary to ensure mutual exclusion between these three cases. With fibonacchi, this check is very cheap to make.

However, consider the `foo` procedure below, where `someExpensiveCheck` performs a computationally-expensive check to see if some condition is true for a given input:

```
foo(Input) :-
    someExpensiveCheck(Input),
    doOneThing(Input).
foo(Input) :-
    \+ someExpensiveCheck(Input),
    doAnotherThing(Input).
```

As shown, in one world `doOneThing` is executed, but only if `someExpensiveCheck` succeeds. In another world, `doAnotherThing` is executed, but only if `someExpensiveCheck` fails. This ensures mutual exclusion between the two cases, depending on the result of `someExpensiveCheck`.

However, the above code is wasteful, because `someExpensiveCheck` is executed in both worlds, doubling the amount of work that is needed to be performed. This can be overcome with the *implication* operator in Prolog, which uses the symbol `->` followed by `;`. An example refactoring the above code to use implication is shown below:

```
foo(Input) :-
    someExpensiveCheck(Input) ->
        doOneThing(Input);
        doAnotherThing(Input).
```

Semantically, implication will check the given condition first (the part before `->`). If this condition is true, it will execute the code afterwards (the code immediately following `->` but before the `;`), namely `doOneThing(Input)` in this case. Contrarily, if this condition is false, it will execute the code after the semicolon (`;`), namely `doAnotherThing(Input)` in this case. Overall, implication operates much like the usual `if...else` in most programming languages.

4.1 Caveats

The condition in the implication is only ever executed once. If the condition is true in more than one way, then only the first answer from the condition will count. To illustrate this behavior, consider the following code:

```
foo(1).
foo(2).
foo(3).
implicationTest(X) :-
    foo(X) -> true; true.
```

The build-in `true` procedure always trivially succeeds, and is effectively a no-op. Let's take the above code and write a basic query:

```
?- implicationTest(X).
X = 1.
```

As shown above, `implicationTest(X)` succeeds with a single answer, namely that `X = 1`. However, there are no other answers, even though `foo(X)` should have three answers (namely for values `1` through `3`). This is because `foo(X)` is used as the condition for implication, eliminating all answers to `foo(X)` except for the first one returned.

If you want to get all the possible answers from the condition, you should instead use *soft cut*, represented with the `*->` and `;` symbols. The code below rewrites the `implicationTest` procedure to use soft cut instead of implication:

```
softCutTest(X) :-
    foo(X) *-> true; true.
```

Running the same query as before will derive all possible values for `foo(X)`, as shown below:

```
?- softCutTest(X).
X = 1 ;
X = 2 ;
X = 3 .
```

5 Cut (!)

In Prolog, the *cut* (`!`) operator is used to "cut" out prior choice points. For example, consider again the `foo` procedure, duplicated below for convenience:

```
foo(1).
foo(2).
foo(3).
```

Running this procedure with a variable results in three answers:

```
?- foo(X).
X = 1 ;
X = 2 ;
X = 3 .
```

Now let's implement the `bar` procedure, which works just like the aforementioned `foo` procedure but it uses cut in one of the rules:

```
1 bar(1).
2 bar(2) :- !.
3 bar(3).
```

As shown below, a query to the `bar` procedure works differently than a query to the `foo` procedure:

```
?- bar(X).
X = 1 ;
X = 2 .
```

Specifically, with the query above, there is no answer `X = 3`. This is because of the cut (`!`) in line 2 of the `bar` procedure. When execution hits the cut, this tells the Prolog engine to discard the most recent choice point. In this case, this dicards the third fact of the `bar` procedure, eliminating `X = 3` as a possible choice.

Cut can be used in optimization contexts, particularly when we are only interested in pursing answers up until a point. Cut can be inserted at such key points, discarding alternative answers.

5.1 Caveats

As with many of the other operators shown, cut behaves in a way that defies usual logic. In particular, can lead to inconsistent results between different, but related, queries. For example, consider again the `bar` procedure above. Currently, even with the cut, the query `bar(3)` succeeds:

```
?- bar(3).
true.
```

This should strike you as strange, because when `bar` was used as a generator, `X = 3` was not a valid answer. As such, because of this use of cut, `bar` now operates differently depending on the particular query given.

Cut can also make programs very sensitive to the ways they are written. For example, consider the following alternative implementation of `bar`:

```
1 bar(X) :- X = 1.
2 bar(X) :- !, X = 2.
3 bar(X) :- X = 3.
```

In contrast to the code before, this alternative version of `bar` does not succeed on `bar(3)`:

```
?- bar(3).
false.
```

This failure is explained by the fact that the cut now occurs *before* the input of `3` is unified with `2`. As such, this cut eliminates the last case which would have succeeded for an input of `3`. If we had instead moved the cut *after* the unification, as with `X = 2, !.`, then this would have behaved the same. As such, program behavior is highly sensitive to where cut is placed.

