Module Home Page
Title Page
Page 1 of 14
Back
Full Screen
Close
Quit

Lecture 36:
Other Non-computable Problems

Aims:

- To show how to prove that other problems are non-computable, which involves reductions from, e.g., the Halting Problem; and
- To point out how few problems are, in fact, computable.

Module Home Page
Title Page
Page 2 of 14
Back
Full Screen
Close
Quit

36.1. Introduction

- There are many problems for which there is no algorithm. In fact, the number of things which can be computed is infinitesimal compared with the number of things one might like to compute but which cannot be computed.
- To prove a problem P is non-computable,
  - We could give a proof similar to the one we gave for the Halting Problem. This would be a direct but tedious way of proving non-computability.
  - But there is an indirect way of proving non-computability which is usually easier. We exploit the fact that we already have one problem that has been proved to be non-computable, i.e. the Halting Problem. The proof will still be a proof by contradiction; it will also use a reduction.

    We assume that P is computable. Then we show that, if this assumption is true, then the Halting Problem would be computable. But we know the Halting Problem is non-computable. Contradiction! So our assumption is false: P is non-computable.

Module Home Page

Title Page

Page 3 of 14

Back

Full Screen

Close

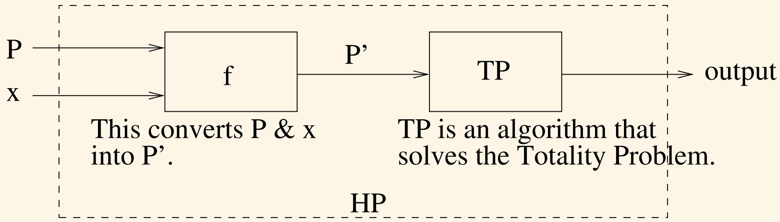
Quit

36.2. The Totality Problem

- We'll prove the following is non-computable:

Problem 36.1. *The Totality Problem*
Parameters: *A MO$_{CC}$A program P.*
Returns: *YES if P would terminate for all its inputs; NO otherwise.*

- Assume that the Totality Problem is computable.
- I.e. we have a MO$_{CC}$A program TP that solves the Totality Problem.
- But in that case, we can write a MO$_{CC}$A program HP to solve the Halting Problem. It will use program TP as a procedure.



HP is an algorithm that solves the Halting Problem.

- So, HP takes in two inputs P and x.
- Function f uses these two inputs to write a new program called P', which

Module Home Page
Title Page
◀◀ ▶▶
◀ ▶
Page 4 of 14
Back
Full Screen
Close
Quit

- takes in an input y but ignores it;
- runs P on x

In other words, f outputs the following program:

```
Algorithm:  P'(y)
 // Ignore y
 P(x);
```

- P' is then the input to TP (the program that solves the Totality Problem).
- So we're using TP to find out whether P' halts on all inputs.
- But what P' does is: ignore its input and simply run P on x.
- So asking whether P' halts on all inputs is the same as asking whether P halts on x.
- So we've managed to write a program that solves the Halting Problem!
- We know that no such program can exist, so there must be something wrong with what we've done. There's nothing wrong with f, so the only part that can be held responsible is TP.
- We conclude that a program TP, solving the Totality Problem, cannot exist.

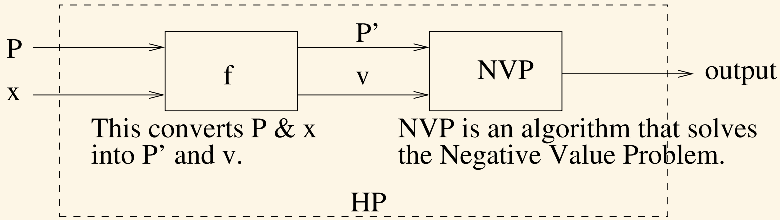
36.3. The Negative Value Problem

- We'll prove the following is non-computable:

Problem 36.2. *The Negative Value Problem*

Parameters:	*A MO$_{CC}$A program P that does not require any input and a variable v used in P.*
Returns:	*YES if v ever gets assigned a negative value when P is executed; NO otherwise.*

- Assume that the Negative Value Problem is computable.
- I.e. we have a MO$_{CC}$A program NVP that solves the Negative Value Problem.
- But in that case, we can write a MO$_{CC}$A program HP to solve the Halting Problem. It will use program NVP as a procedure.



HP is an algorithm that solves the Halting Problem.

- So, HP takes in two inputs P and x.

- Function f scans P to identify a variable v that is not used by P and then writes a new program called P', which
  - takes in an input y but ignores it;
  - runs P on x;
  - then assigns -1 to v.

  In other words, f outputs the following program:

  Algorithm: P'(y)

  // Ignore y
  P(x);
  $v := -1$;

- P' and v are then the input to NVP (the program that solves the Negative Value Problem).
- So we're using NVP to find out whether P' assigns a negative value to v.
- But what P' does is: ignore its input, run P on x and assign -1 to v.
- So asking whether P' ever assigns a negative value to v is the same as asking whether P halts on x. (Why? Because we'll only get to the command in which -1 is assigned into v if P halts on x.)
- So we've managed to write a program that solves the Halting Problem!
- We know that no such program can exist, so there must be something wrong with what we've done. There's nothing wrong with f, so the only part that can be held responsible is NVP.

