Great Ideas in Computing

University of Toronto CSC196
Fall 2023

Week 6: October 23 - October 27 (2023)

Week 7 slides

Announcements:

- I have posted Assignment 2 which is due October 27, 9AM.
  I will give an example of a neural net this morning so as to help with the last question on the assignment.
- The quizzes have been graded by Aniket and I will hand them back on Wednesday after I look them over. For those that did not do well, there are plenty of opportunities to still get a good grade for this course. In particular, **participate in class and tutorial!**
- On November 1, our second guest Ashkay Srinivasan, will lead a discussion on "modern cryptography'. **NOTE:** I previously said October 31 which is a Tuesday. The talk is next Wednesday, November 1.

The weeks agenda

This weeks agenda

- A neural net computing f:

$$y = f(x_1, x_2, x_3, x_4, u) = \begin{cases} 1 & \text{if } u = x_i \quad \text{for some } i. \\ 0 & \text{otherwise} \end{cases}$$

  with the same activation function

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

- Continue discussion of Search Engines
- Begin complexity theory

Why is search so profitable?

Companies such as IBM and (initially) Microsoft did not try to commercialize search, not recognizing the profitability of search. Indeed, should one charge for information or should the business model be based on advertising? Or it possible that search would not be profitable?

We now know that search has turned out to be extremely profitable for companies based on advertising. The main way that Google and other comapnies sell advertising for search has spawned major research in algorithm design and auction theory. We will say more about auctions, game theory and mechanism design.

We can view the process of assigning queries to advertisers (say wanting to display an *ad* as an *online biparitite graph matching problem*).

When a query arrives it needs to be assigned to one (or more, depending on how many advertising slots will be displayed) ads.

The "adwords" assignment problem

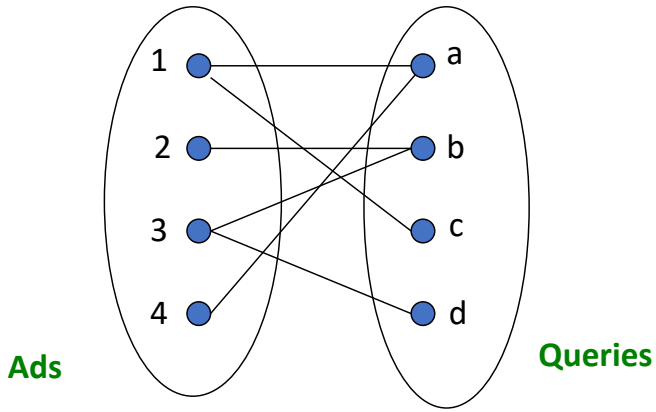


Figure: Figure taken from USC lecture notes by Rafael Ferreira da Silva

Each advertsiser may have a budget (say for a given day) and indicates for given queries (or keywords) what it is willing to pay for that query but never exceeeding its budget for all the queries assigned to that advertiser.

The search engine adjusts this advertiser *bid* for a query based on how well it thinks the ad matches the query and then decides whether or not to assign an advertising slot to an advertiser and the price paid by the advertiser (depending on the slot) for each click by search users for the ad.

The semantic web

We will end our discussion of search engines about where we began when I said, like other great ideas, sometimes these great ideas become so entrenched that it is hard to make further progress.

Is this the case with key word search? What kinds of "information needs" are beyond today's search engines? See 2008 "Ontologies and the Semantic Web" article by Ian Horrocks and also his 2005 Lecture by the same title.

The vague goal of the semantic web is "to allow the vast range of web-accessible information and services to be more effectively exploited by both humans and automated tools."

A more specific goal is to *integrate* information that occurs in the web but not in one decument.

Is our experience with large language models encouraging us to use longer queries?

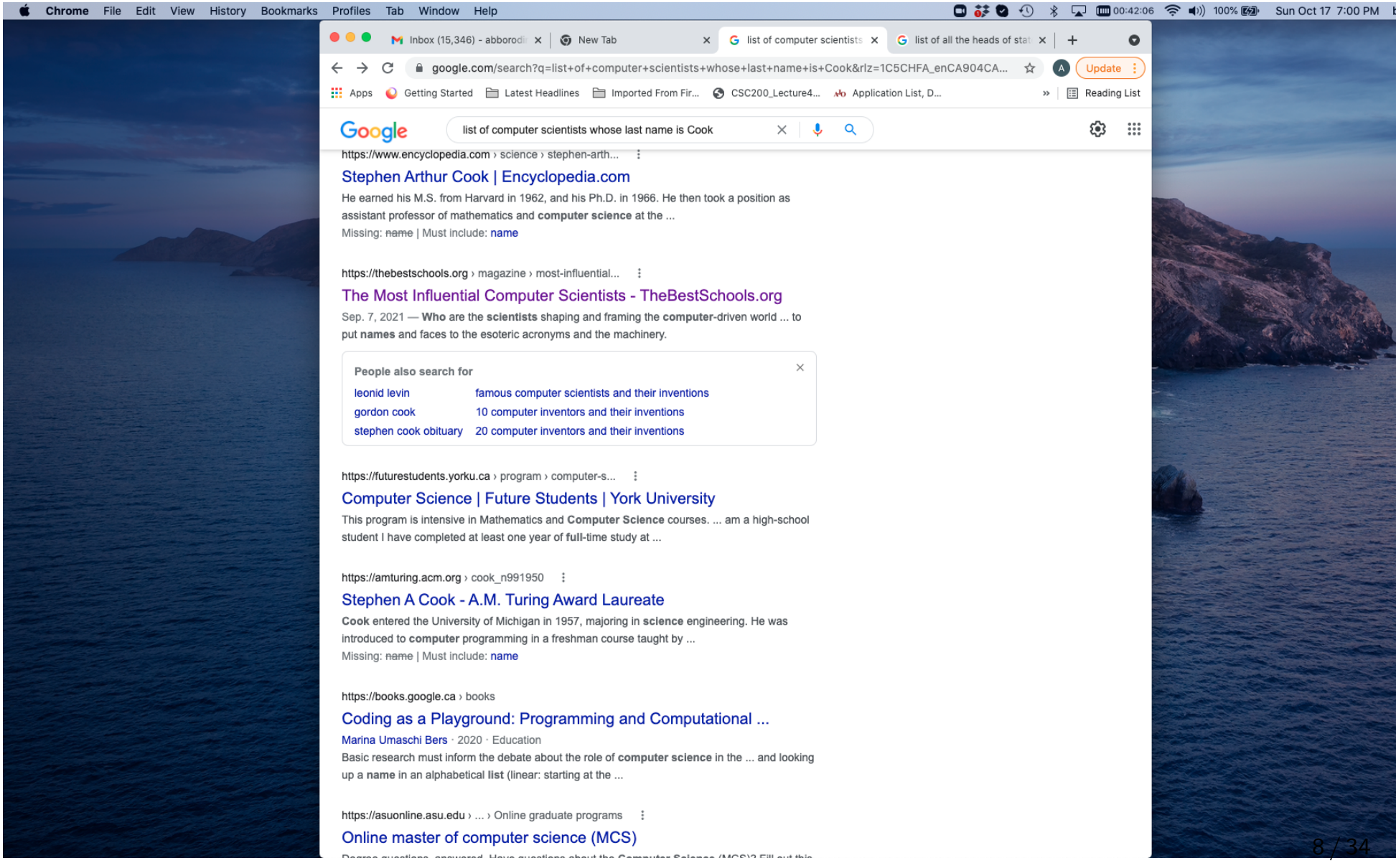
Some specific examples of information that might not exist in any one document

