**DATE: 9/4/2019**

Things to remember:

* Anything that is attached like (projects, attachment, submission of homework) it must be sent to [cs381projectsQC@gmail.com](mailto:cs381projectsQC@gmail.com) from student queens college qmail account.
* If student have any questions, please email in professor’s qmail account which is [rgoldberg@qc.cuny.edu](mailto:rgoldberg@qc.cuny.edu) .
* For Homework Submission:
  + In the subject line of email please include: GA <short description about homework>Date Assigned
* For Project Submission:
  + in subject line of email please include: GA<short description of project> Date Due.
* If you cannot find any email from professor, please make sure to check your junk/clutter/spam/trash/delete folders.
* Make sure to check your sent folder after sending any homework or project submission. It’s student responsibility to check if the emails are delivered or not.
* All emails should be sent through student qmail account.
* There are tentative dates of exam which may or may not happen in the given dates. Exact dates will be provided to student, at least a week before the exam.
* **Genetic Algorithm:** is a scientific method for searching algorithm that work like human genetic like DNA, genetic growth etc.

“√” 2 where “√” is symbolic mean

**Time Complexity:**

Space, time

bit 20 – 0/1

addressing byte – 23 – 0 to 7

kilobyte - 210

Megabyte - 220

Gigabyte – 230

Terabyte – 240

Petabyte – 240

* Genetic Algorithms are not algorithms that are used to solve problems in genetics or biology but rather the opposite, biology inspired algorithms are used to solve problems in computer science.
* **LISP** is an interpreted language, like Python, JavaScript.
* Examples of algorithms that were inspired by ideas in nature are Ant Colony Optimization, which involves finding optimal solutions in routing information similar to how ants find optimal routes to food sources. And Swarm Intelligence, which is used in AI to simulate populations that interact with their environment or each other in a way similar to other animals like ants, and birds.
* **Numerical Analysis**: According to Wikipedia, it is study of algorithms that use numerical approximation for the problems of mathematical analysis.
* What falls under the term Genetics in Genetic Algorithms?
  + Not limited to just Genetics but to other branches of science as well as math such as numerical analysis.
  + With time genetics became too broad of a topic and had to be narrowed down.
* Genetic Algorithms was about creating AI, Artificial Intelligence and exploring A Life, Artificial Life. The purpose of the algorithm was not to solve some big underlying problem but to explore the similarities between the human brain and a computer. John McCarthy was the one who created artificial intelligence as well as creating the Lisp programming language that was to be used in association with developments in artificial intelligence. McCarthy from MIT introduced artificial intelligence and LISP (Symbolic Processing List Processor) in mid-1964.
* Comparisons between the human brain and computers were the driving force for the creation of genetic algorithms and the creation of AI. With human DNA having something that is similar to a number system in base 4 it can be compared to computers that rely on binary number systems in base 2. Human DNA is made up of ACGTbase4 and computer “DNA” is made up of BinaryBase2.
* ELIZA was an early natural language processing computer program created from 1964 to 1966 at the MIT Artificial Intelligence Laboratory by Weizenbaum to attempt the Turing Test. The Turing test was created by Alan Turing to test where a machine can exhibit intelligence that is comparable to that of another human by having someone ask questions to both the machine as well as another human and if they are not able to tell the machine from the human, then the machine passes the test.
* Unknown to Church's work, [Alan Turing](https://en.wikipedia.org/wiki/Alan_Turing) created Turing machines. Given an encoding of the natural numbers as sequences of symbols, a function on the natural numbers is called [Turing computable](https://en.wikipedia.org/wiki/Computable_function) if some Turing machine computes the corresponding function on encoded natural numbers.
* Along with Alonzo Church Turing created the Church-Turing thesis. Computability depends on countability. If you can count it then you can compute it.
* Alonzo Church created a method for defining functions called the λ-calculus. Within λ-calculus, he defined an encoding of the natural numbers called the Church numerals
* Math is about consistency. Consistency does not necessarily mean truth. An example is in math false implies false is a truthful statement but both statements remain false themselves.
* **Transcendental numbers**- it is a real number or complex number that is not an algebraic number. Example are: pi, e.
* Georg Cantor established set theory and that real numbers are more numerous than **natural numbers**. If one were to store any natural number in bits you would need a finite amount of them while you would need an infinite amount of bits to store real number such as the **transcendental numbers** pi and e.
* George Cantor had done research in the 1870’s to 1890s Revolutionizing Mathematics and removing philosophical notions from the field.
* **Computability** also has to do with space and time can your algorithm fit on a machine?
* As time went on storage became less of an issue and time became the main focus. How long does it take for a solution to be reached and will algorithm eventually stop and receive an answer (effective computation).
* Juris Hartmanis is a computer scientist and computational theorist who received Turing award in 1993 and wanted to argue that computability depends on *practically* do you have enough time and space resources even if it won’t go through an infinite loop.
* Finite memory machine is a finite state machine which is a computation model that can used to simulate sequential logic, or it is to represent and control execution flow.
* **Natural vs Real numbers** (encoding not storage)
  + All-natural numbers can be represented in binary using log(N) number of bits, finite
  + Real numbers cannot be represented, like pi, natural log, log, etc., infinite
  + The reason of showing this was to highlight the Church-Turing thesis.

**Keywords:** Genetic Algorithm, Time complexity, LISP, Numerical Analysis, Transcendental numbers, computability, natural number, real number

**DATE: Sept. 5, 2019**

(**II**) Review on the first homework**:** To define and contrast the six related fields.

* **Genetic Algorithm**
* The genetic/biological inspired approach to produce output, the means to provide the solution for the problem.
* Data is stored in Array.

Note**:** **Heuristic** – “Rule of Thumbs”

* **Genetic Programming**
* It is the cousin of Genetic Algorithm that provides you the code [or “the strategies”] to get the answer.
* Data is stored in Trees or Graphs.

(Cont. to ***Genetic Programming***)

* UML, Unified Modeling Language, sees the flow chart

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Diagram of visualization of logic  code level of Design Process  The diagram is either  a Tree or a Graph. |  |

Can a Genetic Algorithm go into an infinite loop?

* First, let’s discuss about the term **infinite loops**. Are they bad? If yes, why are they bad?
* First, it never gives an answer. Infinite loops are not allowed in algorithms because you do not get an answer. And in generic term, output is the means of receiving an answer. The problem we are trying to tackle is, we have a problem to solve, and we want a solution. Solution means getting an answer. But if the code runs into an infinite loop, we don’t have an answer.
* Let’s dive into the basics first:
* **Algorithm** – it is a finite sequence/list of steps to solve a problem.
* **Linear Programming** - is a system of equations which has a set of inputs with a finite function which results in an output.
* So, can an algorithm have infinite loop? – By definition, an algorithm cannot go into infinite loop. Because it is a **finite** number of steps to solve a problem.
* Now that it is clear an algorithm cannot go into an infinite loop, let’s look at a higher aspect of algorithm. What is a **Process?**
* Process is an execution of code that drives an algorithm.

Now here’s a question. How any real-time monitoring system works? Does it just follow a finite step to get a solution and stop? Obviously not, because we have to keep displaying the solution for a given amount of time. So, it should be clear that any real-time monitoring system generally must have an infinite loop.

