Unsolvable Problems Do Not Exist
Only the Absence of Input Prevents Calculation

Bradley G. Homer
Florida State University
bgh9320@my.fsu.edu

Abstract

What is input? In a Turing machine context, input is a series of one or more concrete symbols upon which the machine acts. In this context, input is always defined. Output, on the other hand, is generally accepted to either be defined or undefined. If the output is undefined for a machine and input pair, the problem is regarded as not Turing-computable. This paper explores the hypothesis that for the non-computable problems, it is the input which is undefined, which in turn leads to an undefined output. Then, the nature of input and decisions based on that input is examined in a machine learning context; in particular, how some artificial intelligence algorithms avoid undefined input by disallowing certain decision tree models, yet theoretically retain the ability to address any true problem. The set of all Turing-decidable algorithms is complete; algorithms which are not Turing-decidable (paradoxes) are actually problems for which some of the input is simply undefined.

Introduction

What is input? Is it merely what is on the right-hand side of an assignment? Or is there an implicit requirement for something of substance to actually be provided? In mathematics, input is often represented as a special case, as a parameter to a function, such as x in the ubiquitous f(x). The equal symbol ('=') is generally accepted as the equality operator, which merely means what is on the left-hand side equals what is on the right-hand side. However, this operator often gets overloaded. In fact, it must be overloaded to cover assignment of variables, such as what happens in the body of our f(x). This has led to the use of more explicit conventions such as ':=' for assignment, '=' for equals or '=' versus '==' in programming languages like C++, or the use of additional keywords, such as 'Let', which explicitly precedes assignment. In an assignment, the right-hand side is input, which is being assigned into the variable on the left-hand side. However, input is also required for statements which are concerned with either stating or testing the equality of what is on the right and left of the equal sign. For example, in an ordinary mathematics equality statement in the form of 'x = y', x and y are atoms which are input to the 'equates' function. A function without input is no more than an idea at rest. For example, observe the function:

$$f(x) = x + 1$$

If input is never supplied, this function is of little more value than an idea on paper. It is not until a value for x is actually supplied that we can expect to get a return value.

Whether input has been supplied or not in a given context is precisely what this paper is all about. We will show how input is of central importance in determining whether a problem is solvable or not. The solvability of problems is not a new topic. It is a broad subject which could easily outstrip the scope of this brief paper. Therefore, we will only be doing a cursory review of some of the most important past work concerning the solvability of problems. We will briefly describe Hilbert's use of meta-mathematics and his challenge, Gödel's incompleteness theorems, Turing's halting problem, and the Church-Turing thesis. We will focus on the parts of these works which will give additional insight to the questions and statements regarding input raised within this introduction.

Following our brief review of past work, we will explore the true meaning and importance of input, especially from an information sciences perspective. We will see how core, simplistic mathematics may ironically be hiding some necessary complexity when it comes to input. Therefore, we will then be diving into more detail about input in the context of algorithms. By focusing on a concise derivative of Turing's Halting Problem proof as an example, we will get to the core of where it goes awry (missing input), and

how this missing input or absence of input is intertwined with the relatively modern concept of the deadlock. Then, in relation to deadlock, we look at the field of artificial intelligence; particularly how and why machine learning models avoid deadlock altogether. Finally, we will look at a practical example of a subset halt() program, written in C++, which demonstrates the programmatic detection of the absence of input.

Background

David Hilbert (January 23, 1862 – February 14, 1943) was a mathematician who presented a list of 23 unsolved problems in 1900, a portion of which remain unsolved today and continue to challenge mathematicians. In 1920, he proposed a metamathematics research project, known as Hilbert's program, where he sought to formulate a complete and solid foundation for mathematics where all mathematics stems from a finite set of axioms, and that this system would be provably consistent through some means such as epsilon calculus. Metamathematics is intended to strip away all intrinsic meaning from the various symbols in mathematics, allowing rigorous rules of structure and logic to dictate relations between statements containing such symbols. This was sought to be done such that statements could be consistently deduced as all following from a finite list of originating axioms. The result would be a complete axiomatic foundation of mathematics, where no contradictory statements could be derived without violating some part of the system (Nagel, Newman, and Hofstadter, 2001). One of Hilbert's 23 unsolved problems is the *Entscheidungsproblem* (decision problem), which asks for an algorithm which takes as input a proposition in the form of a statement of first-order (or finitely greater) logic and answers whether the statement is universally valid or not.

