CSE 2300 Discrete Structures
Developed by Jim Rutherfoord

CSE 2300 Course Syllabus

Module 1 – Importance of Discrete Structures
The purpose of this sub-module is to introduce you to the importance of discrete structures for computing. In particular, on completion of this sub-module, you will be able to

1. Explain the term "discrete" as used in this context
2. Explain why discrete structures are so important in computing.

Please read http://cnx.org/content/m14586/latest/?collection=col10768/latest by They Duy Bui.
You can also watch the file "11-01-00: What kinds of problems are solved in discrete math?" from http://www.oercommons.org/courses/discrete-mathematics/view. This is an introductory lecture by Shai Simonson, but you will need to make sure you have RealPlayer or some other tool to display videos. Clearly, it is an introduction to the course as taught by Dr. Simonson but it does provide some idea about what discrete mathematics is and why it is important for computing.

Finally, there is some more information on pages ix and x in Thomas VanDrunen's book "Discrete Structures and Functional Programming", which you can find at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.9659&rep=rep1&type=pdf .

Module 2 – Propositional Logic

Logic is the study of valid reasoning and, as such, is relevant to much more than just computing. It turns out that there are many different types of logic, of which we will consider two, namely propositional and first-order predicate calculus, in some detail in this module. However, there is another logic that is of some interest in computing, namely temporal logic, and we will very briefly cover temporal logic as well.

Moreover, since this course is about discrete structures in computing, we will also discuss a few examples of how logic is relevant in computing.

As said, propositional logic is the simplest logic. It is used to reason about propositions or sentences that are either true or false but not both. Not all sentences are propositions. For example, questions are sentences but are not true or false and therefore are not propositions.

In propositional logic, we form complex propositions from simpler propositions using the connectives ~ (NOT) , & (AND), v (OR) and → (IMPLIES, or IF .. THEN), and we determine the truth value of a complex proposition (i.e., whether it is true or false) from the truth value of the propositions that it is composed out of. Note that different authors will use different symbols for the connectives.

Video: http://youtu.be/-svsnPl7qcQ

Video class lecture: http://www.youtube.com/watch?v=-v3u1VGXc6M

Video Class Lecture Axioms: http://www.youtube.com/watch?v=9J-jFz9iJLM

Free textbook chapter link: https://cnx.org/contents/IdMjj0pQ@1.1:LhBnDMwS@1/Discrete-Structures-Logic

Module 3: Predicate Logic

Class video: http://www.youtube.com/watch?v=FBWS3RNsI7A

Video: http://youtu.be/YbNmPievBak

Link free textbook: https://cnx.org/contents/IdMjj0pQ@1.1:LhBnDMwS@1/Discrete-Structures-Logic

Module 4: Logic in Computing

Declarative logic: Normally, when you write a computer program, you not only instruct the machine what you want it to achieve, you also tell it how to achieve it. Thus, the following code fragment shows how to calculate the average of three numbers:

```
double average (int num1, int num2, int num3) {
      int sum = num1 + num2 + num3;
      double avg = sum/3;
      return avg;
}
```

Note that in the above, we have to specify not only what is to be computed; we also have to specify how it is to be computed. Clearly, this is a very simple example but, as you know, for large programs this gets more and more complicated, and we typically have to comment the code to remind ourselves and others what the code is meant to achieve.

An alternative style of programming is called "declarative programming", in which we tell the program what we want it to determine for us, but leave it up to the program itself, or rather the interpreter or compiler for the programming language in question, to determine how the result is computed.

We briefly discuss two examples of declarative programming languages, namely SQl and Prolog. Depending on the major you are in, you will already have come accross at least one of them or will do so in the near future.

PROLOG: Another way in which logic is used in computing is through a programming paradigm called logic programming. As you probably know, there are different approaches to creating programming languages, often referred to as programming paradigms. In this module, we will very briefly discuss programming paradigms. Programming paradigms are covered in great detail in CS3123 Programming Language Concepts.