For example:

AWAC - [Airborne early warning and control](https://en.wikipedia.org/wiki/Airborne_early_warning_and_control), a mobile, long-range [radar](https://www.britannica.com/technology/radar) surveillance and control system for air defense. The system was developed by the U.S. Air Force to detect aircraft in 1963. It is still running. The fact to note here is the plane should never land with a radar to detect any object coming into USA. That means, it should give a continuous output within every fraction of second for an infinite amount of time. So, it is by far related to infinite loops.

The take-away is the process that drives the algorithm, which is still finite steps, is infinite for most real-time applications.

Thus, the answer to the big question is: Yes, a genetic algorithm can have an infinite loop as long as they get a progress and output along the way. Similar to a process.

Let’s be clear about some aspects of Genetic Algorithm and its related terms.

Firstly, Genetic algorithm never explains the past. It is just a way of looking at the genetics to devise artificial solution and develop the physical computing. Evolution here means iterative change.

Can an algorithm get a solution?

* Algorithm by definition gets to a solution, but a GA does not guarantee a solution.
* It is just some **Heuristic**, a “rule of thumb”, so to speak.
* In terms of solving a problem, people devised certain things to make life easy. Such as Flowchart. We also discussed about UML (Unified Modeling Language).

**UML** – way of picturesque documenting of our projects

**Flowchart** – visualization of logic (tree or graph in data structure)

Defining and Analyzing the six related terms of GA:

* **Soft computing** - putting together probabilities and conditional logics to get to the solution which might not be 100 percent accurate.
* **Evolutionary computing** – just a generic name to wrap up the development of evolutionary programming, genetic algorithms and genetic programming.
* **Evolutionary programming** - hybrid of Genetic Programming and Genetic Algorithm. We have the framework of the code but need to know the value of parameters, which is provided by GA. It’s a GP in the essence but using GA to figure out the value of parameters to get to the solution.
* **Genetic programming** – it is just growing of code/programs not the solution itself. We use tree or graph data structure to store the solution.
* **Genetic algorithms** - it is all about producing the solution. We store the results in array – basic data structure.
* **Evolutionary strategies** – It’s the spinoff of GP where the selection process is broader than GP but change process is limited.

**Keywords:** Genetic Algorithm, Heuristic, Genetic Programming, Infinite Loop, Algorithm, linear programming, finite, process, UML, Flowchart, Soft computing, Evolutionary computing, Evolutionary programming, Genetic programming, Genetic algorithms, Evolutionary strategies

**DATE: Sept. 9, 2019**

* Richard Karp (1972) – Wrote “Reducibility Among Combinatorial Problems” with Steven Cook.
  + This document deals with NP Hard problems; we call this “Karp 21” because it has 21 computational applications/problems that are NP-Complete. Problems like these show how one problem has other derivatives.
  + FUN FACT: Karp only proved 20 problems, Stephen Cook proved the other one a year prior.
  + Stephen Cook (1971) – Proved the existence of a NP-Complete problem by showing that the SAT problem is NP- Complete; emphasized on satisfiability; Cook-Levin Theorem (<http://pages.cs.wisc.edu/~shuchi/courses/787-F07/scribe-notes/lecture09.pdf>).
* Richard Karp wrote a set of 21 NP-complete problems using Steven Cook’s Boolean satisfiability problem Karp was able to derive 20 more problems that are NP-complete.
* Solvability vs Complexity
  + **Solvability** refers to whether there exists a machine
  + **Complexity** refers to bounds on resources necessary for your code/solution/implementing your algorithm.

**Solvability** vs. **Complexity** –

* In earlier years the computer wasn’t born yet, only Turing machines.
* It wasn’t until Von Neumann (http://www.alanturing.net/turing\_archive/pages/Reference%20Articles/What%20is%20a%20Turing%20Machine.html) in 1941 in Princeton that configured wires that they had a machine that was inspired by Alan Turing’s work (http://www.inf.ed.ac.uk/teaching/courses/propm/papers/Cook.pdf).
* Their concerns weren’t complexity, but nowadays as programmers we care about it.

**Complexity** is the bound of resources necessary for implementing your algorithm.

* By studying Turing’s work, you only need two things: CPU and Memory. The CPU is divided into three parts: **Primary**, **Secondary**, and **Tertiary.**
* **Primary** – The CPU interacts with the RAM. It is a bit more costly; here, every value temporary or not is stored in memory.
  + **Moore’s Law**: assumed that every CPU could help you develop a better CPU; takes about 18 months for people to use an old computer to design a new one, speed doubles every 18 months, stayed true for the last 60 years but recently started to break down; while the CPU has been doubling every month, the memory has not; hence, the CPU Speed grows faster than memory speed <http://www.mooreslaw.org/>.
  + Here is an analogy in relation to Moore’s Law: *A chain is only as strong as its weakest link;* hence*, a CPU is only as fast as its memory.*
  + Nowadays the CPU interacts with Cache instead of Ram, but it is obviously marketed differently.

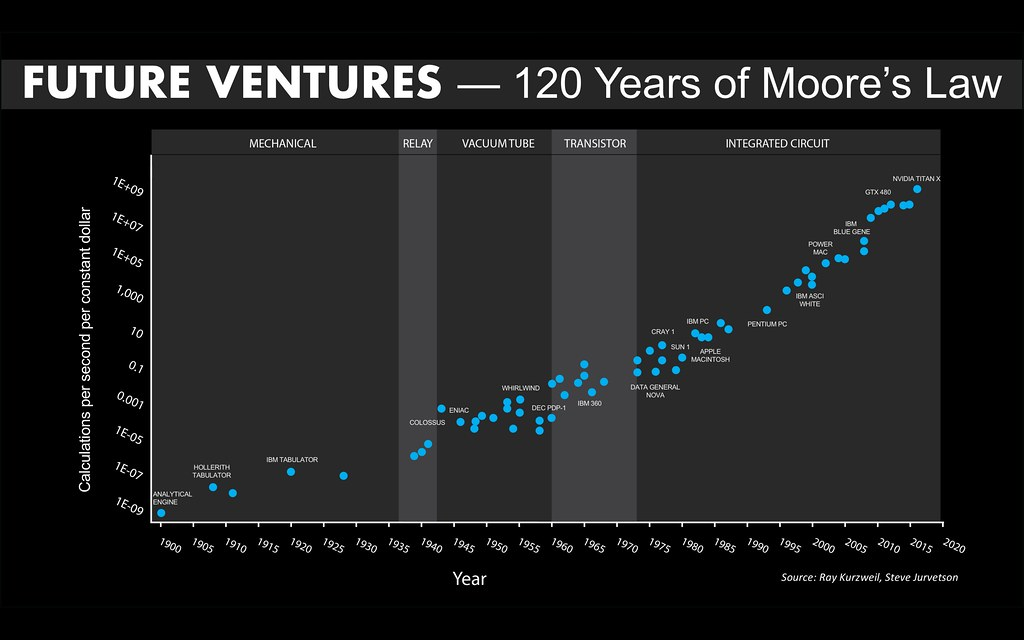


Image: <https://live.staticflickr.com/481/31409423572_62da9ae565.jpg>

* **Secondary** – longer term storage which is the hard drive; speed, cost, and storage capacity are the payoffs.
* **Tertiary** – long term storage but not fixed; is removable such as USB drives, blue ray, and disks; not directly accessed by the CPU.
* People didn’t care if the machine operations took a long time (**Complexity**), their goal was **Solvability.**