Module Home Page
Title Page
Page 7 of 14
Back
Full Screen
Close
Quit

- We conclude that a program NVP, solving the Negative Value Problem, cannot exist.

Module Home Page
Title Page
◀◀ ▶▶
◀ ▶
Page 8 of 14
Back
Full Screen
Close
Quit

36.4. Reductions Again

- You can see that, in these proofs, the non-computability of one problem is being established by finding a *reduction* from a problem that is already known to be non-computable.
- In the examples, we have shown

  The Halting Problem reduces to The Totality problem
  The Halting Problem reduces to The Negative Value Problem

- If P_1 is known to be non-computable and P_1 reduces to P_2, then P_2 must be non-computable too. The reason is that, otherwise, we could solve P_1 by an algorithm that would transform P_1's inputs into a suitable form and ask P_2 for the answer.
- This is like the reductions we were using to show that a problem is **NP**-hard. The difference there was that we also required the reduction to have worst-case polynomial time complexity, whereas here efficiency is not the issue so the reduction can use as much resource as it needs.
- In both cases, once we have such a reduction, P_1 cannot be worse than P_2.
- It's common to show reductions from the Halting Problem. But any problem that has been proved to be non-computable can be used. E.g. now that we know that the Totality Problem is non-computable, we can use that, if we wish, in future proofs. Indeed, in the next section, we use the Totality Problem to prove the non-computability of the Equivalence Problem. We do this by showing

  The Totality Problem reduces to The Equivalence Problem

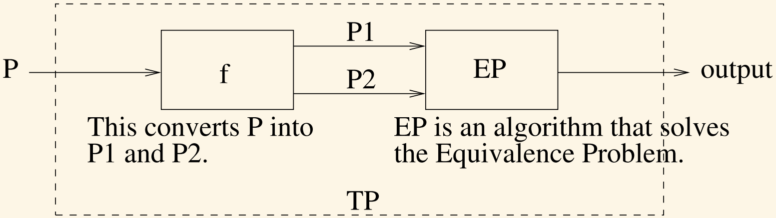
36.5. The Equivalence Problem

- We'll prove the following is non-computable:

Problem 36.3. *The Equivalence Problem*

Parameters:	*Two MOCCA programs P_1 and P_2.*
Returns:	*YES if P_1 and P_2 solve the same problems (same outputs for same inputs); NO otherwise.*

- Assume that the Equivalence Problem is computable.
- I.e. we have a MOCCA program EP that solves the Equivalence Problem.
- But in that case, we can write a MOCCA program TP to solve the Totality Problem. It will use program EP as a procedure.



TP is an algorithm that solves the Totality Problem.

- So, TP takes in one input P.

Introduction
The Totality Problem
The Negative Value . . .
Reductions Again
The Equivalence Problem
Rice's Theorem
Concluding Remarks

Module Home Page
Title Page
◀◀ ▶▶
◀ ▶
Page 10 of 14
Back
Full Screen
Close
Quit

- Function f uses this input to write a new program called P_1, which
  - takes in an input x;
  - runs P on x;
  - returns "CS2205" (for example)

  and it also writes a new program P_2, which
  - takes in an input x but ignores it;
  - returns "CS2205"

  In other words, f outputs the following programs:

Algorithm: $P_1(x)$

P(x);
return "$CS2205''$;

Algorithm: $P_2(x)$

// Ignore x
return "$CS2205''$;

- P_1 and P_2 are then the input to EP (the program that solves the Equivalence Problem).
- So we're using EP to find out whether P_1 and P_2 are equivalent.
- But what P_1 does is run P on x and return "CS2205", and what P_2 does is return "CS2205".
- So asking whether P_1 and P_2 are equivalent (same outputs for same inputs) is the same as asking whether P halts on all inputs.

Module Home Page
Title Page
Page 11 of 14
Back
Full Screen
Close
Quit

- So we've managed to write a program that solves the Totality Problem!
- We know that no such program can exist, so there must be something wrong with what we've done. There's nothing wrong with f, so the only part that can be held responsible is EP.
- We conclude that a program EP, solving the Equivalence Problem, cannot exist.

36.6. Rice's Theorem

- Maybe you're feeling that almost any interesting question that we ask about algorithms is not computable. You'd be basically right!
- Virtually all problems that involve writing a program that takes in another program P and tries to answer a question about the behaviour of P are non-computable.
- *Rice's Theorem.* Think of a task that some algorithms perform and others do not (such as outputting 28, computing the square root of the input, always giving the same output irrespective of the input, etc). There is no algorithm that can take in an arbitrary program P, inspect P and tell whether P performs that task.
- This has consequences for compilers and virus checkers.

Module Home Page
Title Page
◀◀ ▶▶
◀ ▶
Page 13 of 14
Back
Full Screen
Close
Quit

36.7. Concluding Remarks

- We've only grazed the surface of this topic!
- Some of the non-computable problems are less computable than others.
- E.g. the Halting Problem and the Totality Problem are both non-computable but the Totality Problem is less computable than the Halting Problem!
- In fact, there is an infinite hierarchy of levels of (non-)computability!

Acknowledgements

I based some of this on [Har92] and [GL82].

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.

References

[GL82] L. Goldschlager and A. Lister. *Computer Science: A Modern Introduction*. Prentice-Hall, 1982.

[Har92] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2nd edition, 1992.