There are four main programming paradigms, namely

- Procedural programming
- Object-oriented programming
- Functional programming
- Logic programming

In the procedural programming paradigm, a program is explicitly regarded as a sets of procedures. C is an example. In object-oriented programming, a program is seen as a collection of objects interacting with each other. Java and C++ are examples of object-oriented programming languages, although -in my opinion- poor examples. In the functional programming paradigm, a program is seen as a collection of functions in the sense in which this term is defined in mathematics and covered later in this course. Lisp is an example.
For our purposes, the programming paradigm that is more relevant is logic programming. Under this programming paradigm, programs are seen as sets of logical sentences, expressing facts and rules about some problem domain. Programming then becomes a matter of interrogating the program.
Perhaps the best known example of a programming language is Prolog. A Prolog consists of a series of facts and rules, expressed as Horn clauses. A Horn class is a universally quantified conditionals, in which there is only one statement in the consequence and the antecedent is a conjunction. The following is a very simple example of a prolog program

```
sheep(sam)
sheep(susie)
```

```
male(sam)
female(susie)
ram(X) :- sheep(X), male(X)
ewe(X) :- sheep(X), female(X)
```
The first four lines simply give some facts, while the last two code up the rules that a male sheep is a ram, and a female sheep is an ewe. Prolog uses capitals as variables. The last two prolog clauses can be rendered in the language of predicate logic that we have used as
$(\forall x)[(\text{sheep}(x) \ \& \ \text{male}(x)) \rightarrow \text{ram}(x)]$
$(\forall x)[(\text{sheep}(x) \ \& \ \text{female}(x)) \rightarrow \text{ewe}(x)]$
In order to interrogate a Prolog program, we simply ask the question we want the program to answer. Thus, if we want to know whether Sam is a ram, we simply type
```
?- ram(sam).
```
and the program will eventually answer "yes".
We can also ask the program to find any rams. We would do so by using a variable:
```
?- ram(X).
```
and the program will answer
X = sam.
One of the main problems with Prolog from a logical point of view is the way in which it deals with negation. Prolog uses "negation-as-failure". As your exercise for this sub-module, use the internet to find out what negation-as-failure means, and give the definition and the source, and the explanation why negation-as-failure is problematic from a logical point of view, in the associated drop box.

Module 5 – Program Correctness

A final appplication of logic in computing in general and computer science in particular is in the area of program correctness. A program is correct when it does what it is intended to do, and it is formally correct if it can be mathematically proven to be correct.

Normally, we use testing methods to determine program correctness. Suppose that you are asked to write a program that prompts a user for a number and then calculates and displays the square of that number. Then, once written, you will test the program by inputing a number of numbers and making sure that the number that is displayed is indeed the square of the number that you input. You may also see what happens when the user enters an illegal input, such as a string.

While this type of testing is acceptable for many applications, there are cases where you will want to be more certain that the program is correct. An example might be a routine that controls a nuclear power plant. For applications such as these, one would like to be able to mathematically prove that the program is correct.

Clearly, in order to be able to formally prove that a program is correct, you need a language in which to specify very precisely what you want the program to achieve, and computer scientists and software engineers have therefore develop a number of fomal specification languages, including Z (pronounced "zed", not "zee") and VDM (http://en.wikipedia.org/wiki/Specification_language). Most formal specification languages are based on formal logic.

Godel's Incompleteness Algorithm: Logicians are interested in proving results about the various logics that they have defined. Such results are often called "meta-logical" results, as they are results *about* the logic in question.

One particular interesting meta-logical result is Gödel's first incompleteness result. The theorem states that no consistent system of axioms whose theorems can be listed by an effective procedure is capable of proving all truths about the relations of the natural numbers (http://en.wikipedia.org/wiki/G%C3%B6del%27s_incompleteness_theorems). In other words, there is no computer program that can prove everything that is true about arithmetic.