Kurt Gödel's (April 28, 1906 – January 14, 1978) incompleteness theorems in 1931, using an axiomatic system, demonstrated that no such system could prove all true propositions within the system. He did this by constructing a statement within the system which asserts itself as unprovable; if provable this would be false, contradicting the concept of provable statements always being true in a consistent system. However, according to Gray (2000), regarding Gödel's theorem in reference to Hilbert's claim that there are no unsolvable problems in mathematics, Gödel asserted ", I wish to note expressly that [this theorem does] not contradict Hilbert's formalistic viewpoint." As a side note of interest, the demonstrated axiomatic system uses the equal symbol ('=') and other operators, but assignment and equality are not explicitly differentiated from one another in the system's axioms. Likewise, it uses an operator for 'if…then…' structures, and input for the first part of this structure is implied.

Alan Turing (June 23, 1912 – June 7, 1954), is widely regarded as the father of computer science and artificial intelligence. With his Turing machine, he formalized what an algorithm is in a mechanistic way. In a Turing machine context, input is a series of one or more concrete symbols upon which the machine acts. In this context, input is always defined. Output, on the other hand, is generally accepted to either be defined or undefined. If the output is undefined for a machine and input pair, the problem is regarded as not Turing-decidable. In 1936, Turing delivered his paper "On Computable Numbers, with an Application to the *Entscheidungsproblem*". In it, he substituted theoretical implementations of Turing machines in the place of Gödel's metamathematical numbering system, and proved using similar logic that the halting problem for Turing machines is undecidable. The halting problem proposed by Turing has to do with being able to decide algorithmically regarding a given Turing machine *and its input* – will it halt or will it run forever? In this paper, we will be focusing in on Turing's halting problem. Rosen (1999) provides us with an excellent, concise summary of Turing's proof for the halting problem:

> Proof: Assume there is a solution to the halting problem, a procedure called H(P,I). The procedure H(P,I) takes two inputs, one a program P and the other I, an input to the program P. H(P,I) generates the string "halt" as output if H determines that P stops when given I as input. Otherwise, H(P,I) generates the string "loops forever" as output. We will now derive a contradiction.
>
> When a procedure is coded, it is expressed as a string of characters; this string can be interpreted as a sequence of bits. This means that a program itself can be used as data. Therefore a program can be thought of as input to another program, or even itself. Hence, H can take a program P as both of its inputs, which are a program and input to this program. H should be able to determine if P will halt when it is given a copy of itself as input.
>
> To show that no procedure H exists which solves the halting problem, we construct a simple procedure K(P), which works as follows, making use of the output H(P,P). If the output of H(P,P) is "loops forever," which means that P loops forever when given a copy of itself as input, then K(P) halts. If the output of H(P,P) is "halt," which means that P halts when given a copy of itself as input, then K(P) loops forever. That is, K(P) does the opposite of what the output of H(P,P) specifies.
>
> Now suppose we provide K as input to K. We note that if the output of H(K,K) is "loops forever," then by the definition of K we see that K(K) halts. Otherwise, if the output of H(K,K) is "halt," then by the definition of K we see that K(K) loops forever, in

violation of what H tells us. In both cases, we have a contradiction.

Thus, H cannot always give the correct answers. Consequently, there is no procedure that solves the halting problem.

Input I, as described by Rosen in his summary above, is of interest to our subject.

Alonzo Church (June 14, 1903 – August 11, 1995) rounds out our list of notable people in our cursory review as one of the mathematicians who linked together the Turing machine and other methods of defining functions together as being computationally equivalent processes. Known as the Church-Turing thesis, it asserts that if some algorithm exists to carry out a calculation, then that same calculation can be carried out by a Turing machine (or other equivalent algorithmic process.) As part of this, Church had independently determined that there is no computable function which decides whether two given λ calculus expressions are equivalent or not, which agreed with Turing's halting problem. (Davis, 1965).