One example Horrrocks gave is to retrieve a "list of all the heads of state of EU countries". Of course, once such an example is given, it is likley (as in this example) that one can successfully find the required information in a single query. (Why was this a difficult search in 2008 and an easy search today? It was the fourth document in my search on October 17,2021.

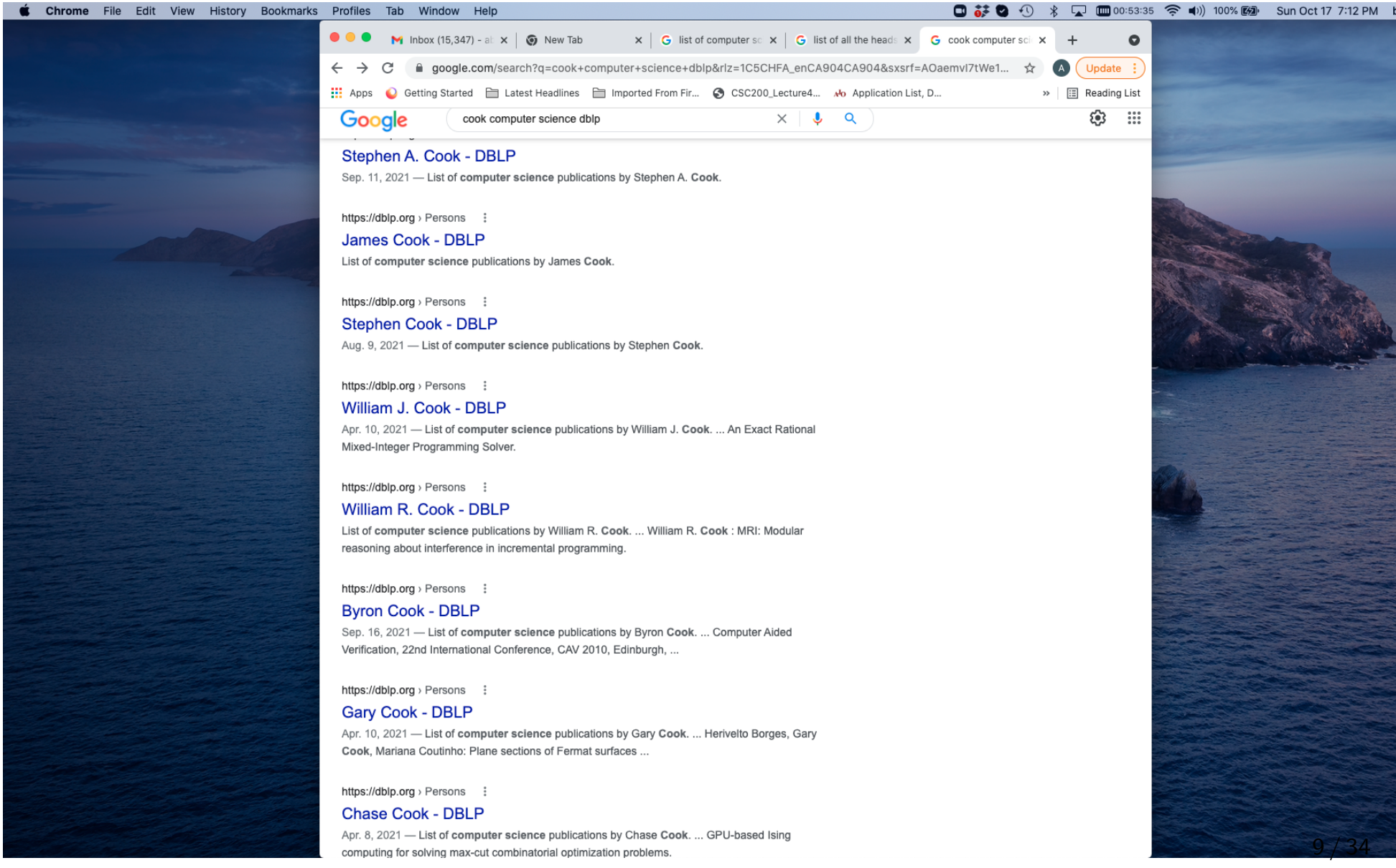
"The classic example of a semantic web application is an automated travel agent that, given various constraints and preferences, would offer the user suitable travel or vacation suggestions". This example still seems beyond something we can easily do with current search engines.

