Great Ideas in Computing

University of Toronto CSC196
Fall 2022

Week 3: September 26-September 30 (2022)

Week 3 slides

Announcements:

- This week there was no tutorial and the class is being held on Wednesday and Friday.
- Next week, the class will be held on Monday and Friday and the tutorial on Wednesday. After next week we return to the "normal" Monday and Wednesday schedule for classes and Fridays for tutorials. The dates for the guest presentations is Rahul Krishman (ML and medical diagnosis) Oct 7, David Lindell (computational vision) October 12, Daniel Wigdor (HCI) November 4, Fan Long (block chain technology) November 14.
- Assignment A1 is due Friday, October 7 at 8AM. Due to the rearrangement of classes, this doesn't give us much time so I am making this a reasonable short and I think not difficult assignment. I will post the remaining questions today/tonight.

This weeks agenda

- We will quickly review and conclude our discussion of the data structures for the dictionary data type that we discussed last week. That is, unordered lists, oredered lists, linked lists, and search trees. In particular, we can consider balanced binary search trees. (We will be using the white board for illustration.)
- Search trees will lead us to an introduction of the mathematical concept and terminology of graphs (also called networks) and directed graphs/networks.
- Our final data structure for a dictionary is a hash table. Hash tables natually lead us to start discussing the importance of randomization in computation.

Quick review of last weeks data structures for a (dynamic) dictionary

As we saw there are tradeoffs between different data structures for the Dictionary data type. (Moreover, tradeoffs are inherent in algorithms and more generally in life.)

The choice of any particular data structure or algorithm will usually depend on the intended application or class of applications (to the extent that is known). In choosing a data structure what operations are being done more often than others is arguably the main consideration.

Are we mainly searching for records with relatively few updates or are we frequently updating the data base?

For a dynamic dictionary of n items stored in an array N words, we can search an ordered list in "time" $O(\log_2 n)$ whereas searching in an unordered list requires $O(n)$ in the worst case. What is "average time" and what is "time"? But updating in an inserting a new item in an unordered list (if we know n) is constant time whereas inserting a new item into a sorted list requires $O(n)$ time.

Search trees as a special case of graphs/networks and terminology

Let's again consider the search tree data structure which I will again illutrate on the white board.

In discussing search trees, we will use the terminology of graph theory so as to make definitions precise. Lots of notation and definitions but hopefully the illustrations will make things clear. A search tree is labelled rooted tree which is a special case of a directed graph.

A directed graph (also called a directed network) $G = (V, E)$ consists of a set of nodes (also called vertices) V and a set of edges E (also called links) where each edge is an ordered pair (u, v) of nodes. We are drawing a representation of an edge on the board. **Note:** We are assuming simple directed graphs where $u \neq v$; i.e., no self loops.

A graph (also called an undirected graph) is almost formally the same but now edges are unordered pairs.
Note: The beginning chapters of the text by Easley and Kleinberg gives lots of examples and applications of graphs/networks.

A few more graph theoretic definitions

The in-degree of a node v in G is $|\{(u, v) \in E\}|$; that is, the number of edges into v. The out-degree of a node v is $|\{(v, z) \in E|\}$; that is the number of edges leaving v.

A directed path π from v to z in a directed graph is a sequence of edges $(u_0, u_1), (u_1, u_2), \ldots (u_{\ell-1}, u_\ell)$ with $v = u_0$ and $z = u_\ell$. The path length of π is ℓ. We are only considering simple paths such that $u_i \neq u_j$ for $i \neq j$.

A directed tree is a special case of a directed graph with the folllowing properties:

- There is a unique node (called the root having in-degree 0 ; lets call the root r.
- For every node $v \neq r$, there is a unique path from r to v.

It follows that every node $v \neq r$ has in-degree 1.
The depth $depth(v)$ of a node v in a directed tree is the length of the unique path from r to v. Note: We can slightly abuse terminology and say that r is at depth 0.

A leaf in a directed tree is a node having out-degree 0.

Search trees as labelled directed trees; binary search trees

In a graph or directed graph we may want to label the nodes and/or edges.

A search tree is a directed tree where the nodes are labelled by record identifiers or (also called "keys"). We could then use pointers to the record corresponding to the key (or if the records don't contain much information, then store the entire record in a node).
Note: Alternatively, we might only store the keys/records in the leaves of the tree.

For simplicity, we will only consider binary search trees where every node has out-degree at most 2. (In a strict binary tree, every non-leaf node has out-degree exactly 2.)

In a binary search tree, the left (resp. right) "subtree" of a node v consists of all the nodes whose identifiers (i.e. labels) are less than (resp. greater than) the identifier of v.