So-called unsolvable problems continue to be a source of interest and angst to this day. However, the field of artificial intelligence gives us some interesting insights. Planning graphs are special data structures which allow us to derive reasonably accurate heuristic estimates about how we can reach some goal state from some starting state. Mutual exclusion (or mutex) links record conflicts in a structure which are not possible. Russell and Norvig (2010) give us an example in two actions which are mutex: *Have(Cake)* and *Eat(Cake)*. Any of three conditions establishes a mutex relationship between two actions in a planning graph:

1. *Inconsistent effects:* one action negates the effect of the other.
2. *Interference:* one of the effects of one action is the negation of a precondition of the other.
3. *Competing needs:* one of the preconditions of one action is mutually exclusive with a precondition of the other.

The way a planning graph deals with such impossibilities is to record them as what they are: impossible, mutually exclusive choices, with no way to reach the desired end goal via such links. An alternative way of describing this is to say that actions on literals such as *Have(Cake)* and *Eat(Cake)* are input which comprise the nodes of a planning graph, and invalid relationships are marked as impossible, and do not lead to the desired output (a path to the goal.) In other words, a mutex relationship is invalid input.

What Is Input?

Understanding what input is and the full nature of its presence and the consequences of its absence for algorithms is essential for an analysis of what can and cannot be calculated. A function with no supplied input, whether internal, external, subtle or explicit, cannot return – it cannot be calculated. We will show that for the non-computable, undecidable problems, it is the input which is at best undefined or completely absent, which in turn leads to an undefined output.

In discrete mathematics, input to a function is often represented in a format similar to the following:

```
y = f(x)
```

Where x is the input to function f(x) and y is the output. Demonstrating actual input and output for an example function such as:

```
f(x) = x + 1
```

is trivial:

```
f(3) = 3 + 1 = 4
```

Where 3 is the input and 4 is the output. Translating our simple function example into one which is more programming code-like, we have:

```
int f( int x )
{
  x = x +1;
  return x;
}
```

However, the following code is equivalent:

```
int f( int x )
{
  x = x + g();
  return x;
}

int g()
{
  return 1;
}
```

In this latter example, g() always returns a value, 1. This value is external to function f(x). Likewise, despite the fact that g()'s output did not enter f(x) through its header, it is input to f(x). The true, total input to f(x) in the latter case must include the output of g() =1, as well as x = 3. So, input to a function can be pulled from within the body of a function in addition to whatever input is passed through its traditional header.

Conditional logic can also be employed within the body of a function, potentially resulting in different output for

different input. For example, in a common mathematical format:

$$g(x) = \begin{cases} 1 & if\ h(x) = 0 \\ 0 & otherwise \end{cases}$$

And in a program code format:

```
int g( int x )
{
 if ( h(x) == 0 )
   return 1;
 else
   return 0;
}
```

We've already demonstrated that one function's output can comprise another function's input. In both examples above, g(x)'s output is emphasized. And, it is easy to presume x as the limit of g(x)'s input. In English textbook descriptions, the function above would be commonly referred to as "g of x," as in the function g on the input variable x. In the strictest sense, however, x is not the complete input for g(x), as h(x) also has something to contribute, and without examining it, we do not know what other external inputs may be making their way into the body of h(x).

If an algorithm candidate sets up an input which is undefined, then that algorithm is not Turing decidable. No algorithm can accept undefined input. Likewise, a structure which sets up inescapable, undefined input, is incomplete. If any part of an algorithm candidate's always-invoked input relationship is undefined due to deadlock, then that algorithm candidate is undefined, is at best incomplete, and perhaps not an algorithm at all. Undefined input, by its definition, is incalculable. If, for a particular input i, an algorithm branches into an unresolvable deadlock, then the total input for f(x) at subset input i is undefined; any other true algorithm requiring f(x) and its input will not be able to accept f(x) and its total input, at subset input i, since the total input is undefined.

If all input an algorithm accepts and/or sets up is defined, then that algorithm is Turing decidable.
The relation on the natural numbers "Tx eventually halts when started with input y" *is* a Turing decidable relation, because for any algorithm where all input is defined, that algorithm is Turing decidable.

Some examples:

- while(true) {} is an algorithm since true is defined.
- The infinite loop:

```
function a() {
 return b()
}

function b() {
 return a()
}
```

Both a() and b() are algorithms, as each input is defined as the return of the other, and is equivalent to while(true), and there is no deadlock, just an infinite loop.