I decided to create the following query "list of all computer scientists whose last name is Cook". In my first search, most of the retrieved documents are not useful but the first of the retrieved documents is for Stephen Cook and the second document is a very incomplete list of computer scientists.

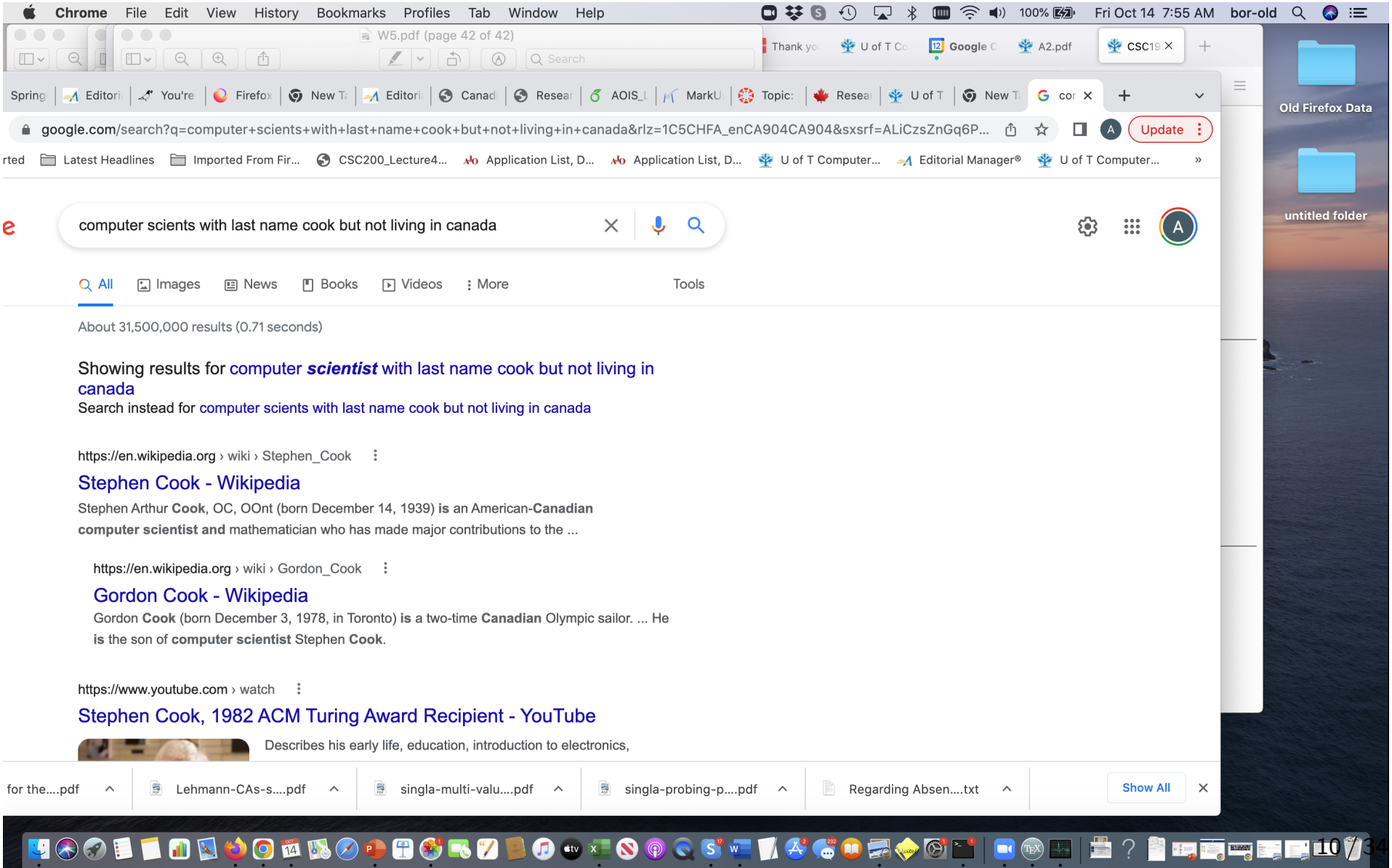
Screenshot of my query for computer scientists with last name Cook