The technique that Gödel developed to prove this result can also be used to prove that the halting problem is undecidable. The halting problem (http://en.wikipedia.org/wiki/Halting_problem) is essentially the problem of deciding, given a program and an input, whether the program will eventually halt on this input or will run for ever, and Turing proved that the halting problem is not decidable

Yet another was of reformulating the problem is in terms of predicate logic:

There is no computer program that, for any set of propositions Γ and a proposition φ, will be able to determine whether φ follows from Γ or not.

In fact, predicate logic is semi-decidable: There is a computer program that will stop and say "yes" if φ actually follows from Γ, but there is no program that will say "no" if it does not.

To illustrate the problem (and note this is an illustration and not a proof), consider the following proposition:

A person is Jewish if their mother is Jewish.

Assuming that we do not know directly whether Sam is Jewish or not, we could try to determine Sam's Jewishness by figuring out whether Sam's mother is Jewish. Again, assuming that we have no direct evidence to determine whether Sam' mother is Jewish, we could try to prove that the mother of Sam's mother is Jewish, and so on. You see the problem.

Clearly, we are probably still interested in creating a computer program to automate reasoning in predicate logic, often referred to as an automated theorem prover for first-order predicate calculus, but we also want to make sure that the program does not run forever in the case that the proposition we want to prove does not follow.

The solution to this problem lies in the use of heuristics, rules-of-thumb that give the right answer in most cases, but that are not guaranteed to give the right answer in all cases. The use of heuristics is prevalent in Artificial Intelligence, and one could argue that one of the reasons for the emergence of Artificial Intelligence is the fact that we are interested in finding solutions for problems for which we can prove that there are no solutions that are guaranteed to work, such as automated theorem provers for first order predicate calculus.

An example of a heuristic that we can use in our example above is to stop searching once we have applied the rule six times. In other words, once we cannot directly determine that the mother of the mother of the mother of the mother of the mother of the mother of Sam is Jewish or not, we simply stop searching and assume that she was not Jewish, and we conclude that we cannot prove that Sam is Jewish. You will see the problem: We would draw the wrong conclusion if there actually is information stretching back 10 generations along Sam's maternal lineage that the mother in question was Jewish.

Free Textbook chapter: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.9659&rep=rep1&type=pdf

Module 6 – Algorithms

Ted Video: https://youtu.be/6hfOvs8pY1k

What is an Algorithm: Perhaps the most fundamental concept in computing is the concept of an algorithm. An algorithm is a finite set of unamniguous and precise instructions that

- can be executed by a computer and that
- takes an input, does some computaton, and produces an output, and
- terminates.

It is an abstract computer program, if you will.

Clearly, the problem with this definition is what it means to be "executable". The concept of an algorithm is often explained by drawing an analogy with recipes (for example, a recipe to make rice-and-peas (http://www.foodnetwork.com/recipes/bobby-flay/jamaican-rice-and-peas-recipe/index.html). However, there are many well-written recipes in which the steps are not executable by every cook. For example, not all of us know what to do when we are told to "deglaze the pan with white wine".
Fortunately, in the context of computing, we can define what we mean by an executable step, and we discuss this in detail in the sub-module entitled "Algorithms".
Nor surprisingly, there are often many different algorithms to achieve a particular task. For example, there are at least four well-understood algorithms for sorting a list. The question therefore naturally arises whether we can compare different algorithms in some abstract, mathematical way, for example to determine whether one runs faster than an other. The answer is that we can and the branch of discrete mathematics that allows us to do so is called "Complexity Analysis". In complexity analysis we express the running time of an algorithm in terms of the size of the input. Thus, we can express the running time of a sorting algorithm in terms of the numbers of items in the list to be sorted. The sub-module "Complexity Analysis" provides more detail.
As we shall see as well, in calculating the complexity of an algorithm, we ignore a number of factors. As a result, complexity analysis is not necessarily the best way to compare two algorithms, and in some cases, we may want to turn to alternative ways of comparing algorithms. One option is to conduct a timing experiment. In a timing experiment, we implement the different algorithms in a programming language and then measure the time that each takes to run. As your term project for this course, you will conduct some timing experiments for different sorting algorithms, and compare the results from your timing experiments with the results from a complexity analysis.
Many of the concepts introduced in this module will be discussed in far greater detail in subsequent courses, including the Data Structures course and the Analysis of Algorithms course. In this module, we will merely scratch the surface, and we will for example only discuss iterative algorithms, and ignore recursive algorithms.