**The Halting Problem –** Un-solvability of the Halting Problem (on the Turing Machine) –

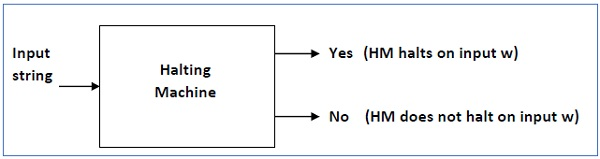


Image: <https://www.tutorialspoint.com/automata_theory/turing_machine_halting_problem.htm>

* Imagine you had a tool that can take a look at your code, debug, cleans up your code and can tell if the code can go through infinite loops; according the unsolvability of the halting problem, you cannot have such a code, because there is a limit to the universe of knowledge.
* Here are some links that expand on what the halting problem is:
  + <https://www.geeksforgeeks.org/theory-of-computation-halting-problem/>
  + <https://www.scientificamerican.com/article/why-is-turings-halting-pr/>
* **Rice’s Theorem** – any nontrivial property about a language is undecidable! Expanded to other Turing machine cases.
  + <http://kilby.stanford.edu/~rvg/154/handouts/Rice.html>
  + <https://www.tutorialspoint.com/automata_theory/rice_theorem.htm>
* Computabilitycan only occur over a finite amount of numbers, but the odds are overwhelmingly against you because there are a lot of numbers. In turn, you increase the odds, but there are cases in computational geometry where there has been success.
* When a selection decision is necessary the computer “randomly” selects which one (NO GUARANTEES that it will work!)
* **Polynomial** is a function that cycles a fixed amount of time and is “easy”.
* **Exponential** is a function that cycles forever and is “hard”.
* A computer needs only CPU and its memory in order to function. Every other piece of hardware is an accessory and is just there for convenience.
* **Brute force** is an algorithm that runs in exponential time. For a particular problem all possible solutions are generated until an answer is found. The only prerequisite is that you have a way to properly and quickly verify if the proposed solution is a correct one.
* **Random algorithm** (Random Number Generator) is an algorithm that is supposed to return a number at random with the condition that every number can be equally selected, similar to a dice roll where every number on the die has an equal chance of occurring. The problem with random number generators is that it is not truly random because if you look at the source code, then it is possible to predict the next number. One way that was used to achieve randomness was to use atmospheric pressure readings to generate numbers. The purpose for these random algorithms is for when a selection decision is supposed to be made by the computer so it randomly selects an answer but without guarantee that it is the right one.
* **Heuristic** is a process where the computer makes a decision based on prior knowledge but ultimately guess on the remaining choices. This process also does not guarantee a correct solution but it is a close estimate and returns a solution in a reasonable amount of time.
* **Deterministic** is an algorithm where every step is well defined in advance. Every input has their own steps it goes through to reach an output. This algorithm guarantees the same solution for a particular input.
* **Non-deterministic** is an algorithm where different outputs can be reached when using the same input. Used to find approximate solutions when it is too costly to find the exact solution.
* Heisenberg uncertainty principle says that if you observe an event, you cannot measure it because once you insert yourself, it changes. So even if we were to use some natural random measure for randomness, to insert the number into the computer it needs to be rounded (because we can only use countable data (natural numbers) in computation). Because we cannot store it in its entirety, we are changing the data by entering it.
* So using RNGs, if we don’t know the answer, we generate a random number and see what happens.
* There is a claim that only 10% of a deterministic algorithm runs for any particular input, so 90% of the algorithm is not used.
* Only brute force and deterministic algorithms are guaranteed to find a solution.
* Brute force is one extreme, because there are no intelligent decisions being made. The other extreme would be non-deterministic algorithms.
* Nondeterministic- “harry potter” computing, we want the algorithm to tell us an answer instantaneously out of a set of options.

Oracle computing- computing modeled after an oracle/seer in that it gives an instant answer “magically”. Theorists came up with this to explain nondeterministic computing.

Dijkstra wrote about programming languages and said that nondeterministic java would be exactly the same as regular java, except for one extra command. This command would be: instantaneously choose one of the following such that f(x)=0. Time would be constant and constants are dropped in complexity, so a command like this wouldn’t affect nondeterminism/criteria that it’s a nondeterministic algorithm.

A deterministic algorithm A is specified by:

* A countable set D (the domain)
* A countable set R (the range)
* A finite alphabet delta such that delta\*^R=0
* An encoding function E: D->delta\*
* An transition function T: D->delta\*uR.

The **class NP** is very extensive. Loosely, a recognition problem is in NP if and only if it can be solved by a backtrack search of polynomial bounded depth. A wide range of important computational problems which are not known to be in P are obviously in NP. For example, consider the problem of determining whether the nodes of a graph G can be colored with K colors so that no two adjacent nodes have the same color. A nondeterministic algorithm can simply guess an assignment of colors to the nodes and then check in polynomial times whether all pairs of adjacent nodes have distinct colors.

In Theoretical Computer Science, the two most basic classes of problems are P and NP.

**P** includes all problems that can be solved efficiently. For example: add two numbers. The formal definition of "efficiently" is in time that's polynomial in the input's size.

**NP** includes all problems that given a solution, one can efficient verify that the solution is correct (the acronym stands for nondeterministic polynomial (time), which is quite confusing). An example is the following problem: given a bunch of numbers, can they be split into two groups such that the sum of one group is the same as the other. Clearly, if one is given a solution (two groups of numbers), it's dead simple to verify that the sum is the same. (This problem is called number partitioning).

What's unknown is whether problems such as the one above have an efficient algorithm for finding the solution. This is the (in)famous P = NP problem, and the common conjecture is that no such algorithm exists.

Now, NP hard problems (the above being one) are such problems that were shown that if they can be efficiently solved (which, as mentioned, is believed to not be the case), then each and every problem in NP (each and every problem whose results can be efficiently verified) can be efficiently solved. In other words, if you're up to showing that P=NP (or, for that matter, the reverse), you might want to take a stand at any of those NP-hard problems since they are "equivalent" in some way to all others.

What's somewhat confusing is that many people confuse NP-hard with NP-complete and so they use NP-complete to refer to those hard problems (e.g. "this problem is NP-complete so it's unlikely to be solved"). But, NP-complete problems are a slightly different animal (and a subset of NP-hard problems) and their "completeness" is usually not relevant for claiming hardness. If they're impractical, isn't it as useful as counting angels on pinheads?

Some exponentials are more impractical than others.

The better ones lead to solutions of moderate-sized real problems.

Improvements mean big differences in solvable problem sizes (typically multiplying problem size by some factor) while faster technology doesn't help so much (typically adding only a small constant to the problem size).

Polynomial time algorithmic has been criticized on the basis that if everything's fast, you don't care exactly how fast it is. This may not always be true (e.g. if server must handle many requests, speed matters) but exponential problems more likely to lead to visible runtimes where improvements can be perceived or even make the difference between solving and not solving a problem.

The alternative approach of approximation algorithms (more standard in theoretical CS) is not always suitable (approximations can be bad, e.g. for graph coloring problems; the cost of computing a true optimal answer may be made up for by the value of having that answer; or it may be a problem where approximation does not make sense).