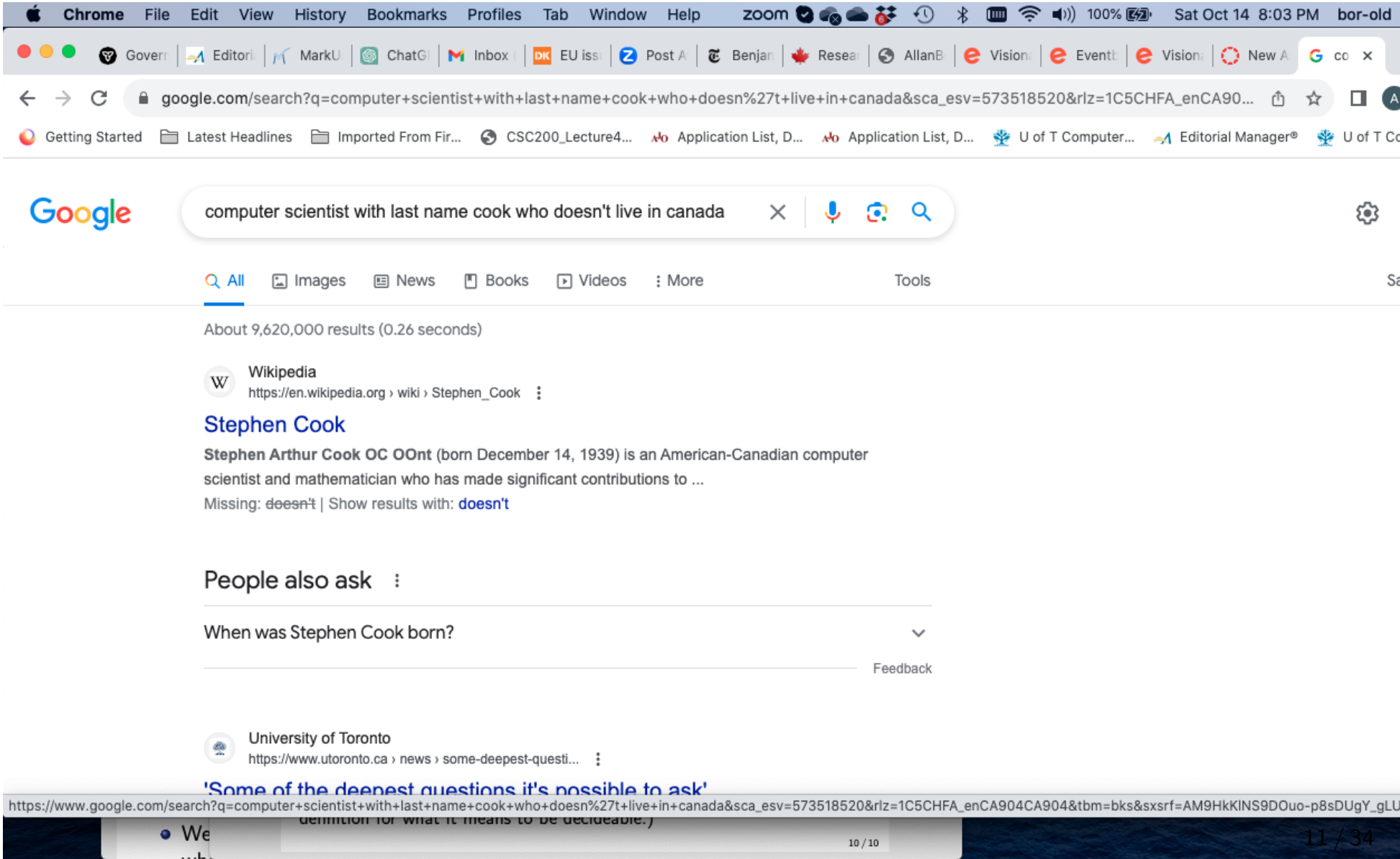
Another search to find other computer scientists with last name Cook



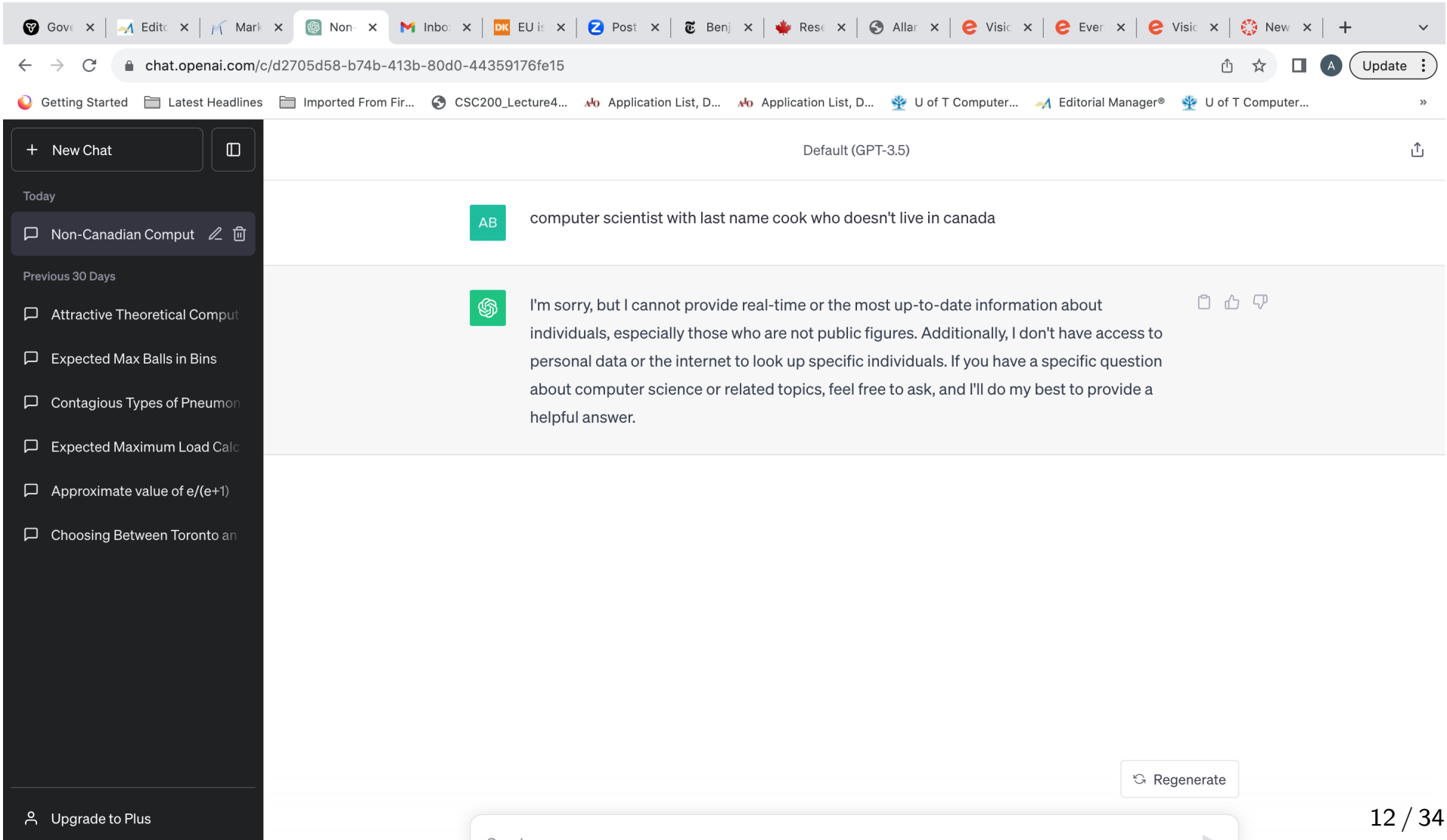
October 14, 2022 search to find a computer scientist named Cook not living in Canada.