Term Algorithm: As said before, an algorithm is essentially an abstract computer program. As we said as well, often there are many different algorithms for achieving the same task, and we want to find some way to formally compare the different algorithms. This in turn means that we need to find some way to specify algorithms.

Since our algorithms take an input and produce an output, they are very similar to functions in the mathematical sense (and discuss in some detail below). After all, functions take an input and produce an output that is specific to that input. That is, functions always produce the same output on the same intput. Moreover, since we want out algorithms to be executable by a computer, one could say that an algorithm is a computable function. Mathematicians started to try to characterize what makes a function computable well before actual computers had been built. Thus, the earliest definition of computable functions was provided by Alan Turing in 1936 when he defined what he called a "Logical Computing Machine" and what later became known as a Turing machine (http://en.wikipedia.org/wiki/Turing_machine). Other definitions included lambda calculus, register machines and μ-recursive functions (see http://en.wikipedia.org/wiki/Computable_function for an overview and links to the various definitions that have been proposed).
The good thing about all these definitions is that they are equivalent. For example, all functions that are computable if you use Turing machines are also computable if you use lamba calculus, and vice versa, and the same applies for all other definitions. The bad thing is that it is very hard if not impossible to specify algorithms that do some real work in any of these formalisms, and we therefore need a more intuitive way of specifying algorithms, and fortunately there is.
In order to specify an algorithm, we need the following constructs:

- An assignment operator to assign a value to a variables and create an assignment statement (e.g., = in Java or C++);
- A series of operators to compare values (e.g, ==, <, >) and thus create tests;
- Some logical operators to combine tests into more complex tests (e.g., && and || in Java and C++);
- A conditional to branch depending on the outcome of a test (if-then-else);
- A way to make a sequence of statements;
- A way to repeat statements.

Free Textbook chapter: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.9659&rep=rep1&type=pdf

Module 7 – Complexity Analysis

Big O Complexity: As said, in many cases, there are different algorithms to achieve the same task, and for obvious reasons, we will want to use the best algorithm to create the computer program. Now, there are many ways to define "best". For example, one algorithm may be better than another because it is easier to understand, and hence to translate into a programming language, like Java or C++. Or an algorithm may be better because it uses less memory. However, in complexity analysis, we consider an algorithm better if it runs faster.

There are of course many factors that influence the speed of a computer program, other than just the algorithm that underlies the program and the size of the input to the algorithm. For example, the type of computer and in particular the speed of the processor in the computer have a big influence on the speed of the program, and there is a somewhat corny joke among computer scientists that the best way to speed up a program is to save it to a USB drive and wait for the next generation computer to come out.

However, since we are comparing algorithms, i.e. abstract computer programs, we can ignore most of these other factors and we only consider the size of the input, and we express the running time of an algorithm as a function of the size of the input. The notation that we use to express the complexity of an algorithm is O (big-Oh). Moreover, when we give the complexity of an algorithm, we ignore all terms other than the term that most determines the growth of the value. Thus, if we determine that an algorithm runs in `n.log(n) + 73` time units, where `n` is the size of the input, we state that the complexity of the algorithm is O(n.log(n)). The rationale is that, as `n` gets sufficiently large, the contribution of the other terms to the value of the function that determines the running time becomes negligible.

Link: http://www.youtube.com/playlist?list=PL2_aWCzGMAwI9HK8YPVBjElbLbI3ufctn

Determining Complexity Analysis: In the sub-module on algorithms, we saw that in order to specify an algorithm, we needed

- An assignment operator to assign a value to a variables and create an assignment statement (e.g., = in Java or C++);
- A series of operators to compare values (e.g, ==, <, >) and thus create tests;
- Some logical operators to combine tests into more complex tests (e.g., && and || in Java and C++);
- A conditional to branch depending on the outcome of a test (if-then-else);
- A way to make a sequence of statements;
- A way to repeat statements, for which we used iteration (e.g., for-loops).