A representation of Turing's Halting Problem:

```
boolean main(f, i){
   return halt(f, i)
}
boolean halt(function f, inputSubset i){
   // Do amazing calculations here
   // and return whether f halts or
   // not, for inputSubset i (plus
   // any other input pulled within
   // f's body.)
}

void contrary(void){
   if(halt(contrary(), null))
     while(true)
}
```

Input to contrary() is output of halt() call, but input is undefined - not because of the null in the inputSubset call, but because halt() never receives all of contrary's input, because it cannot. Contrary's input is undefined due to deadlock. contrary() is an invalid algorithm because it sets up an inescapable deadlock; there is no input scenario where all of contrary's input is defined. halt(), however, is a legitimate, Turing-decidable algorithm because there is an infinitely large set of functions f with defined total input upon which it can operate.

Silberschatz, Galvin and Gagne (2012) provide us with the necessary conditions for a deadlock to occur, which we then apply to our halting problem here:

1. Mutual exclusion: at least one resource must be held in a non-sharable mode. In the case of the halting problem, the 'resource' is the input channel for halt(); while executing its very clever code, it realizes that contrary()'s input requires a return from halt() to decide whether contrary() halts or runs forever. But, halt() requires defined input for contrary() prior to returning true or false to halt() and subsequently to contrary(). So, halt(), by not returning, is effectively holding the input channel to contrary() open - contrary() will never get its answer, and therefore will never acquire defined input such that it can proceed with execution.
2. Hold and wait. As previously stated, halt() is holding open the input channel to contrary(); it cannot return a true or false to contrary(), prior to deciding whether it halts or runs forever, because a true or false return from halt() must come after a decision has been made.
3. No preemption. Only halt() can decide to release contrary's input channel by returning a true or false value. halt() cannot give an answer as to whether contrary() will run forever or not, so contrary() can

never acquire the defined input it needs to execute fully. contrary()'s definition puts halt() in full control of its input channel.

4. Circular wait. halt() requires defined input for contrary() in order to make a decision as to whether it will halt or run forever, prior to releasing contrary()'s input channel by returning an answer. contrary() is waiting for the return from halt() in its input channel, which is being held by halt(). Therefore, there is a circular wait in effect.

All four conditions hold for a deadlock to occur in the halting problem. Therefore, there is a deadlock upon trying to decide whether contrary() halts or runs forever. By definition, the execution of contrary() will always result in a deadlock. contrary()'s total input is always undefined due to this deadlock.

Deadlock is a condition between two or more processes which closes down an input channel. Whether an algorithm explicitly takes input at its onset, or receives input within its body during execution, or has input that is embedded atomically and statically within the structure of the algorithm itself, receiving actual input on all defined input channels is a fundamental requirement for the execution of a program. This paper is arguing that structures which are shown to be not effectively calculable also fail to meet this basic prerequisite and therefore fail to meet the basic requirement of qualifying as a legitimate problem. Asking for a response from a program when all of its input channels have not been satisfied is like asking for the numeric value of x+1 without first providing a value for x.

Experiment

Using the ideas stated above, an interesting exercise is to create a greatly simplified, working, subset version of halt(f, i) and a few example programs to be used against it. This is what was performed. To keep unnecessary complexity from obfuscating our simple exercise, some basic ground rules and assumptions were used:

- They were written as simplistic console C++ programs, and compiled on a Linux host using g++.
- while(true) participates as the sole example of a code block which runs forever.
- return 0 participates as the sole example of a code block which halts.
- halt(f, i) assumes the presented program files:
    - have already been verified as compilable – they follow proper syntax.
    - There is only a single if statement which may be present, and its condition deals specifically with making a call back to halt(f, i)
- Minimalist, easy-to-follow code parsing and logic is used within halt(f, i)
- The sample programs to be passed to halt have no input passed to them via their header; the only input they receive are within their bodies, a call to halt.

Our subset halt(f, i) was coded as halt.cpp, referenced via a header file in our scenario programs and primary executable main.x. halt's prototype:

```
bool halt(char* codeFileName,
        char* headerInput[]);
```

The primary executable takes a minimum of one argument, for the code to be analyzed:

```
usage: main.x <filename> <input parameter 1>
<input parameter 2> ...
```

The code in the file identified by filename, together with its input, is determined by the compiled code from halt.cpp, to either halt or run forever.