One can sometimes make the exponential part of the time bound depend on a parameter other than problem size, which could be much smaller ("fixed parameter tractability").

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is O(n^k) for some nonnegative integer k, where n is the complexity of the input. Polynomial-time algorithms are said to be "fast." Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time. Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

**Keywords:** Solvability, complexity, primary, secondary, tertiary, Moore’s law, the halting problem, rice’s theorem, polynomial time, exponential time, brute force, random algorithm, heuristic, deterministic, non-deterministic, NP class, P class

**DATE: Sept. 11, 2019**

Non-Deterministic Algorithm, model, procedure, process, etc., whose resulting behavior is not entirely determined by its initial state and inputs.

1. The programmer knows (at least) the range/set of values that contain the solution.
2. The programmer has to inform the computer (i.e. code) this information. **GIGO**: Garbage in Garbage Out implies bad input will result in bad output.
3. The computer always makes the right choice (**selection**).

**Brute force:** Brute force search is a general technique of solving an algorithm by generating all possible solution to a problem and comparing all solutions that satisfies the problem. It doesn’t include any shortcut to improve the solution but tries all possibilities until a correct solution is found.

1. There exists an effective verification procedure to very proposed solution.
2. The non-deterministic computer choose/selects the correct choice in O(1) time based on the given range of values and satisfying the given set of constraints.

Baseline vs Benchmarks

**Baseline**

a baseline is the expected values or conditions against which all performances are compared. A baseline is a fixed reference point.

**Benchmark**

A benchmark is about analyzing the relative performance of an application. If the application performs below the given baseline, then it is regarded as broken.

NP Hard VS Complete

**Complete** is a subset of hard problems.

Complete ⊆ Hard

**NP hard problems**

NP-hard problem is also known as Non-deterministic polynomial time hard. NP-hard problems are at least as hard as every problem in NP reduction. If X is NP-hard, the problem Y in NP-complete can be reduced to X in polynomial time.

**NP complete problems**

NP complete are verified (NP) problems that can be reduced in polynomial time. If Y is in NP, then X in NP can be reduced to Y in polynomial time

Optimization vs Complete

Optimization

**Optimization** is a process of modifying a system to make some features of it work more efficiently or use fewer resources.

**Complexity Zoo**

Complexity Zoo is a website created by Scott Aaronson which contains a (more or less) comprehensive list of Complexity Classes studied in the area of theoretical computer science.

Reduction

A **reduction** is an algorithm for transforming one problem into another problem. A sufficiently efficient reduction from one problem to another may be used to show that the second problem is at least as difficult as the first. Problem A is reducible to problem B if an algorithm for solving problem B efficiently (if it existed) could also be used as a subroutine to solve problem A efficiently.

**Practice question 1:**

Alice is a Chemistry professor.

She’s researching about a hard problem.

She meets Bob in the lunch room, a Physics professor who is also dealing with a problem in his research, but he doesn’t know exactly how hard is his problem.

They tell each other about their research problems.

After listening to Bob’s problem, Alice tells Bob she knows a way the answer for her problem can help solving Bob’s problem, so she tells Bob she can help him to solve his problem if he helps her to solve her problem.

Whose problem is harder?

**Solution:** **Alice’s hard problem ≥ Bob’s? problem**

Alice’s problem is either harder or has the same level of difficulty as Bob’s because the answer from her problem will also solve Bob’s problem. In other words, a hard problem cannot be solved using an easy solution, but an easy problem may be solved using a hard solution.

**Practice question 2:**

Again, Alice meets Bob in the lunch room. They both have a problem to solve in their researches. Bob’s problem is a hard problem, though, Alice doesn’t know how hard her problem is.

Alice tells Bob she knows a way the answer for her problem can help solving Bob’s problem, so she tells Bob she can help him to solve his problem if he helps her to solve her problem.

Whose problem is harder?

**Solution:** **Alice’s? problem ≥ Bob’s hard problem**

Alice’s problem is either harder or has the same level of difficulty as Bob’s because the answer from her problem will also solve Bob’s problem.

**Practice question 3:**

In what situation can Alice’s problem be an easy problem, but still generate a solution for Bob’s hard problem?

**Solution:**

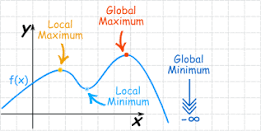
Only when the cost to covert Alice’s chemistry problem solution into a solution that will solve Bob’s physics solution is a hard process. This is also called the **cost of doing business**, or the **transformation cost**, or the **translation cost**).

So, Every NP hard problem reduces to satisfiability.

\*a complete problem is also hard but hard is not complete

Hard problems are typically optimization problems and complete problems are generally decision problems

local max/min is a possible solution whereas global max/min is the true solution.



An optimization problem would need a solution to be verified within a reasonable amount of time, but this is unlikely and therefore will never be considered a complete problem

**Keywords:** GIGO, selection, brute force, baseline, benchmark, complete, np hard, np complete, optimization, complexity zoo, reduction, transformation cost/translation cost

**DATE: Sept. 16, 2019**

REDUCIABILITY AMONG COMBINATORIAL PROBLEM BY RICHARD KARP

There are three types of approach to solve the problem: -

Brute Force Random Genetic

1. **Brute Force: -** Brute Force algorithm is a type of algorithm in which we try all the possibilities to get the solution of the problem. A common example of brute force is digit password.
2. **Random:** It is a pseudorandom number generator also known as a deterministic random bit generator (DRBG). PRNG refers to an algorithm that produce sequences of random numbers.
3. **Genetic:** In genetic algorithm, we generate a population of every points at each iteration so the beat point of population approaches to an optimal solution.

These three approaches use the same type of data structure.

There are three types data structure are:

1. **Parameter File:** It is file that contains at least one parameter and its assigned value. The hierarchy is uses by the data integration service for parameter file to identify parameter values to use when you run a specific project.
2. **Static Structure:** A static data structure is a collection of data in memory that is fixed in size. So, the maximum size is known in advance. The content in the data structure can be changed but without changing the memory space allocated to it. Example of static data structure is array.
3. **Dynamic Data structure:** In dynamic data structure the size of structure is not fixed and can also be modified during the operation performed on it. So, it helps the programmer to control how much memory is utilized and allocated and deallocated from the heap as needed.

Not every exponential time solution is always bad

On the other hand, not every polynomial time solution is good

The Millennium Prize Problems – the Clay mathematics Institute of Cambridge established 7 prize problems, one of them being P=NP where the winner would get a prize of 1,000,000.

Complexity class P vs NP

* P = NP is an unsolved problem. It asks the problem whose solution is verified quickly that can also solved quickly.
* Clay Mathematics Institute give US$1,000,000 prize for the first correct solution.

Problems that are classified as P can be solved with worst case time of polynomial time. An algorithm exists that solves any instance of size n. It can be solved by a deterministic turning machine.

NP problems are different from P problems. You can think of P problems as those that follow one path, whereas NP problems are those that can take different paths. Because of this, it is solvable by a non-deterministic turning machine. One way the Turing machine deals with these choices is to create copies of itself and follow more than one path. However, the time can increase exponentially due to the copies being created. If any path leads to acceptance, then the input is accepted.