We can use these constructs to determine the time-complexity of an algorithm by using the following rules:

- The complexity of an assignment, a test and a combination of test is constant, i.e. it is O(1).
- The complexity of an if-then-else statement is the maximum of the complexity of the then-part and the else-part. Thus,
  O(`if <some_test> { <then-part> } else { <else-part> }`) = MAX(O(`<then-part>`), O(`<else-part>`))
- The complexity of a sequence of statement is the maximum of the complexity of the statements in the sequence. Thus,
  O(`<statement-1>; <statement-2>; .... ; <statement-n>;`) = MAX(O(`<statement-1>`), O(`<statement-2>`), ... , O(`<statement-n>`))
- The complexity of a for loop is the complexity of the body of the loop times the number of times we go throught the loop. Thus,
  O(`for(i = 0; i < n; i++){<body>}`) = n * O(`<body>`)

Watich the videos "Time complexity analysis - How to calculate running time?" and "Time complexity analysis - some general rules" at http://www.youtube.com/playlist?list=PL2_aWCzGMAwI9HK8YPVBjElbLbI3ufctn for more details and some examples. As before, you will have to watch some adverts. Also, the video "Time complexity analysis - some general rules" mentions asymptotic notations and in particular theta. You can understand the material without knowing additional details about these concepts, but if you are interested watch the video "Time complexity analysis: asymptotic notations - big oh, theta ,omega" at the same site.

Video Complexity Analysis: http://www.youtube.com/playlist?list=PL2_aWCzGMAwI9HK8YPVBjElbLbI3ufctn

Module 8 – Sets

Free textbook chapter: http://cnx.org/content/m15772/latest/?collection=col10768/latest

Video on Sets: https://youtu.be/t3XdRbPNtdg?list=PLUpS0WwSvA3e7HtgzNHMivo0T8V0etX_Z

Paradox of Sets: The material referenced above covers what Duy Bui calls "naive set theory". There is a complication with naive set theory, which was first published by Bertrand Russell in 1901, although the problem was known by some mathematicians before. Russell constructed the set of all sets that are not a member of themselves, i.e.

$\{ x \mid x \notin x \}$

and then asked whether this set is a member of itself. In other words, is

$\{ x \mid x \notin x \} \in \{ x \mid x \notin x \}$?

A little thought will show the problem. If this set, let's call it R, is a member of itself, then it must satisfy the condition of not being a member of itself. In other words,

$R \in R \rightarrow R \notin R$

On the other hand, if R is not a member of itself, then it satisfies the condition of of being a member of R. In other words

$R \notin R \rightarrow R \in R$.

We therefore have a paradox that can only be resolved by complicating naive set theory. You can read more about Russell's paradox and how mathematicians have dealt with it at http://en.wikipedia.org/wiki/Russell%27s_paradox.

Link to Paradox: http://en.wikipedia.org/wiki/Russell%27s_paradox.

Relations: One of the topics covered in the material on relations is orders. You will also recall that creating an ordered list of items is central to the term project in that you were asked to implement two different sorting algorithms and compare their performance in a timinig experiment.

The question may arise why sorting is so important. The reason is that if you can sort a set of items, and you can access any position in the set in a constant time, the complexity of determining whether a given item is an element of that set goes down from O(n) to O(log(n)).

Clearly, if your set is not ordered (or if you cannot access any position in the set in a constant time), then the only way in which you can determine whether a given item is an element of the set is to compare it with the first element in the set. If they are identical, you are done; if they are not, you compare it with the next element in the

set, and so on. If the item is not an element of the set, then we will need to compare the item with every element in the set, and we therefore make n comparison where n is the number of elements in the set. If the item is an element of the set, then, on average, the item you are looking for will be somewhere half way down the list, and you will therefore have to make on average $1/2n$ comparisons. In other words, you can expect to have to make $1/2n$ comparisons. Since we ignore the constant, the complexity of this type of search, which is called "linear search" is $O(n)$.

