CS120: Intro. to Algorithms and their Limitations Anshu & Hesterberg

Lecture 17: The Church-Turing Thesis

Harvard SEAS - Fall 2023 *2023-11-02*

1 Announcements

- Recommended reading: MacCormick §5.6–5.7, §7.7–7.9

2 Loose Ends: Efficient algorithm for 2-SAT

Our goal is to prove the following theorem:

Theorem 2.1. *There is an algorithm that solves the computational problem 2-SAT in time $O(nm)$, where n is the number of variables and m is the number of clauses.*

The algorithm is as follows.

```
1  TwoSATSolve(φ)
   Input  : A CNF φ with width 2
   Output : A satisfying assignment, else ⊥
2  Construct the implication graph G = (V, E);
3  Set (x_0, ..., x_{n-1}) = ("unassigned", ..., "unassigned");
4  foreach i = 0, ..., n − 1 do
5  |  dist_{1,i} =BFS(G, x_i, ¬x_i)
6  |  dist_{2,i} =BFS(G, ¬x_i, x_i)
7  |  if dist_{1,i} < ∞ AND dist_{2,i} < ∞ then return ⊥;
8  foreach i = 0, ..., n − 1 do
9  |  if x_i is assigned then i = i + 1 and GOTO line 8;
10 |  if dist_{1,i} = ∞ AND dist_{2,i} < ∞ then
11 |  |  Set x_i = 1
12 |  |  For all variables x_j such that there is a path from x_i to x_j, set x_j = 1.
13 |  |  For all variables x_j such that there is a path from x_i to ¬x_j, set x_j = 0.
14 |  if dist_{1,i} < ∞ AND dist_{2,i} = ∞ then
15 |  |  Set x_i = 0
16 |  |  For all variables x_j such that there is a path from ¬x_i to x_j, set x_j = 1.
17 |  |  For all variables x_j such that there is a path from ¬x_i to ¬x_j, set x_j = 0.
18 |  Delete all the edges incident to or from all assigned literals.
19 Assign the remaining variables such that the literals in each connected component of
     resulting graph G get the same value.
20 return (x_0, ..., x_{n-1});
```

Runtime: The loops on Lines 4 and 8 each run for at most n iterations. In each iteration, Lines 5 and 6 take $O(n+m)$ steps. Lines 12, 13, 15, and 16 take at most $O(n)$ steps since it suffices

consider the vertices discovered by the BFS from x_i (Line 5) and $\neg x_i$ (Line 6). Line 18 takes $O(m)$ steps and Line 19 can be implemented in $O(n+m)$ steps (see section). Thus, the overall runtime is $O(n(n+m))$. Note that $m = \Omega(n)$, which simplifies that runtime to $O(nm)$.

Correctness: Lines 4-7 identify if there is variable such that "x_i implies $\neg x_i$" and "$\neg x_i$ implies x_i" (a *bad cycle* - see Figure 4 in Lec 16). If so, the input formula is unsatisfiable, and we return so.

Suppose the algorithm does not abort (return $\perp$) at this stage. Then we claim that the algorithm finds a satisfying assignment. Lines 10 and 14 check whether any literal implies its negation: if $\neg x_i$ implies x_i, we set $x_i = 1$ (similar to Figure 3 in Lec 16), and if x_i implies $\neg x_i$, we set $x_i = 0$. Since there were no *bad cycles* before, and we set variable values only according to implications, there are still no *bad cycles* and no false clauses. Lines 12, 13, 16, and 17 set the values of variables only according to implications, which don't create any bad cycles. Line 18 deletes only edges which are redundant (as 'anything implies TRUE' and 'FALSE implies anything'). This step does not create any new bad cycles - if a digraph has no bad cycles, a subgraph cannot have a bad cycle. Further, this step removes any path from x_i to $\neg x_i$ or vice-versa (whichever existed).

After Line 18 has been executed, we have ensured that there are no paths between any two literals of the same variable, and there are still no bad cycles. Thus, the graph is divided into at least two connected components (similar to Figure 2 in Lec 16), with only one variable for each literal in each connected component. Since 'FALSE implies FALSE' and 'TRUE implies TRUE', we can arbitrarily pick one of TRUE and FALSE to assign to each literal in a connected component: at the end of this, we've assigned a value to each variable and not violated any implications, that is, we've found a satisfying assignment. Note that this also proves Theorem 3 in Lec 16.

3 Introduction to Limits of Computation

Thus far in CS 120, we've focused on what algorithms can do, or what they can do efficiently. In the remainder of the course, we'll talk about what algorithms can't do, or can't do efficiently.

In particular, recall Lecture 3's lemma about reductions:

Lemma 3.1. *Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:*

1. *If there exists an algorithm solving Γ, then there exists an algorithm solving Π.*
2. *If there does not exist an algorithm solving Π, then there does not exist an algorithm solving Γ.*
3. *If there exists an algorithm solving Γ with runtime $g(n)$, and $\Pi \leq_{T,f} \Gamma$, then there exists an algorithm solving Π with runtime $O(T(n) + g(f(n)))$.*
4. *If there does not exists an algorithm solving Π with runtime $O(T(n)+g(f(n))$, and $\Pi \leq_{T,f} \Gamma$, then there does not exist an algorithm solving Γ with runtime $O(g(n))$.*

In the last unit of the course, we'll use the item 2: we'll find a problem Π which we can prove is not solved by any Word-RAM algorithm, then reduce Π to other problems Γ to prove that no Word-RAM algorithm solves them.