Cut can also mess with program optimizations, because it's difficult for Prolog engines to reason about as well as humans. As such, programs that use cut may operate more slowly than their cousins which do not use cut.

From an intuitive standpoint, cut is also particularly weird. Cut operates *retroactively*: it can only eliminate choice points which were already created.

Overall, while cut gives you a lot of power, it should be avoided. Excessive use of cut is not considered idiomatic, and it can make programs fragile and difficult to understand. If there is a way to refactor code to avoid cut, then usually this refactoring is the right way to go.

6 Metaprogramming

One last feature of note is that of metaprogramming. While metaprogramming usually runs contrary to optimizations (code with metaprogramming often runs slower, even significantly slower), it is a commonly employed feature for abstracting over computations. With this in mind, judicious use of metaprogramming can help shorten code and avoid code duplication.

The "meta" in "metaprogramming" refers to the fact that we write programs which effectively write (usually small) programs. This is more commonly seen in dynamic languages like Python and Ruby with the built-in `eval` function, which allows evaluating a string as executable code. For example, in Python, we can do:

```
eval(''1 + 2'')
```

...which would give back `3` as a result, even though `''1 + 2''` is a string. Considering that strings can be constructed dynamically, this means that we can generate and dynamically evalaute *potentially any* program, which is a very powerful capability.

Prolog allows us to perform a very similar operation, though instead of `eval` it is called `call`. The built-in `call` procedure takes an atom or a structure and evaluates it as code. For example, consider the following query:

```
?- X = length([foo, bar], Len), call(X).
X = length([foo, bar], 2),
Len = 2.
```

As shown, while the variable `X` holds a structure, this structure is treated as code with `call(X)`. As code, this structure represents a call to the built-in `length` procedure, with parameters `[foo, bar]` and `Len`. This then finds the length of the given list, as per the usual definition of the `length` procedure.

The `call` procedure can also accept additional arguments to pass along to the procedure, which will be provided in order. For example, the above query could be rewritten as follows:

```
?- X = length([foo, bar]), call(X, Len).
X = length([foo, bar]),
Len = 2.
```

The actual call made by `call` above is no different than the call previously made; it still ends up calling `length([foo, bar], Len)`. However, instead of `Len` being hardcoded into the definition of `X`, it is now passed along as a separate parameter. If we think of the variable `X` as holding a parameterized computation, this allows us to pass parameters to that parameterized computation, in this case `Len`. This is far more flexible.

> ***Sidenote:*** If you're familiar with higher-order functions/anonymous functions/closures, the above unification of `X` to `length`, along with the subsequent use of `call`, probably looks a lot like creating a higher-order function and then calling it. In fact, `call` is a much more powerful operation, as we can construct arbitrary structures corresponding to wholly arbitrary programs. In contrast, higher-order functions effectively hardcode the computation they perform; we must specify the program held in a higher-order function ahead of time. However, `call` usually is only used in a manner consistent with higher-order functions; it is rare to actually create an arbitrary program.

6.1 Practical Uses

With judicious use of `call`, we can abstract over computations. For example, consider the problem of iterating over a list, which is a very common part of a lot of operations. We can extract out the actual iteration part into a separate procedure that uses metaprogramming, and then perform different iterations that do different things depending on the computation provided. An example which does this is shown below:

```
1 iterateOver([], _).
2 iterateOver([H|T], Procedure) :-
3     call(Procedure, H),
4     iterateOver(T, Procedure).
5
6 printAll(List) :-
7     iterateOver(List, writeln).
8 allEqual(List, This) :-
9     iterateOver(List, =(This)).
```

...along with some corresponding queries:

```
?- printAll([1, 2, 3]).
1
2
3
true.

?- allEqual([A, B, C], 5).
A = B, B = C, C = 5.
```

With the above query involving `printAll`, the built-in `writeln` procedure (which writes out its input, followed by a newline) ends up being called on every element of the list, in order. That is, `printAll` issues the following dynamically-generated calls, in order:

1. `writeln(1)`
2. `writeln(2)`
3. `writeln(3)`

The `allEqual` procedure is a bit more complex. It will end up issuing calls to unification (`=`) for each element of the list, unifying each element with whatever `This` is. With this in mind, the example query above involving `allEqual` ends up issuing the following calls, in order:

1. `5 = A` (i.e., `=(5, A)`)
2. `5 = B` (i.e., `=(5, B)`)
3. `5 = C` (i.e., `=(5, C)`)

6.2 Caveats

Since `call` involves dynamic code generation, it can be very difficult to reason about. This is especially true given that Prolog lacks proper types, potentially leading to programs which are fragile and difficult to read (notice a theme?). However, unlike the other operators, `call` is generally considered idiomatic, and is occassionally even encouraged. This is likely because `call` is the only effective means Prolog can abstract over computation, at least without a lot of additional custom setup.