October 14, 2023 search to find a computer scientist named Cook not living in Canada.



October 14, 2023 search using Chat-GPT3.5 to find a computer scientist named Cook not living in Canada.



Query: Australians doing research in theoretical computer science

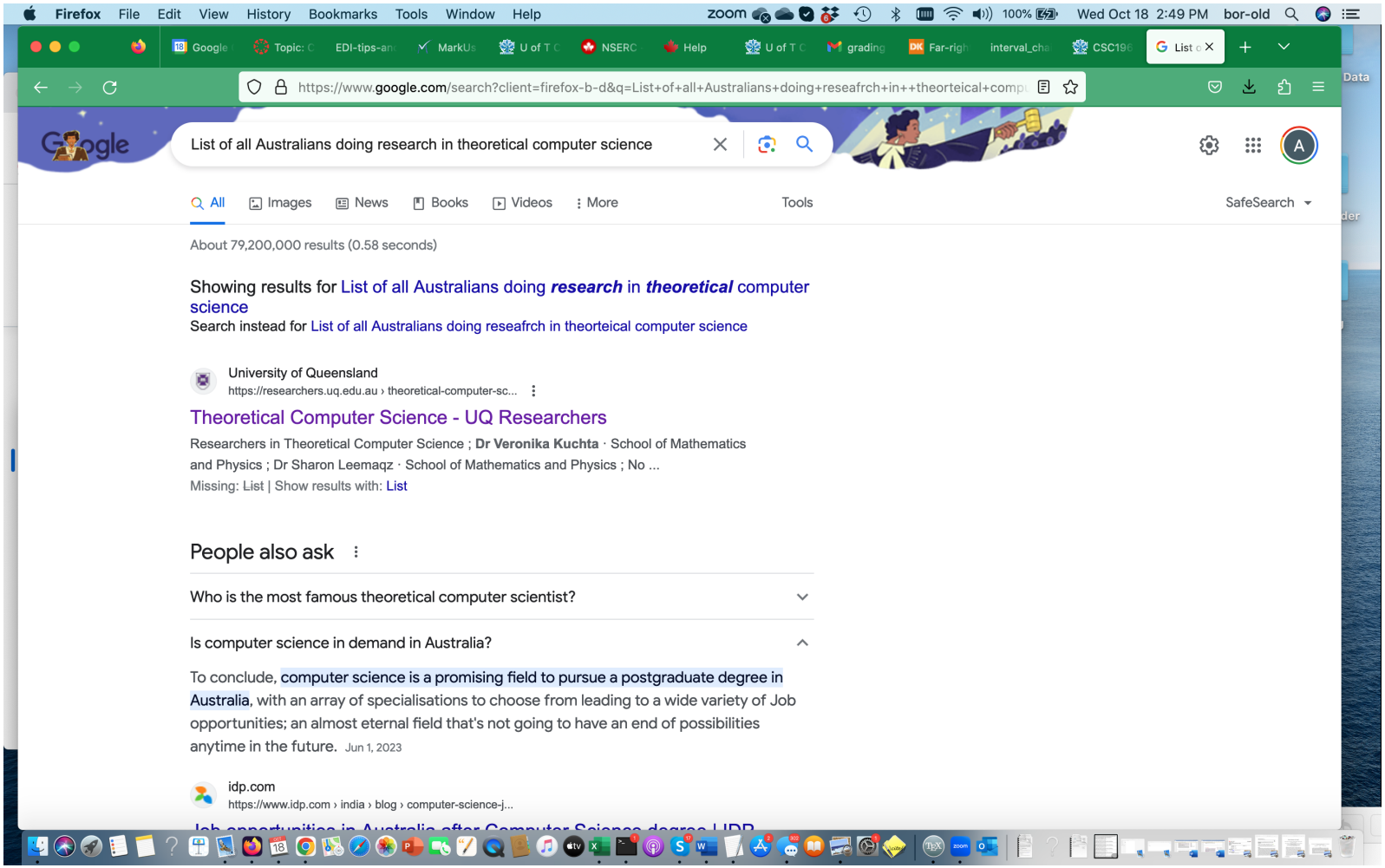


Figure: Response to query about Australians in TCS research

What would you do next?

Photo queries

Today we can have queries such as “find me all documents where this exact photo exists” or “find me a document that contains a photo closest to the query photo”. “find me all document that contain a photo like the one in this document”

If the photo comes from a document with text an especially if the photo has a caption, we might already have enough informartion about the photo to do a key woed search.

What if the photo is just something you scanned? One way this can be done (and perhaps this is the main idea) is to treat the photo as a vector compriseds on the pixels. Then we can have a indexed list of photos (each represensented as a vector) and then the problem becomes a well studied problem in computational geometry; namely, the problem of *nearest neighbour search in high dimensions*.

Of course, we (or the search engine) can use a deep learning algorithm to classify a given photo before searching for similar photos.

Complexity theory; the extended Church-Turing thesis

We recall the Church-Turing thesis, namely that every computable function f is Turing computable. More precisely, there is a Turing machine M such that on every input x, M halts and outputs $f(x)$. That is, the Church-Turing thesis equates the informal concept of "computable" with the matehmatically precise concept of "Turing machine computable".