Similarly, in the upcoming second-last unit of the course, we'll use item 4: we'll assume that the problem $\Pi = SAT$ is not solved quickly by any Word-RAM algorithm, then reduce SAT to other problems Γ to prove that no Word-RAM algorithm solves them quickly.

Before we do so, let's consider how fundamental Word-RAM is to the statements above. That is, if we prove limitations of Word-RAM programs, are those limits specific to Word-RAM or are they more general/independent of technology? Could find substantially faster algorithms by choosing a different model of computation than Word RAM, like Python or Minecraft? The answer is conjectured to be "no".

To explain why, we'll first recall our simulation arguments which state that the same problems are solvable by Word-RAM programs, Python programs, and so on.

4 The Church–Turing Thesis

Theorem 4.1 (Turing-equivalent models)**.** *If a computational problem* Π *is solvable in one of the following models of computation, then it is solvable in all of them:*

- *RAM programs*
- *Word-RAM programs*
- *XOR-extended RAM or Word-RAM programs*
- *%-extended RAM or Word-RAM programs*
- *Python programs*
- *OCaml programs*
- *C programs (modified to allow a variable/growing pointer size)*
- *Turing machines*
- *Lambda calculus*
- $\vdots$

Moreover, there is an algorithm (e.g. a RAM program) that can transform a program in any of these models of computation into an equivalent program in any of the others.

The Church–Turing Thesis: The equivalence of many disparate models of computation leads to the Church–Turing Thesis, which has (at least) two different variants:

1. The (equivalent) models of computation in Theorem 4.1 capture our intuitive notion of an algorithm.
2. Every physically realizable computation can be simulated by one of the models in Theorem 4.1.

This is not a precise mathematical claim, and thus cannot be formally proven, but it has stood the test of time very well, even in the face of novel technologies like quantum computers (which have yet to be built in a scalable fashion); every problem that can be solved by a quantum algorithm can also be solved by a RAM program, albeit much more slowly.

Proof idea: A theorem like this is proven via "compilers" and simulation arguments like we have seen several times, giving a procedure to transform programs from one model to another (e.g. simulating XOR-extended Word-RAMs by ordinary Word-RAMs). Like we have seen, we can write simulators for RAM programs in high-level languages like Python and OCaml, and conversely those high-level languages are compiled down to assembly code, which is essentially Word-RAM code.

Simple and elegant models: The λ calculus and Turing machines are extremely simple (even moreso than the RAM model) and mathematically elegant models of computation, coming from the work of Church and Turing, respectively, in 1936, in their attempts to formalize the concept of an algorithm (prior to, and indeed inspiring, the development of general-purpose computer technology). Turing machines are similar to the Word-RAM model, but with a fixed word size and memory access only at a pointer that moves in increments of ± 1. We won't have time to describe the lambda calculus, but it provided the foundation for future functional programming languages like OCaml, and one of the theorems in Turing's paper established the equivalence of Turing machines and the λ calculus.

Input encodings: One detail we are glossing over in Theorem 4.1 is that the different models have different ways of representing their inputs and outputs. For example, natural numbers can be represented directly in RAM programs, but in a Turing machine they need to be encoded as a string (e.g. using binary representation), and in the lambda calculus, they are represented as an operator on functions (which maps a function $f(x)$ to $f^{(n)}(x) = f(f(\cdots f(x))))$. So to be maximally precise, these models are equivalent up to the representation of input and output.

4.1 The Strong (or Extended) Church–Turing Thesis

The Church–Turing hypothesis only concerns problems solvable at all by these models of computation (Word-RAM programs, etc.). We haven't even seen any problems that are *not* solvable by Word-RAM programs—that will be a topic for the end of the course. There is, however, a stronger version of the Church–Turing hypothesis that also covers the efficiency with which we can solve problems.

> Extended Church–Turing Thesis v1: Every physically realizable model of computation can be simulated by a Word-RAM program (or Turing Machine, or Python program) with only a *polynomial* slowdown in runtime. Conversely, any physically realizable model of computation can simulate Word-RAM programs in real time only polynomially slower than their defined runtime.

The Strong Church–Turing Thesis is not a precise mathematical claim, and thus cannot be formally proven. In fact, randomized algorithms, massively parallel computers, and quantum computers all could potentially provide an exponential savings in runtime. (For randomized algorithms, however, it is conjectured that they provide only a polynomial savings, as discussed in Lecture 8.)

If we modify the statement with some qualifiers, then these challenges no longer apply:

> Extended Church–Turing Thesis v2: Every physically realizable, deterministic, and sequential model of computation can be simulated by a Word-RAM program (or Turing Machine, or Python program) with only a polynomial slowdown in runtime. Conversely,

> any physically realizable, deterministic and sequential model of computation can simulate Word-RAM programs in real time only polynomially slower than their defined runtime.

"Deterministic" rules out both randomized and quantum computation, as both are inherently probabilistic. "Sequential" rules out parallel computation. This form of the Extended Church–Turing Thesis has stood the test of time for the approximately fifty years since it was formulated, even as computing technology has changed tremendously in that time.

Note: in contrast to the above claims about Word-RAM, we had a pset where Word-RAM simulated RAM with an exponential slowdown, and our RAM to Word-RAM simulation theorem also has a slowdown factor that can get exponentially large (due to bitlength of numbers). So, the choice of base model (Word-RAM) is important here in a way it isn't for the regular Church–Turing Thesis.

Considering computational efficiency when comparing models of computation will be the subject of the next few lectures.