Final comments on binary search trees.

A perfectly balanced binary search tree is one in which depth of the tree (i.e. the maximum path length to any node) is $\lfloor \log_2 n \rfloor n$. For our purposes, it is OK if the depth is "close" to $\log_2 n$.

In a balanced search tree with n keys, a search can be done in time $O(\log_2 n)$ time. I hope this is easy to see as this is like binary search. And it suffices to be "nearly balanced".

What is not so easy to see is that a balanced binary search tree can be rebalanced after an update or an insertion/deletion in $O(\log_2 n)$ time. So balanced search trees seem like a good compromise for dynamic dictionaries.

As pointed out in class, a linked list can be viewed as a special case of an oerdered search tree where the root contains the smallest key and all nodes (except the leaves) have out-degree 1.

I was asked if there is any reason to use an sorted list (in any array) rather than a binary search tree?.

Final comments on binary search trees.

A perfectly balanced binary search tree is one in which depth of the tree (i.e. the maximum path length to any node) is $\lfloor \log_2 n \rfloor n$. For our purposes, it is OK if the depth is "close" to $\log_2 n$.

In a balanced search tree with n keys, a search can be done in time $O(\log_2 n)$ time. I hope this is easy to see as this is like binary search. And it suffices to be "nearly balanced".

What is not so easy to see is that a balanced binary search tree can be rebalanced after an update or an insertion/deletion in $O(\log_2 n)$ time. So balanced search trees seem like a good compromise for dynamic dictionaries.

As pointed out in class, a linked list can be viewed as a special case of an oerdered search tree where the root contains the smallest key and all nodes (except the leaves) have out-degree 1.

I was asked if there is any reason to use an sorted list (in any array) rather than a binary search tree?. If we had a static dictionary, a sorted list would be faster by some constant factor and would take up less memory by a

End of Wednesday, September 28 class

http://www.cs.toronto.edu/~bor/303s20 is the home page for the 2020 spring version of CSC303 (Social and Information Networks). There you will find a link to the course contents. Take a look at the slides for week 1 for all the graph concepts you will need for this course and probably for most future courses. .

You will also find many examples of applications of graphs and directed graphs. As I said in class today, graphs are used throughout CS and many other disciplines. Getting familiar with these concepts is important.

We will continue on Friday with a discsussion of hash tables.

Start of Friday, September 30 class

Announcements:

- I have added two questions to Assignment 1 (A1) and the assignment is now complete. I can answer questions on the Assignment today and Monday and Vignesh can also take questions on Wednesday. Please be sure to submit your aassignment on time.
- Next week the class meets Monday and Friday with the tutorial on Wednesday. On Friday, Professor Rahul Krishnan visits.

Todays agenda

- We will discuss hash tables as our last example of a way to implement a dictionary. This discussion will naturally involve randomness and probability. As I mentioned last class, the two mathematical fields that you will often encounter in CS are graph theory and probability.
- Then we will start a new topic, namely we want to consider the fundamental question "What does it mean to be computatble". This to me is one of the greatest of the great ideas even though (like other great ideas) we may be quite unaware of the importance of having a precise definition for what is and what is not computable.

A hash table: One more way to implement a dictionary

We have a hash function $h : I \rightarrow M$ where $I = \{ID_1, \ldots ID_N\}$ is the set of all possible integer identifiers and $M = \{A[0], \ldots, A[m-1]\}$ is a small set of memory locations

That is, we are going to hash each of the $N = |I|$ possible items to a small set of $m = |M|$ memory locations.

Here we can have $N >> n$ where n is the actual number of items we are storing.
What is a suitable hash function h?

A hash table: One more way to implement a dictionary

We have a hash function $h : I \rightarrow M$ where $I = \{ID_1, \ldots ID_N\}$ is the set of all possible integer identifiers and $M = \{A[0], \ldots, A[m-1]\}$ is a small set of memory locations

That is, we are going to hash each of the $N = |I|$ possible items to a small set of $m = |M|$ memory locations.

Here we can have $N >> n$ where n is the actual number of items we are storing.
What is a suitable hash function h?
One possiblility is $h(ID) = (a \cdot ID + b)(mod\ p)(mod\ m)$ where p is a large prime.

A hash table

Ignoring conflicts in the hash table, can search in constant time for a particular item

A hash table

Ignoring conflicts in the hash table, can search in constant time for a particular item

Need to deal with conflicts; i,.e multiple items hashing to the same place in the hash table. When there is a conflict, one possibility is to use a pointer to a linked list containing the IDs that have been matched to the same place in the hash table.