The extended Church-Turing thesis equates the informal concept of "efficiently computable" with the mathematical precise concept of computable by a Turing machine in polynomial time".

More precisely, the extended Church-Turing thesis states that a function f is efficiently computable if there is a Turing machine M and a polynomial $p(n)$ such that on every input x, M halts in at most $p(|x|)$ steps and outputs $f(x)$.

Here we are assuming $x \in \Sigma^*$ for some finite alphabet Σ and $|x|$ represents the length of the string x.

The extended Church Turing thesis continued

In what follows, I will use n to be the length of a an input string; $n = |x|$.

Do we believe the extended Church Turing Thesis?

Why we might accept the extended Church Turing thesis

- We can simulate in polynomial time a random access von Neumann random access machine if we say, as we should, that the time for basic operations on bit operands is $O(1)$. This is a robust definition.
- That is, there is a polynomial function $p_2()$ such that if a function f is computable in time $p_1(n)$ on a von Neumann random access machine, then f is computable in polynomial time $p(n) = p_2(p_1(n))$ on a Turing machine. For example, if $p_1(n) = n^3$ and $p_2(n) = n^2$ then $p(n) = n^6$.
- For problems involving say enormous graphs, we may need sublinear time; for other problems we may need linear or near linear times. But as an abstraction, we are saying that a polynomial time algorithm is "efficient".

Why we should be less accepting of the extended Church-Turing thesis

While we are very confidant about the Church-Turing thesis (for defining "computable"), there are various reasons to be a little more skeptical about the extended Chruch-Turing thesis.

- An algorithm running in a polynomial time bound like n^{100} is not an efficient algorithm.
- An algorithm running in an exponential time bound like $(1 + \frac{1}{1000})^n$ is an efficient algorithm for reasonably (but not too) large input lengths. Note: $(1 + \frac{1}{k})^k \to e \approx 2.72$
- While we can simulate classical computers (i.e. von Neumann machines) in polynomial time, we do not know how to simulate non classical computers (e.g., quantum computers) in polynomial time.
- Factoring is an example of a problem that can be computed in polynomial time by a quantum computer whereas we do not believe factoring is polynomial time computable on a classical computer. So it is possible that we will have to change of definition of "efficiently computable" to be polynomial time on a quantum computer.

So should we accept the extended Church-Turing thesis?

We can accept the extended Church-Turing thesis, arguing as follows:

- Polynomial time computable functions usually have reasonably small asymptotic polynomial time bounds; that is, $n, n \log n, n^2, n^3$‘. There are some exceptions (like n^6, but generally speaking we don't usually encounter polynomial time bounds asymptotically bigger than n^3.
- The robustness of polynomial time (in terms of being closed under composition is not sensitive to the precise model of computing and definition of a time step. This enables us to define our concepts in terms of Turing machines (once we restrict outselves to classical computer models). Linear functions are also closed under composition but linear time computation is very model dependent.
- While non-classical models may contradict the thesis, **so far** we do not have non-classical computers (e.g., quantum computers that go beyond a small number of quantum bits) that are practical in a commercial sense.

End of Monday, October 23, 2023

Agenda for Wedmesday, October 25.

- Discuss the quiz.
- I will repeat the last few slides an then contiunue with the (literally) million \$ P vs NP question. Namely,
  1. Defining the classes P and NP
  2. NP cpmpleteness
  3. The $P \neq NP$ *conjecture* and its importance.

Complexity theory; the extended Church-Turing thesis

We recall the Church-Turing thesis, namely that every computable function f is Turing computable. More precisely, there is a Turing machine M such that on every input x, M halts and outputs $f(x)$. That is, the Church-Turing thesis equates the informal concept of "computable" with the matehmatically precise concept of "Turing machine computable".

The extended Church-Turing thesis equates the informal concept of "efficiently computable" with the mathematical precise concept of computable by a Turing machine in polynomial time".

More precisely, the extended Church-Turing thesis states that a function f is efficiently computable if there is a Turing machine M and a polynomial $p(n)$ such that on every input x, M halts in at most $p(|x|)$ steps and outputs $f(x)$.

Here we are assuming $x \in \Sigma^*$ for some finite alphabet Σ and $|x|$ represents the length of the string x.

The extended Church Turing thesis continued

In what follows, I will use n to be the length of a an input string; $n = |x|$.

Do we believe the extended Church Turing Thesis?

Why we might accept the extended Church Turing thesis

- We can simulate in polynomial time a random access von Neumann random access machine if we say, as we should, that the time for basic operations on bit operands is $O(1)$. This is a robust definition.
- That is, there is a polynomial function $p_2()$ such that if a function f is computable in time $p_1(n)$ on a von Neumann random access machine, then f is computable in polynomial time $p(n) = p_2(p_1(n))$ on a Turing machine. For example, if $p_1(n) = n^3$ and $p_2(n) = n^2$ then $p(n) = n^6$.
- For problems involving say enormous graphs, we may need sublinear time; for other problems we may need linear or near linear times. But as an abstraction, we are saying that a polynomial time algorithm is "efficient".

Why we should be less accepting of the extended Church-Turing thesis

While we are very confidant about the Church-Turing thesis (for defining "computable"), there are various reasons to be a little more skeptical about the extended Chruch-Turing thesis.

- An algorithm running in a polynomial time bound like n^{100} is not an efficient algorithm.
- An algorithm running in an exponential time bound like $(1 + \frac{1}{1000})^n$ is an efficient algorithm for reasonably (but not too) large input lengths. Note: $(1 + \frac{1}{k})^k \to e \approx 2.72$
- While we can simulate classical computers (i.e. von Neumann machines) in polynomial time, we do not know how to simulate non classical computers (e.g., quantum computers) in polynomial time.
- Factoring is an example of a problem that can be computed in polynomial time by a quantum computer whereas we do not believe factoring is polynomial time computable on a classical computer. So it is possible that we will have to change of definition of "efficiently computable" to be polynomial time on a quantum computer.