NP – Nondeterministic (NP) – More like a question

Polynomial Exponential n=variable

nk kn K=constant

Polynomial algorithms typically are nested loops

P= you can solve the problem in Polynomial time

NP = Verify the solution in Polynomial time

# of steps vs # of choices

The difference between deterministic and non-deterministic Java is that ND Java has the option to instantaneously select a correct solution.

Instantaneous correct solution (correct solution) – Guaranteed

The P VS NP problem – This is one of the great unsolved math and computational problems of our time. Also called P=NP. The question is, can all NP problems be solved in polynomial time, which would mean that the NP problems are P problems if the right algorithms were used or is it that some NP problems will have no solution that can be solved in polynomial time and will remain unsolved no matter what algorithms we develop?

The goal is to prove either P=NP (all NP problems belong to P and can be solved in polynomial time) or P≠NP (no matter what algorithm we use, it will be larger than polynomial time). We also need to consider NP-Hard and NP -Complete problems.

NP-complete – class of NP problems where time complexity is greater than polynomial time, are verifiable in polynomial time and belong to NP hard problems. NP hard problems are at least as hard as the hardest NP problems, but don’t need to be verifiable in polynomial time.

If we can show that P=NP, many new ideas and innovations can occur in the future.

Set of problems from Karp’s paper:

1. Satisfiability
2. Integer Programming
3. Satisfiability with at most 3 literals per clause
4. Knapsack

References

1 - <https://interestingengineering.com/p-vs-np-np-complete-and-an-algorithm-for-everything>

2 - Reducibility among Combinatorial Problems by Richard Karp

Boolean Satisfiability Problem

The Boolean satisfiability problem (SAT) is problem in which variables of given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates is TRUE and this case formula is satisfiable otherwise if no such variable exists and it evaluates FALSE then it unsatisfiable. SAT is first problem that proven to NP-complete by Cook Levin theorem.

Reduction

A reduction is an algorithm for transforming one problem to another problem. In problem A is reducible to problem B if an algorithm for solve the problem B efficiently and that can be used as subroutine to solve problem B. and children solve the parent problem

These are some problems that are complete:

SATISFIABILITY

CLIQUE 0-1 INTEGER SATISFIABILITY WITH AT

PROGRAMMING MOST 3 LITERAL PER CLAUSE

NODE SET

COVER PACKING CHROMATIC NUMBER

FEEDBACK NODE EXACT COVER CLIQUE COVER



FEEDBACK ARC



DIRECTED HAMILTON CIRCUIT 3-DIMENSIONAL MATCHING



UNDIRECTED HAMILTON HITTING SET



CIRCUIT STEINER TREE



DET COVERING KANAPSACK



SEQUENCING PARTITON MAX CUT



Key Goal: If you were to design an algorithm, that is non-deterministic, to solve a problem in polynomial time, can you also solve it in deterministic time?

The P = NP question is an issue of complexity, not solvability.

|  |  |
| --- | --- |
| **(d)P** | **NP** |
| 1. P means “polynomial-time algorithm” 2. P uses deterministic computer (ie. deterministic Java) 3. P means you can solve the problem in polynomial time 4. *Focus*: Number of steps (defined in loop structure) | 1. NP uses non-deterministic computer (ie. non-deterministic Java) 2. NP means that you can verify the solution in polynomial time 3. NP means instantaneous, *correct* selection (guaranteed) 4. *Focus*: Number of choices |

Difference between polynomial and exponential:

**Polynomial** (n^k): where k is the constant exponent and n is the variable. From an algorithmic perspective, a polynomial algorithm typically consist of nested loops.

**Exponential** (k^n): where k is the constant base and n is the variable exponent.

According to Dijkstra, a deterministic and non-deterministic version of a language is almost the same. The only difference is that the non-deterministic version has the capability for an instantaneous and *correct* selection.

From a complexity point of view, the computer can guarantee to select the correct answer, instantaneously, under constant time. This is significant as this non-deterministic aspect plays no role in the overall complexity of the algorithm (since constants are ignored in the complexity analysis).

Further questions and remarks

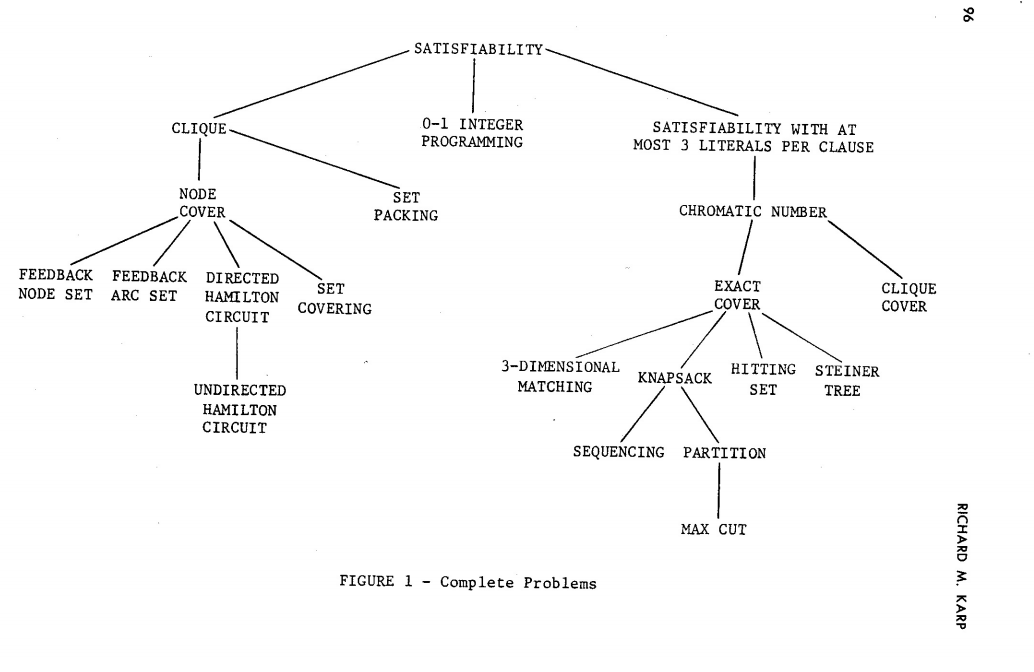
1. If you are in P, you are in NP since P is a subset of NP (since you can certainly verify the given problem in polynomial time). The question is whether NP is a subset of P?
2. If P were not equal to NP, and you have an algorithm that is NP-Complete, can you solve it deterministically? Yes. You can always solve an NP-Complete problem because there exists a brute-force approach to solve the problem. And you can verify it, since that problem lies in NP. The problem is that the brute-force method is highly exponential. (ie for length n, there are 2^n bit strings)
3. If P were equal to NP, a polynomial-time algorithm does not necessarily provide a practical and reasonably-time algorithm. For example, n^100 is, from a complexity perspective, in polynomial time but the constant is too big. On the other hand, not every exponential time algorithm is deemed to be “bad”, certainly not as bad as n^100.

**Reduction**

Notion of reduction: Given two problems, A and B, if problem A reduces to problem B, then problem B solves problem A.

When looking at the tree consisting of the 21 problems, we must take a *recursive* approach, that is, the parents are known and the children are unknown. The parent reduces to the child, thereby the child solves the parent’s problem. The parent is “extending” to the child problem. These problems are grouped together because the data structures used to solve each batch of problems are similar.

Note: Satisfiability is already NP-complete (Cook had already proven it a year before Karp published this paper).