A hash table

Ignoring conflicts in the hash table, can search in constant time for a particular item

Need to deal with conflicts; i,.e multiple items hashing to the same place in the hash table. When there is a conflict, one possibility is to use a pointer to a linked list containing the IDs that have been matched to the same place in the hash table.

Note: When we draw random numbers in the execution of an algorithm, we are not drawing truly random numbers. The generation of pseudo random numbers and pseudo random functions is an interesting and substantial topic, one related to complexity theory and cryptography, two of our future topics.

Hash tables lead us to introduce the use of probability, pseudo random numbers and functions.

A hash table

Ignoring conflicts in the hash table, can search in constant time for a particular item

Need to deal with conflicts; i,.e multiple items hashing to the same place in the hash table. When there is a conflict, one possibility is to use a pointer to a linked list containing the IDs that have been matched to the same place in the hash table.

Note: When we draw random numbers in the execution of an algorithm, we are not drawing truly random numbers. The generation of pseudo random numbers and pseudo random functions is an interesting and substantial topic, one related to complexity theory and cryptography, two of our future topics.

Hash tables lead us to introduce the use of probability, pseudo random numbers and functions.

Conceptually, we think of a hash funcftion as randomly placing an item in a memory location. Think of the item as a ball and memory as a collection of bins.

The birthday paradox

The birthday paradox: In probability theory, the birthday problem or birthday paradox concerns the probability that, in a set of n randomly chosen people, some pair of them will have the same birthday. By the pigeonhole principle, the probability reaches 100% when the number of people reaches 367 (since there are only 366 possible birthdays, including February 29). However, 99.9% probability is reached with just 70 people, and 50% probability with 23 people. These conclusions are based on the assumption that each day of the year (excluding February 29) is equally probable for a birthday.

Our intuitive view of hashing as a balls and bins trial is good at some level; but we do need to remember that we are only approximately true randomness.

What can't a computer do?

When we see the rather spectacular ways in which computer algorithms can perform, it is natural to ask whether or not there is anything that eventually we cannot do by computers.

Watching this evolution of computation and communication over say the last 80 years (since the earliest general purpose computers) and, in particular some of the most recent applications of machine learning, one can be forgiven for perhaps believing that there are no ultimate limitations.

But if we are going to ask about the limitations of computation in a precise way, well then we will need a precise mathematical framework.

This will lead is to the seminal 1930s work of Alan Turing (and independently Alonzo Church). To appreciate the seminal (and I would even say surprising) nature of this work, we consider Hilbert's 10th problem.

Computer Science as a mathematical science

David Hilbert was one of the great mathematicians of the late 19th and early 20th centuries. He asked the following question in **1900** known as Hilbert's 10^{th} problem:

"Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers"

Here is a more familiar way to ask this question:

Given a polynomial $P(x_1, \ldots, x_n)$ with integer coefficients in many variables, decide if P has an integer root. That is, do there exist integers $i_1, \ldots, i_n$ such that $P(i_1, \ldots, i_n) = 0$?

As an example, $P(x) = x - 2$ clearly has an integer root whereas $P(x) = x^2 - 2$ does not have an integer (or rational) root.

What is computable? What is decidable?

Hilbert's question was essentially to ask if there is an algorithm that could decide whether or not a given multivariate polynomial has an integer root. Hilbert didn't mention the words "algorithm" or "computer" but he did articulate the need to solve the problem in a finite number of "steps".

Hilbert believed there was such a decision procedure but did not formalize what it meant to say that a problem solution is *computable*.

Terminology: If the problem is a decision problem (i.e., where the solution is to output YES or NO) then we usually say decidable rather than computable.

Following a series of intermediate results over 21 years, in 1970 Matiyasevich gave the first proof that Hilbert's 10th problem was undecidable (in a precise sense we will next discuss).

Note: The problem is decidable for polynomials in one variable.

A precise definition for the meaning of "decidable"

We have studied the von Neumann model as a model of computation but we never gave a precise definition but more or less relied on our prior knowledge of how we think computers work. And we didn't give a definition for what is an *algorithm*.

Computers are continually getting faster and have larger memories so must our concept of what is computable also be constantly changing? Could Hilbert's problem become decidable tomorrow?

We also briefly touched upon the complexity of operations with respect to the data structures for the dictionary data type. Must the complexity of operations and the complexity of algorithms also change constantly?

This raises a fundamental question: Is there an ultimate precise model of computation with respect to which we would then have a precise meaning of a computable function? Or must we continually be changing our understanding of what is and what is not computable?

A precise definition for the meaning of computable (decidable) continued

High level models such as the von Neumann model provide a good intuition for what we have in mind when we say a function f is decidable. **But we really need a precise mathematical model if we want to prove mathematical results.**

Independently in 1936, Alonzo Church and Alan Turing published formal definitions for what it meant to be computable. These papers were very influential for the von Neumann model which comes about 10 years later.