So should we accept the extended Church-Turing thesis?

We can accept the extended Church-Turing thesis, arguing as follows:

- Polynomial time computable functions usually have reasonably small asymptotic polynomial time bounds; that is, $n, n \log n, n^2, n^3$‘. There are some exceptions (like n^6, but generally speaking we don't usually encounter polynomial time bounds asymptotically bigger than n^3.
- The robustness of polynomial time (in terms of being closed under composition is not sensitive to the precise model of computing and definition of a time step. This enables us to define our concepts in terms of Turing machines (once we restrict outselves to classical computer models). Linear functions are also closed under composition but linear time computation is very model dependent.
- While non-classical models may contradict the thesis, **so far** we do not have non-classical computers (e.g., quantum computers that go beyond a small number of quantum bits) that are practical in a commercial sense.

But what if quantum computers become practical?

Lets assume the quantum computers or other non-classical computers become practical. We are about to discuss the P vs NP issue and the $P \neq NP$ conjecture, the central question in complexity theory.

This conjecture is formulated with respect to the extended Turing thesis. That is, we are accepting the definition that "efficiently computable" means polynomial time computable by a Turing machine. Will everything about this question and conjecture become useless if we someday have available more powerful non-classical (e.g., quantum) computers?

But what if quantum computers become practical?

Lets assume the quantum computers or other non-classical computers become practical. We are about to discuss the P vs NP issue and the $P \neq NP$ conjecture, the central question in complexity theory.

This conjecture is formulated with respect to the extended Turing thesis. That is, we are accepting the definition that "efficiently computable" means polynomial time computable by a Turing machine. Will everything about this question and conjecture become useless if we someday have available more powerful non-classical (e.g., quantum) computers?

No, the theory we will be developing can be reformulated in terms of a new computational model. We will have new functions (like factoring integers) which will now become efficiently computable (assuming they were not efficiently computable classically). But still there will be an analogous complexity theory based on the (for now hypothetical) new computational model. Moroever, our current belief is that there are problems in the class NP that are not computable in polynomial time on a quantum computer.

Polynomial time computable decision problems

We will now restrict attention to decision problems; that is $f : \Sigma^* \to \{YES, NO\}$. Σ is a finite alphabet and Σ^* is the set of all strings over Σ. We can also identify $\{YES, NO\}$ with say $\{1, 0\}$.

Equivalently, we are considering languages $L \subseteq \Sigma^*$.

The class of languages (decision problems) P is defined as the set of languages L that are decideable in polynomial time on a Turing machine; that is the languages that are "efficiently decideable".

In what follows, I will assume we have some agreed upon way that we represent graphs $G = (V, E)$ as strings over some finite alphabet Σ. Without refering to the representation, let $L_{connected} = \{G = (V, E) | G$ is a connected graph$\}$.

It is not difficult to show that $L_{connected}$ is in the class P. (For example, we can use breadth first search.)

A language "probably not" in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in $V\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The "well-known" *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

A language "probably not" in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in $V\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The "well-known" *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

But suppose that a given graph G has Hamiltonian cycle. How can I convince you that G has such a cycle

A language "probably not" in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in $V\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The "well-known" *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

But suppose that a given graph G has Hamiltonian cycle. How can I convince you that G has such a cycle

I can simply show you a Hamiltonian cycle C (assuming I know C) and you can easily and efficiently verify that C is indeed a HC. That is, I can prove to you that G has a HC.

A language "probably not" in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in $V\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The "well-known" *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

But suppose that a given graph G has Hamiltonian cycle. How can I convince you that G has such a cycle

I can simply show you a Hamiltonian cycle C (assuming I know C) and you can easily and efficiently verify that C is indeed a HC. That is, I can prove to you that G has a HC.

But can I efficiently prove to you that G does *not* have a HC?

A language "probably not" in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in $V\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The "well-known" *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

But suppose that a given graph G has Hamiltonian cycle. How can I convince you that G has such a cycle

I can simply show you a Hamiltonian cycle C (assuming I know C) and you can easily and efficiently verify that C is indeed a HC. That is, I can prove to you that G has a HC.

But can I efficiently prove to you that G does *not* have a HC?

""Probably not"

NP: the class of languages which are "efficiently verifiable"

Using the HC problem as an example, lets define what it means to be efficiently verifiable.

Let L be a language (like L_{HC}) that satisfies the following conditions: There is a polynomial time decideable relation $R(x, y)$ and a polynomial p such that for every x, $x \in L$ if and only if there exists a y with $|y| \leq p(|x|)$ and $R(x, y) = TRUE$.

$R(x, y)$ is a verification relation (or predicate) and y is called a certificate that verifies x being in L.

The class NP is the class of languages (decision problems) that have such a verification relation and certificate.

For example HC is in NP. Namely, given a representation x of a graph $G = (V, E)$, a certificate y is an encoding of a sequence of vertices specifying a Hamiltonian cycle C . $R(x, y)$ checks the conditions for $y = C$ being a simple cycle containing all the nodes in V .

The million $ question: Is $P \neq NP$

This is literally (and not just figuratively) a million $ question for someone who solves the question. In fact, it is worth much more that just one million $ for a proof that either $P = NP$ or a proof that $P \neq NP$.

Cook defined the concept of NP-completeness and gave a couple of examples of such problems, namely SAT and $CLIQUE$, problems in NP that are believed to not be in P.

We wil define NP-completeness and the evidence for the conjecture that $P \neq NP$.

The important consequence of NP completeness is that if *any* NP decision problem turns out to be in P, then $P = NP$. Since we strongly believe $P \neq NP$, this means that we strongly believe that no NP complete problem can be in P.

NP: the class of languages which are "efficiently verifiable"

Using the HC problem as an example, lets define what it means to be efficiently verifiable.