[source: Reducibility Among Combinatorial Problems by Richard Karp, 1972]

Problems 1, 2, 11, and 12

Note: These problems are grouped together because the same algorithm can solve all of the following below.

Problem 1: Boolean satisfiability problem

* A **Boolean predicate** is a mathematical function where the inputs and outputs are from the set of {0 or 1} using and/or/not
* Satisfy the truthfulness of the function by finding a sequence of Boolean predicates
* The data structure used is a n-sized bit strings with 0 and 1’s stored in a 2^n size array

Problem 2: 0-1 Integer programming

* Integer programming is a branch of optimization that involves creating mathematical equations to solve problems.
* Uses a series of binary (set of {0, 1}) to arrive at a solution when there are two mutually exclusive options.
* In a given problem, each variable, represented by either a 0 or 1, could represent the selection or rejection of an option.

Problem 11: 3-Satisfiability (3-SAT)

* Determine the satisfiability of a formula where each clause is limited to at most three literals.
* Reduce the unrestricted SAT problem to 3-SAT by transforming each clause L1 to Ln to a conjunction of n-2 clauses, where X2 to Xn-2 are variables that do not appear elsewhere.

Problem 18: Knapsack

* Given a set of items, each with a weight and value, the goal is to optimize the number of each item such that the total weight is less than or equal to the given limit while the total value of the knapsack is as large as possible.
* This problem can be applied to the topic of resource allocation and financial constraints.

**Keywords:** Brute force, random, genetic, parameter file, static structure, dynamic data structure, polynomial, exponential, reduction, Boolean predicate

**DATE: Sept. 18, 2019**

An important thing to keep in mind is that we are not suggesting only one solution but rather many solutions and taking the best one that fits our tolerance for the problem at hand. This part of the lecture will evaluate the assigned problems of Satisfiability,0/1 integer programming ,3-SAT, and Knapsack respectively marked 1,2,11,18 for reference in Karps pdf. Additional information and links will be provided for better understanding towards the end of the word document.

Review of Problems 1, 2, 11, and 18 from Richard Karp’s Paper: Reducibility Among Combinatorial Problems

1. **Satisfiability (SAT)**
   * Involves Boolean predicates
     + Built using operators AND, OR, NOT
     + Constants are true (1) and false (0)
     + (X1, …,Xn) ∈ {0,1}
     + P(X1, …,Xn) 🡪 {0,1}
   * Objective is to determine particular truth assignments that make a function true
   * Assign values to the Boolean variables such that formula is true and satisfiable
   * Solution is a bit string of size n

* 2 SAT
  + Special case
  + Can be solved in polynomial time
  + Consists of at most 2 literals per clause
  + Expressed in Conjunctive Normal Form (conjunction of clauses, where each clause is a disjunction of two variables or negated values

1. **0/1 Integer Programming**
   * Involves some or all variables to be integers

* Strictly two options are available in order to establish a linear problem-solving framework
  + Consists of executing a series of decision problems involving “yes” or “no” choices, represented by 1 and 0 respectively, in order to obtain the correct answer
  + Solution is represented as a bit string

1. **3 Satisfiability (3 SAT)**
   * Consists of at most 3 literals per clause
   * (Xi V Xj V Xk) Λ (…) Λ… Λ (…)
   * Determine a truth assignment such that each clause is satisfied
   * Typically used as a starting point for proving that other problems are also NP hard
   * Can be generalized to k-satisfiability, where each clause is limited to at most k literals
2. **Knapsack Problem**
   * Given:
     + A set of n items, with a weight and value associated with each particular item (not all items are required to be placed in the knapsack)
     + A knapsack of capacity W (dimensions originally ignored)
     + Assumption that each item is of equal importance
   * Objective is to obtain the maximum value of the items placed in the knapsack without exceeding the knapsack’s capacity
   * Can be solved by creating two arrays representing weight and value, and determining the maximum value subset, such that the sum of the weights of the subset is <= W
   * Fractional Knapsack
     + Special Case
     + Allows the ability to take a portion of a particular item
     + Greedy Algorithm will guarantee solution
   * Solution involves a bit string of 0s and 1s, where each bit represents whether or not a certain item is in the knapsack

Each of the problems described above are essentially the same in regards to obtaining the solution. However, it’s how you verify the solution that matters.

**Linear Programming**

* Introduced by George Dantzig
* System of Equations where variables aren’t raised to a particular power other than 1
* Variables are from the set of Real Numbers
* Possibly linear constraints on variables x, y, and z

Programming originally referred to solving a linear system of equations

* Karp defined this in a limited way
* No other constraints on x, y, and z
* Modern version of programming has constraints

An important concept in programming is understanding how to convert between certain bases

* The following represents pseudo code for transforming from one base to another:

for soln = 0 to (Math.pow (2, n) – 1) {

**transformBase** (10, 2, soln, solution, n);

if (**verifySolution**(solution) == TRUE

break;

}

Partition Problem

* NP Complete problem
* Involves partitioning a given multiset into two subsets such that the sum of the integers in the first subset is equal to the sum of the integers in the second subset
* Example: partition of S = {e1, …, em} into k subsets
  + Consider each set like a “box”
  + Every box must be used
  + No two boxes have the same item
* S1 ∪ S2 = S
* Intersection of the subsets must be empty

**Keywords:** Linear programming, knapsack, SAT, SAT3, 0/1 integer programming

**DATE: Sept. 23, 2019**

Last class we discussed 1,2,11,18.

Problem 4,6,14,20-partition.

Same algorithm as converting from base10 to base2. Similar to this we will convert from base10 to base K. Using partition: USi= S. When you split it up, it goes in two arrays and are equally distributed.

Pseudo code- for i=0 to Math pow (K, n)-1

And last time k =2, for problems1,2,11,18 are solved with same algorithm.

21-Max cut that belongs to NP

And mincut belongs to P.

**Graph** G=(V, E)

E is subset of V \*V.

2-partition. V such that 2,3,6 is set 1 and 1,4,5 is set 2. Add all the edges for each set.

Partition pi(s)

Alpha. -partition pi ( s, k).

Π(s,k)=(n^-Σ(i=1 to n-1)(-1)^I + C (choose i from n) (n-1)^m))/n!

Where n=k.

Similar algorithm what we did last should solve many problems.

Homework2: Why didn’t we say we can solve 20, 21. We are going to get solution but it is wrong approach.

Case4- set packing, you have L boxes. No duplication. Mutually exclusive. Spanning that covers everybody. (but may not span “U”).

Case 6- Set Covering- V boxes – covers all items “U”. (but may not be mutually exclusive “n”)

Case14- Exact Cover: h boxes

1. “U”- all items covered
2. Mutually exclusive “n”

//as if partition.

4,6,14,20- partition

20- partition problem can be solved by partition code.

4-set packing problem can be solved by combination code.

6- set covering problem can be solved by combination code.

Combination vs permutations

N items choose i from n C (I ,n)= n!/ (n-i)!\* i!

iCn(I,n)

1 combination = 1! **Permutation**

P(i,n), n/(n-I)! = C(I,n) \* i!

Homework 3: which famous category of function does permutation related to and why?

2^n=Σ(i=0 to n) C(I,n)

|powerset(s)|=2^n

|s|=n.