Church's definition was based on a formalism in logic called the lambda calculus where one starts with some basic functions and then specifies ways to compose new functions from existing functions.

Alan Turing proposed a precise model of computation which we will only briefly describe. Turing also went on to show that these two very different models are provably equivalent in the sense that they result in the same set of computable functions.

But are there other models?

For a number of years other models were considered and all turn out to be equivalent (and sometimes weaker) than the Church-Turing models.

This led to the following Church-Turing hypothesis. Every plausible model of computation is equivalent to (or weaker than) Turing's very basic computational model. This is not to say that Turing machines are as easy to program or will lead to the same complexity analysis. **But the meaning of computable does not change.**

In particular, what about quantum computing?

But are there other models?

For a number of years other models were considered and all turn out to be equivalent (and sometimes weaker) than the Church-Turing models.

This led to the following Church-Turing hypothesis. Every plausible model of computation is equivalent to (or weaker than) Turing's very basic computational model. This is not to say that Turing machines are as easy to program or will lead to the same complexity analysis. **But the meaning of computable does not change.**

In particular, what about quantum computing? It could very well be that quantum computing will substantially change our sense of what is "efficiently computable" but it does not enlarge the meaning of "computable".

Note: The Church Turing hypothesis in NOT a theorem. It is an almost universally believed statement about the nature of digital computing. Could someday we come to believe that there are more inclusive models?

But are there other models?

For a number of years other models were considered and all turn out to be equivalent (and sometimes weaker) than the Church-Turing models.

This led to the following Church-Turing hypothesis. Every plausible model of computation is equivalent to (or weaker than) Turing's very basic computational model. This is not to say that Turing machines are as easy to program or will lead to the same complexity analysis. **But the meaning of computable does not change.**

In particular, what about quantum computing? It could very well be that quantum computing will substantially change our sense of what is "efficiently computable" but it does not enlarge the meaning of "computable".

Note: The Church Turing hypothesis in NOT a theorem. It is an almost universally believed statement about the nature of digital computing. Could someday we come to believe that there are more inclusive models? Yes but so far our experience leads us to believe that the hypotheis will continue to be (almost) universally accepted.

A pictorial representation of a Turing machine

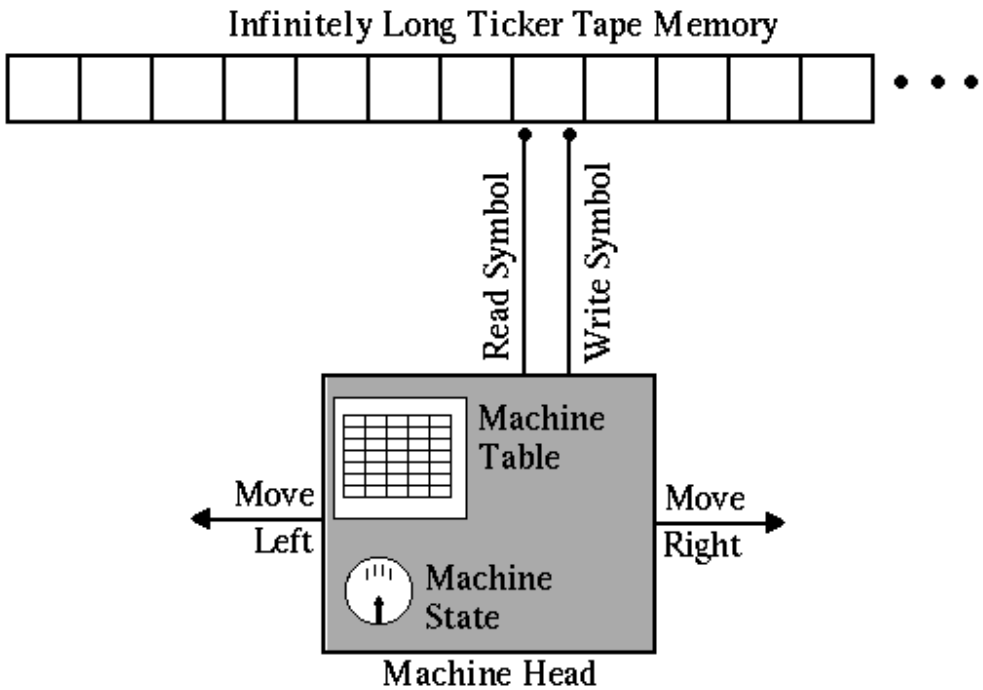

Figure: Figure taken from Michael Dawson "Understanding Cognitive Science"