Let L be a language (like L_{HC}) that saitisfies the following conditions: There is a polynomial time decidable relation $R(x, y)$ and a polynomial p such that for every x, $x \in L$ if and only if there exists a y with $|y| \leq p(|x|)$ and $R(x, y) = TRUE$.

$R(x, y)$ is a *verification* relation (or predicate) and y is called a *certificate* with respect to R that verifies x being in L.

Definition: The class NP is the class of languages (decision problems) that have such a verification relation and certificate.

For example L_{HC} is in NP. Namely, given a representation x of a graph $G = (V, E)$, a certificate y is an encoding of a sequence of vertices specifying a Hamiltonian cycle C. $R(x, y)$ checks the conditions for C being a simple cycle containing all the nodes in V.

Many many decision problems are in the class *NP*

First we will note that the class P (decision problems decideable in polynomial time) is a subset of NP; that is, $P \subseteq NP$. Is this obvious?

Many many decision problems are in the class NP

First we will note that the class P (decision problems decideable in polynomial time) is a subset of NP; that is, $P \subseteq NP$. Is this obvious?

Consider a language L (like $L_{connected}$) that is decideable in polynomial time. Then in the definition of NP, we can let let $R(x, y)$ be the relation that is $TRUE$ iff $x \in L$ ignoring y and $R(x, y)$ is polynomial time since we can decide if $x \in L$ in polynomial time by the assumption that $L \in P$.

Many many decision problems are in the class NP

First we will note that the class P (decision problems decideable in polynomial time) is a subset of NP; that is, $P \subseteq NP$. Is this obvious?

Consider a language L (like $L_{connected}$) that is decideable in polynomial time. Then in the definition of NP, we can let let $R(x, y)$ be the relation that is $TRUE$ iff $x \in L$ ignoring y and $R(x, y)$ is polynomial time since we can decide if $x \in L$ in polynomial time by the assumption that $L \in P$.

In saying $P \subseteq NP$, we have left open the possibility that $P = NP$. However, the widely believed assumption (conjecture) is that $P \neq NP$. This question (conjecture) was implicitly asked by (for example) Gauss (early 1800's), von Neumann, Gödel (1950's) , Cobham, and Edmonds (1960s). The conjecture was formalized by Cook in 1971 (indpendently by Levin in the FSU but his work was not known until about 1973).

More specifically Cook defined the concept of NP-completeness and gave a couple of examples of such problems, namely SAT and $CLIQUE$, problems in NP that are believed to *not* be in P. We wil define NP-completeness and the evidence for the conjecture that $P \neq NP$.

End of slides for week 7

I am appending a few more slides for those who want to see the definition of NP-completeness.

Efficient reductions

At the heart of NP completeness and more generally algorithm analysis is the concept of (efficient) reduction of problems. When we say that problem A "efficiently" reduces to problem B, we can conclude that an efficient algorithm for B will result in an efficient algorithm for A (and equivalently, the contrapositive states that A not efficiently computable implies that B is not efficiently computable).

There are different definitions for what we mean by an efficient reduction and the precise definition matters in terms of what we want to conclude from the reduction.

One major distinction is between a very general type of reduction (which we will just call *poly time reduction* (i.e., the poly time version of Turing reduction) and the more restricted reduction which we will call *poly time transformation* (i.e., the polynomial time version of a Turing computable transformation).

Two types of reductions continued

The general version of reduction $A \leq_T^{poly} B$ means that there is a poly time algorithm ALG that can call a subroutine for B (as often as it likes) and ALG computes A. Here we count each call to the subroutine as 1 step. It is not difficult to see that if $A \leq_T^{poly} B$ and B is computable in polynomial time, then A is computable in polynomial time.

The $\leq_T^{poly}$ reduction is what Cook used in his seminal 1971 paper.

The more restricted transformation (which we call a polynomial time transformation) $A \leq_{trans}^{poly} B$ means that there is a polynomial time function h (transforming an input instance of A to an input instance of B) such that $x \in A$ if and only if $h(x) \in B$. Note that $|h(x)| \leq p(|x|)$ for some polynomial p. Why?

Two types of reductions continued

The general version of reduction $A \leq_T^{poly} B$ means that there is a poly time algorithm ALG that can call a subroutine for B (as often as it likes) and ALG computes A. Here we count each call to the subroutine as 1 step. It is not difficult to see that if $A \leq_T^{poly} B$ and B is computable in polynomial time, then A is computable in polynomial time.

The $\leq_T^{poly}$ reduction is what Cook used in his seminal 1971 paper.

The more restricted transformation (which we call a polynomial time transformation) $A \leq_{trans}^{poly} B$ means that there is a polynomial time function h (transforming an input instance of A to an input instance of B) such that $x \in A$ if and only if $h(x) \in B$. Note that $|h(x)| \leq p(|x|)$ for some polynomial p. Why?
It is easy to see that $A \leq_{trans}^{poly} B$ and $B \in P$ implies $A \in P$.

Following Cook's paper, Karp provided a list of 21 combinatorial and graph theoretical problems that are NP complete. Karp used the more restrictive $\leq_{trans}^{poly}$. If you like names associated with these reductions then we can denote $\leq_T^{poly}$ as $\leq_{Cook}$ and $\leq_{trans}^{poly}$ as $\leq_{Karp}$.

NP-completeness wrt reductions $\leq_T^{poly}$ and $\leq_{trans}^{poly}$

Let's first explicitly give the definition NP-complete.

Definition: A language (or decision problem) L is NP complete if

1. $L \in NP$.
2. L is NP-hard *with respect to some polynomial time reduction*, for example with respect to either $\leq_T^{poly}$, or $\leq_{trans}$ *poly*. That is, if we are using $\leq_{trans}^{poly}$, then L is NP-hard if for every $A \in NP$, there is a polynomial time computable function h such that $w \in A$ if and only if $h(w) \in L$.