In this lecture class, there was focused on 4 problems from Complete Problem

Problems are: 4,6,14,20 and 21

Base 10 to Base k: which means there is an algorithm that take 10 base number and convert it to any base number.

Partition: ⋃Si=S for n items elements

Element Element 0 n-1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

Head1 n

Here from Head1: the number of elements will be start from 0 to n and each boxes represent 1 or 0 bit.

Counter 0 … k-1:

///last time we discuss:

we take a value ‘k’ = 2, for problems 1,2,11,18

for i = 0 to Math.pow(k,n) – 1

where i is iterates from 0 to k to the n power – 1.

problem 21:

**Maximum Cut:** In a graph, a maximum cut is a cut whose size at least the size of any other cut.

Maximum Cut ∈ NP while Minimum Cut ∈ P

G = (V, E) where E ⊆ V×V

Here, G refers to any graph and V refers to **Vertices** and E refers to **Edges**.

In this graph G, there are V = {V1,V2,V3,V4,V5,V6} and E is the subset of V × V.

1

V2

V1

7

6

13

V3

5

4

V6

9

12

10

V5

V4

Cutting Edge are shown in orange color:

Maximum Cut: 13+1+5+9+12 = 40

**Partition:**

Although in mathematics we use this symbol as a 𝝿 but in our lecture it refers partition.

𝝿(S) - intra x

k-partition: 𝝿(S, k) – inter v

the formula: 𝝿(S,k) = (nm – i+1 m)/n!

Problem 4

**Set Packing:** Suppose there is a Set S where a list of subsets. Elements of the Set S are not common to each other, and it is a disjoint.

Example: S= {{2,3},{4,5}}

{2,3,4,5}

L. boxes – there is no duplication.

Mutually Exclusive “n” elements. (but map not span “U”)

Problem 6

**Set covering:** As oppose to Set Packing, it elements of the Set common each other and covers the common elements and are not disjoint.

Example: S= {{2,3},{2,5}}

{2,3,5}

assumes there are ‘k’ boxes

Covers All Items “U”

**Note**: But not be mutually exclusive “n”

Problem 14

**Exact Cover:** is a decision problem to determine if an exact cover exists.

There are, H boxes or more

1. “U” – all items covered
2. mutually exclusive “n”

//as if H-Partition

Solving Problem 20, 4 and 6:

Problem 20:

Partition problem can be solved by partition code

Problem 4:

Set Packing can be solved by partition code

Problem 6:

Set Covering can be solved by Combination code

Combination vs Permutation

Let’s we have N items.

**Combination:**

(n, C, r) = Choose r elements from n Elements

Formula: n!/(n-i)!\*i!

**Permutation:**

(n, P, i) : means n permute i ways.

The formula of Permutation: (n, P, i) = n!/(n-i)! = (n, C, r) \* i!

Notice:

2n = Exponential growth of summation where n choose i.

Power Set(S)

|S|=n which means the elements of Set is equal 2n

|S| refers the cardinality of S or the total number of elements inside the S set.

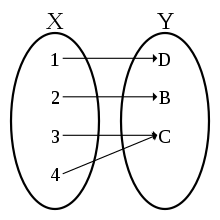
**Keywords:** Graph, permutation, maximum cut, partition, set packing, set covering, exact cover, combination

**DATE: Sept. 25, 2019**

**🡪Surjection** - can be defined as if for every element y in the codomain Y of f there is at least one element x in the domain X of f such that f(x) = y. It is not required that x be unique; the function f may map one or more elements of X to the same element of Y.

In other words, it means every element of x should match to every element of why regardless of x is unique or not. It is also known as onto function.

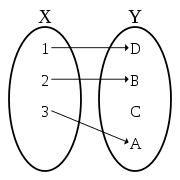
Diagram representation of surjection:



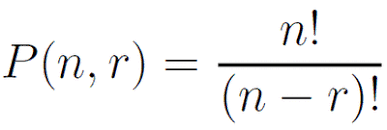
In diagram, you can see as C is linked to two of the x elements which makes this function different from other injection. We also learned there are two onto functions with one partition between them.

**🡪 Injection-**an injective function or **injection** or one-to-one function is a function that preserves distinctness: it never maps distinct elements of its domain to the same element of its codomain.

In simple terms, every element of function’s codomain is image of at most one of the element of its domain.



Permutation are similar to injective functions.



Relation between Permutation and combination. Multiplying combination function with r factorial (r!) where r is a just a number like n gives you permutation.

nPr. ==> n ! /(n-r) ! = ( nCm  \* r!)

With permutation, we care about the order of elements while with combination order doesn’t matter.

🡪 **Base conversion algorithm** (problem 20,21 on karp’s paper) – This is a n-p complete problem that is there is no solution for this.

Brute force algorithm can be applied here.

Let’s start with something simple:

S0= ver 0. And. S1= ver

V0 V1 V2 V3

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |

The first row is m and second row is p.

Order doesn’t make a difference in partitions. It generates twice the amount of strings.

Bit strings approach is ideal for (to generates all subsets) generating the Power set.

Subsets- all possible choices/combinations.

Bit Strings are good for subsets not partitions.

🡪 **Chromatic Number** can be defined as the chromatic number of a graph is the smallest number of colors needed to color the vertices of so that no two adjacent vertices share the same color i.e., the smallest value of possible to obtain a k-coloring.

In simple terms, no two neighbor’s cities or vertices of graphs can have same colors.

Diagram representation🡪

  
G=(V,E)

E ⊊ V X V

n= |V|

V0. V1. - - - - - - - - - - - - - - Vn - 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 0 | 1 | 2 | 4 | 7 | 6 |

This is a k string of length n where there are K possibilities of digits (base k).

Base (in a numerical string numbers of numerals that is occupy a digit)

Problems from karp’s papers7,8,9,10 all deals with cycles.

**🡪 Cycle:** A cycle of a graph, also called a circuit if the first vertex is not specified, is a subset of the edge set of that forms a path such that the first node of the path corresponds to the last.

Cycle: A circuit that doesn't repeat vertices is called a cycle.

**Circuit**: A circuit is path that begins and ends at the same vertex.

**Loop**: A loop is an edge that connects a vertex to itself.

**Hamilton Cycle**: A Hamiltonian path or traceable path is a path that visits each vertex of the graph exactly once. A graph that contains a Hamiltonian path is called a traceable graph.

A Hamiltonian circuit is the one in which it visits every vertex once with no repeats and it must start and end at the same vertex whereas a Hamiltonian path also visits every vertex once with no repeats, but it does not have to start and end at the same vertex.

There are few functions where there are no cycle take place:

1. Matrix multiplication
2. Transitive closure
3. Depth-first search
4. Topological sort

**Partition and surjective function:** A partition of set X is a collection P = {A1, . . . , Ak} of disjoint nonempty subsets of X such that X = .

Let X be a set. If ∼ is an equivalence relation on X, then the collection {[x]: x ∈ X} of equivalence classes of ∼ is a partition of X. Conversely, if P = {A1, . . . , Ak} be a partition of X, then the relation R = . is an equivalence relation on X.

**Permutation and Injection(one-to-one):** A permutation of a set of distinct objects is an ordered arrangement of these objects. For example, how many different batting orders are possible for a baseball team consisting of 9 players. Answer is = 9!