Results

In the process of analyzing how halt(f, i) might deal with a program which makes a decision based on the output of halt(f, i) itself, four different scenarios were discovered which help to gain some intuition about the nature of the absence of input due to deadlock.. All four scenarios use output from halt(f, i) as input:

1. A program file (alwaysrunsforever.cpp) which when compiled and executed always runs forever, regardless of input from halt(f, i)
2. A program file (alwayshalts.cpp) which when compiled and executed always halts, regardless of input from halt(f, i)
3. A program file (yesman.cpp) which when compiled and executed always agrees with halt(f, i), where f = the program's own code and i=its header's input.
4. A program file (contrary.cpp) which when compiled and run always negates halt(f, i)'s output, where f = the program's own code and i=its header's input.

The implementation shows the practical result of each scenario. What follows is a listing of the actual code for each of the four, followed by output of main.x when run against the code file, and a description of how halt(f, i) deals with the scenario:

```
// alwaysrunsforever.cpp
#include <iostream>
#include "halt.h"
using namespace std;
```

```
int main() // no header input
{
   if ( !halt( "alwaysrunsforever.cpp", NULL ) )
      while(true);

   while(true);
   return 0;
}
```

main.x output:

```
alwaysrunsforever.cpp runs forever.
```

Because this program always runs forever, regardless of the return from halt(f, i), halt(f, i) can safely return 'false' upon analyzing alwaysrunsforever.cpp's code and input.

```
// alwayshalts.cpp
#include <iostream>
#include "halt.h"
using namespace std;

int main() // no header input
{
   if ( halt( "alwayshalts.cpp", NULL ) )
      return 0;

   return 0;
}
```

main.x output:

```
alwaysrunsforever.cpp halts.
```

Because this program always halts, regardless of the return from halt(f, i), halt(f, i) can safely return 'true' upon analyzing alwayshalts.cpp's code and input.

```
// yesman.cpp
#include <iostream>
#include "halt.h"
using namespace std;

int main() // no header input
{
   if ( halt( "yesman.cpp", NULL ) )
      return 0;

   while(true);
   return 0;
}
```

main.x output:

```
yesman.cpp halts.
```

Because the program always agrees with halt's analysis, regardless of the return from halt(f, i), halt(f, i) can safely return 'true' upon analyzing yesman.cpp's code and input due to a general preference of procedures which halt over procedures which run forever. yesman.cpp is effectively giving full control of its termination behavior to halt(f, i).

```
// contrary.cpp
#include <iostream>
#include "halt.h"
using namespace std;

int main() // no header input
{
   if ( halt( "contrary.cpp", NULL ) )
      while(true);

   return 0;
}
```

main.x output:

```
Exception: Contradiction found: Missing input due
to deadlock.
```

Because the program always contradicts halt(f, i), halt(f, i) cannot return an answer. There is a deadlock, as described earlier. This particular deadlock means part of contrary.cpp's input is forever missing. The code compiled from halt.cpp throws an error, similar to what is done if the file portion of halt's input is unreadable.

Conclusion

Artificial intelligence models give us insight in how to deal with impossible relationships which result in undefined or absence of input; eliminate it as invalid. Turing's halting problem asks if there could be a program, given another program and its input - can it answer: does it halt or run forever? The answer is connected to the phrase *and its input*. This paper asserts that for all true algorithms, for which valid input can be supplied, the answer is yes.

References

Davis, M. 1965. *The Undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions.* New York: Raven Press.

Gray, J. 2000. *The Hilbert Challenge*, 169-170. Oxford: Oxford University Press.

Nagel, E., Newman, J., and Hofstadter, D. eds. 2001. *Gödel's Proof*, 98-101. New York: New York University Press.

Rosen, K. 1999. *Discrete Mathematics and Its Applications, 4/e*, 181-182. Boston, Mass.: WCB/McGraw-Hill.

Russell, S. and Norvig, P. eds. 2010. *Artificial Intelligence: A Modern Approach, 3/e*, 379-381. Upper Saddle River, New Jersey: Pearson.

Silberschatz, A., Galvin, P., and Gagne, G. 2012. *Operating Systems Concepts, 8/e*, 285-287. New Jersey: John Wiley & Sons.

Turing, A. 1937. On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 2(42): 230-265.