7 Tips on Optimizing Prolog Code

On the surface, Prolog may not seem amenable to any sort of optimization, given its "declarative" nature. However, I put declarative in quotes here for an important reason: once we understand how the engine arrives at a solution, it is entirely predictable. As such, we can exploit this understanding to influence *how* the engine arrives at a solution, potentially leading to more efficient programs if we perform the right manipulations. Since we're starting to talk about how computation is performed, this no longer feels very declarative, but we haven't changed anything regarding how Prolog works. Ideally, we like to think in a purely declarative style as much as possible, ignoring how Prolog actually arrives at a solution. However, sometimes we have to take a more direct approach.

Based on my experiences, the most important optimizations involve changing the order in which conjunctions are performed. In the simplest case, this is just changing `A, B` to `B, A`. However, this often requires a major code restructure. For example, consider the problem of Sudoku, which constrains that values between 1-9 must be present in multiple locations, without duplicates. Specifically, Sudoku enforces these constraints for rows, columns, and 3x3 squares on a grid. A very simple, but correct, solution looks something like the following:

```
isSolution(Grid) :-
    allRowsOk(Grid),
    allColumnsOk(Grid),
    allSquaresOk(Grid).
```

However, the above solution is unlikely to scale to Sudoku boards with more than a few unknown values, which is unrealistically simple even for "easy" Sudoku problems. To understand why, consider the following unsolved Sudoku board:

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

When `allRowsOk` is executed on this board, it will start filling in the unknown spaces with appropriate values. Specifically, it will fill in values so that the row constraint, and only the row constraint, is satisfied.

After filling in the first row, the board may look like the following, where filled-in numbers are shown in *blue*:

3	2	*4*	5	*6*	1	*7*	9	*8*
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

As shown, while only the first row has been filled in, this is already an invalid solution in three distinct ways:

1. The value *6* is duplicated in its column.
2. The value *6* is duplicated in its 3x3 square.
3. The value *7* is duplicated in its column.

Despite the above problems, the `allRowsOk` constraint will *continue filling in from this point*, until the unknowns in the board are filled in. While each row will be ok, clearly columns and 3x3 squares will not necessarily (and almost certainly) not be ok. At this point failure will occur, caused by the corresponding checks `allColumnsOk` and `allSquaresOk`. However, it will take a **very long time** for this failure to ultimately propagate back to the aforementioned point. This is because Prolog explores solutions in a depth-first fashion, and will always explore the most recently-discovered alternative world upon failure. As such, Prolog will end up exploring all alternatives for every possible unknown after the first row before it returns to the first row. The number of alternatives grows exponentially with each unknown, and is potentially as large as 9^{46} in this case (there are 46 unknowns after the first row, each with 9 possibilities). Worse yet, given the depth-first nature of the search, the first value which will be swapped out is the *8*, which was not involved in any of the three aforementioned problems. As such, this will spark *another* fruitless search over the 9^{46}-large state space, and we will keep doing such fruitless searches until we swap out the actual problematic numbers.

Merely reordering the above constraints will not help. For example, if we check columns first, then columns will be ok, but rows and 3x3 squares will illustrate the same problem.

A better solution to this problem is to integrate the row, column, and 3x3 square checks into a single, unified check. Here the idea is that whenever a number is placed, relevant constraints on the row, column, and 3x3 square are all checked immediately, instead of being delayed until the board is completely full. While this won't completely solve the problem (we'll still need search, and some searches are destined to fail), this dramatically cuts down on the amount of search that is doomed to fail. Based on my experiences, this can be the difference between solving a Sudoku board in seconds, and effectively never solving it (even with weeks of CPU time).

In general, the most valuable optimizations are those that reduce the amount of fruitless search which must be performed, as shown above. The usual rule is that your programs should *fail fast*: as soon as you can discover that further search is pointless, failure should be triggered. The longer the distance between making a bad choice and discovering that a bad choice was made, the slower the program runs. Moreover, this distance usually gets exponentially long, as it does with Sudoku.

Other optimizations are possible, and there is a healthy amount of information out there about exploiting things like clause indexing, avoiding needless memory allocations, and the like. Such optimizations can help, though I don't get into them because they generally don't help anywhere near as much as avoiding fruitless search. My personal opinion is that if you're in a position where fruitless search is impossible, or better yet there is no search at all, then Prolog probably isn't the right language for the problem. Prolog can be used as a general-purpose programming language, but the key language features are built around search and problems that need search.