However, if the set is ordered, and we can access any position in the set in a constant time, we start by comparing the item with the element in the middle of the set. If the item is identical to the element in the middle of the set, then we are done; if it is not, and the item is smaller than the one in the middle, we know that the item, if it is an element of the set, will be in the first half of the list; if it is larger, then it will be in the second half. So, if it is smaller, we repeat the process for the lower half of the list; if it is larger then we repeat the process for the larger half of the list. Since we keep dividing the list we are searching in half, and we can divide a list of size n in half $\log(n)$ times, and the complexity of the algorithm, which is called "binary search", is $O(\log(n))$.

Link: http://www.youtube.com/watch?v=wNVCJj642n4

Class Lecture: https://youtu.be/YlhJKoqzcdY

Module 9 – Relations

Free Textbook chapter: http://cnx.org/content/m15775/latest/?collection=col10768/latest

Class lecture: https://youtu.be/ikIyUks8HRI

Video on Relations: http://www.youtube.com/watch?v=q3Z7PiW8FNg&list=PL_D1rGgPr31PjDJPnnsyDJo1eWweVeq03

Video 2 on Relations: http://www.youtube.com/watch?v=h34hZ_hynzE

Video 3 on Relations: http://www.youtube.com/watch?v=hM_iObXeno0

Database and relations: In the module on logic, we encountered an example of declarative programming in SQL. You will recall that SQL is the most widely used language to interrogate databases.

There have been different models of building databases, but the model most widely used at the moment is the relational model, which was due to Edgar F. Cobb. In the relational model, a database is a set of relations in the sense in which we defined the term in this module, also called tables. You can find more details on the relational model at

http://en.wikipedia.org/wiki/Relational_model,

and the model will be discussed in much greater detail in the database course.

Database researchers have also created two formal languages for creating, interrogating and analyzing databases, namely relational calculus and relational algebra. You can find more details about relational algebra at

http://en.wikipedia.org/wiki/Relational_algebra

and about relational calculus at

http://en.wikipedia.org/wiki/Relational_calculus

You will see that relational calculus has two different flavors, namely tuple relational calculus http://en.wikipedia.org/wiki/Tuple_relational_calculus

and domain relational calculus (http://en.wikipedia.org/wiki/Domain_relational_calculus

Module 10 – Functions

Free Textbook chapter: http://cnx.org/content/m15776/latest/?collection=col10768/latest

Class lecture: https://youtu.be/kykmBB74-HQ

Functional Programming: Under the functional programming paradigm, a program consists of a set of functions in the mathematical sense of the word. The language ML that is used in Thomas VanDrunen's book is an example of a functional programming language. Another -widely used- functional programming language is LISP. LISP has a very simple syntax. Here is an example of a simple LISP program

```
(defun mult_through_add (a, b)
     (if (eq a 0)
          b
          a + mult_through_add(a, b - 1)
     )
)
```

`defun` defines a new function and `if` is an in-built function in LISP which takes three arguments. It evaluates the first argument and if it does not evaluate to false, it evaluates the second argument and otherwise it evaluates the third. In other words, it is the if-then-else operator. Clearly, the definition assumes that `b` is a positive number.

Once we have defined a function, we can either call it directly, or use it in the definition of another function.

Find out more details about LISP at

http://en.wikipedia.org/wiki/Lisp_%28programming_language%29

or

http://www.gigamonkeys.com/book/

and see an example of a larger LISP program at

http://www.csc.villanova.edu/~dmatusze/resources/lisp/lisp-example.html

Module 11 – Graphs

Class Lecture: https://youtu.be/vOomN71xYIg

Video on Graphs: http://www.youtube.com/watch?v=HmQR8Xy9DeM

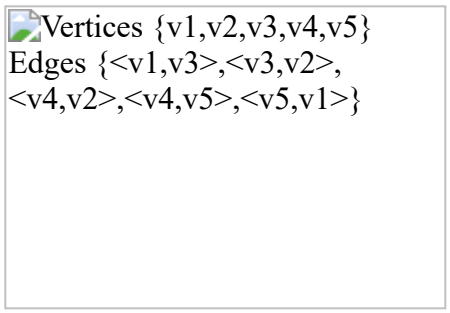
Video 2 on Graphs: http://www.youtube.com/watch?v=cOB85BQ8gX0

Video 3 on Graphs: http://www.youtube.com/watch?v=0t3i30T2NB0

Graphs as Networks: It will not come as a surprise that graphs are used heavily in order to analyze networks. In this module, we will introduce some examples and show how the various properties of graphs that were introduced in the previous sub-module and some of the algorithms that we will introduce can be used to determine properties of networks that are relevant to some applications.

Connectedness - Graphs, perhaps not surprisingly, can prove extremely useful as a tool to analyze computer networks. A good computer network is a connected graph. Given a set of network nodes and connections, which we can obviously analyze as a set of vertices and edges, a useful algorithm would be one that can quickly determine which the graph is connected.

There are different ways in which we can determine whether a graph is connected. One is through search: We can start at an arbitrary vertex, use a graph search algorithm, and count all the vertices we can reach. If the number of vertices we can reach is equal to the number of vertices in the graph, then the graph is connected. There are different graph search algorithms. Two that are particularly useful are breadth-first and depth-first search. Find out more about breadth first and depth first search, and use the dropbox "Graph Search" to show the order in which you would visit the vertices in the graph below, starting at vertex v4.



Adjancency graphs are also useful tool to determine whether a graph is connected. In order to show you how this works, recall that an adjacency matrix is esentially a 2-dimensional array both of whose dimensions are the same size. We will call this the size dimension of the array.

So, to determine whether a graph is connected from its adjacency graph M, we can use the following algorithm:

```
copy M into a new graph M';
changed = 1;
while (changed == 1) {
   changed = 0;
   for (i = 0; i < size dimension of M'; i++) {
        for(j = 0; j < size dimension of M'; j++) {
             if (M'[i,j] = 1) {
                  for(k = 0; k > size dimension of M', k++) {
                      if (M'[j,k] == 1 && M'[i,k] != 1)
                           M'[i,k] = 1;
                           changed = 1;
                      }
                  }
             }
        }
   }
}
```

If M' now completely consists of 1, G is connected.

Link Social Networks: http://en.wikipedia.org/wiki/Social_network

Link E/R Diagrams: http://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model

Link Semantic Networks: http://en.wikipedia.org/wiki/Semantic_network

Link Conceptual Graphs/Networks: http://en.wikipedia.org/wiki/Conceptual_graph

Module 12 – Trees

Link: http://www.saylor.org/site/wp-content/uploads/2011/09/CS202-Graphs1-Srini-Devadas.pdf

Link Definition: http://en.wikipedia.org/wiki/Tree_(data_structure)

Video: https://www.youtube.com/watch?v=58zSgcTj6ZQ&hd=1

Video 2: https://www.youtube.com/watch?v=HmQR8Xy9DeM&hd=1

Video 3: http://www.studyyaar.com/index.php/module/39-trees

Binary Search Trees

- In our discussion of sorting, we saw that if a list is sorted, and you can access any item in a constant time, you can use binary search, rather than linear search, and the complexity of determining whether an item occurs in a list goes down from O(n) to O(log(n)). However, if you cannot access any item in a constant time, then you cannot use binary search. Fortunately, binary search trees (BSTs) solve this problem.

Link: http://en.wikipedia.org/wiki/Binary_search_tree
Video: https://www.youtube.com/watch?v=pYT9F8_LFTM&hd=1
Video 2: https://www.youtube.com/watch?v=rVU_jXyHXqw&hd=1
Video 3: https://www.youtube.com/watch?v=pmsVitdSaVU&hd=1