Let’s assume we have two set s1, s2 and |s1| = n and |s2|

Suppose that n <= k, if k > n then there is no one-to-one function.

For the first element of s1, there are only k possibilities for its image, and for second element of s1 k – 1. By choosing this way for every element of s1, we will obtain k – (n - 1) = possibilities. To get the total number of one-to-one functions we multiply the all number of possibilities we have for n step.

Graph Coloring Problem:

Chromatic Number Problem: The chromatic of a graph is the smallest number of colors needed to color the vertices of so that two adjacent vertices share the same color. (Skiena 1990, P. 210)., it is denoted X(G). The independence number of G is the maximum size of an independent set denoted by α(G).

How large or small can this X, α be in a graph G with n vertices is below,

1≤χ(G)≤n

1≤α(G)≤n

Determining the chromatic number of a graph is known as NP-Hard problems.

**Graph Cycle:** A cycle of a graph is, if the first vertex is no specified, is a subset of the edge set of graph forms a path such that the first node of the path corresponds to the last. Graph cycle is also known as circuit or loop.

**Hamilton Cycle:** A cycle that uses every vertex in a graph exactly once is called a Hamilton cycle, and path that uses every vertex in a graph exactly once is called a Hamilton Path.

Cycle exists check algorithm:

* **transitive closure** - Given a digraph G, the transitive closure is a digraph G' such that (i, j) is an edge in G' if there is a directed path from i to j in G.
* **depth first search** - The depth-first algorithm sticks with one path, following that path down a graph structure until it ends.
* **topological sort** - Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering.
* **matrix multiplication**

Review on the 9-23-2019 homework questions

Which function is closely related to Partitions?

The surjection/ onto function. For every element *y* in the codomain Y of *f* there is at least one element X in the domain X of *f* such that *f*(*x*) = *y*

For an example we learned in a previous class, we can say that we have our element/items in X and our boxes (Y) our elements need to go into.

X Y X Y

1 a 1 a

2 b 2 b

3 3

2 different onto functions but 1 partition

If we want to take order into consideration would we need to divide the onto function by n! which gives us all the other ordering pairs.

Which function is closely related to permutations?

An **injection**, never maps distinct elements of its domain to the same element of its codomain, where the size of the sets are not the same. If they are the same, then it’s a bijection.

m <= n

C (n, m) = n! / (n-m)! **==** C (n, m) x m!

X Y

1 B1

2 B2

3 …

…

Bn

Which function models order in Combinatorics?

N! describes orders (all the orders)

Why shouldn’t we use base transformation algorithm for Karp’s 20th (partition) and 21st (max-cut) problems.

Example to help explain.

4 elements

S0 V0 V1 V2 V3

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 0 | 1 |

S1 V0 V1 V2 V3

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 1 | 0 |

These are the same partition, order n! doesn’t make a difference in surjections.

If were to use base transformation and generate all the strings, we would actually be generating double of what we need. The first bit slot is either a 0 or a 1 and the other bits are all interchangeable. 0XXX OR 1XXX (compliment of each other)

But using base transformation is the best approach for power sets.

Problem 12) Chromatic Number

G = (V, E) R B

E ⊆ V x V

n = |V| G

No two vertices that are adjacent to each other have the same color

Example of an answer stored in an array

The numbers represent the color

S0 V0 V1 ……. V n-1

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 1 | 2 | 5 |

K-string

Length

K possibilities

\*\*Base k \*if you care about order\*

\*\*k-partitions, partitioning the vertices in k colors \*if we don’t care about the order\*

Base (in a numerical string) – number of numerals that can occupy a digit

Cycle problems 7, 8, 9, 10

A cycle is some number of vertices (at least 3) connected in a closed chain.

Problem 7. Feedback node set, set of vertices whose removal leaves a graph without cycles. Feedback node set contains at least one vertex of any cycle in the graph. How many k nodes can we delete?

Problem 8. Feedback arc set, a set containing at least one edge of every cycle in the graph. How many k edges can we delete?

For these, we don’t need to generate all cycles, if we have a certain set, we can delete the edges in an Arc problem to see what remains or delete the nodes for the node problem to see what remains. These problems come down to; Can we delete k edges or vertices and still have a cycle.

Hamiltonian Circuit/Cycle – every node is in the cycle exactly once

For 9 and 10, the Hamiltonian cycle problems directed and undirected graphs, will be solved by generating permutations for only one cycle were interested in and check for the edges are connected for verifications.

**Keywords:** Surjection, injection, Base conversion algorithm, chromatic number, cycle, circuit, loop, Hamilton cycle, partition function, surjective function, permutation, graph cycle, transitive closure, depth first search, topological sort, matrix multiplication

**DATE: Oct. 2, 2019**

Problem 19 **– Job Sequencing:**

This is a permutation problem

What order of jobs one must follow in order to complete all jobs while meeting all deadlines?

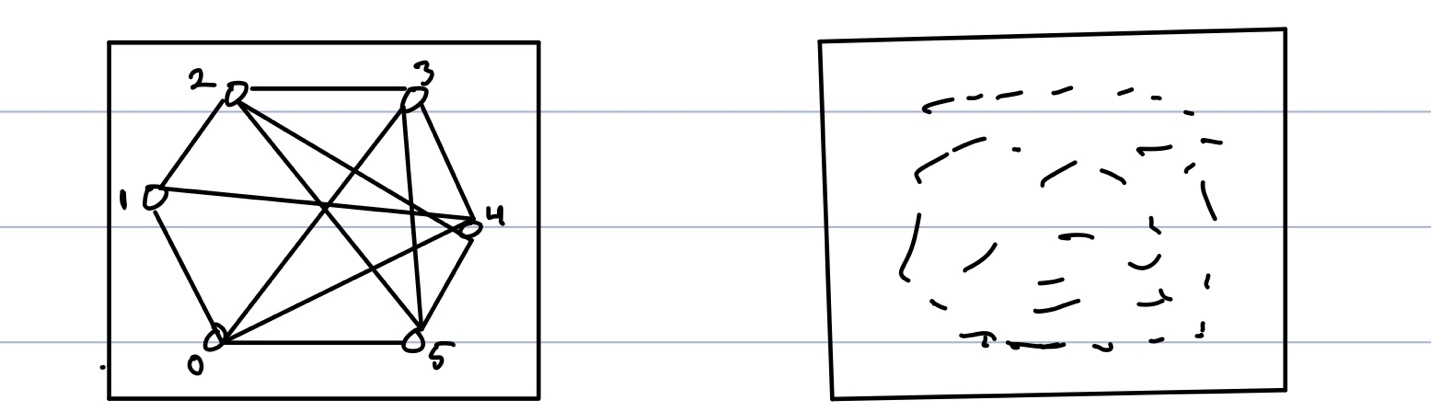
If one gets a task done, one gets paid, otherwise one must pay a penalty.

Problem 13 **– Clique Cover:**

Can one list the cliques in a graph such that every node is part of a clique?

If every edge is part of a clique, then yes, otherwise, no.

Problem #3: **Clique**



K=6 G = (V,E)

On Complete Graph K nodes, = |v|

K6 a) Strongly

E ⊆ V V b) Weekly Connected

E = V V (Complete) c) It is strongly connected if you

Reflexive Property ignore the direction.

Different between Clique and Complete Graph:

1. Clique is a subset of a complete graph
2. Clique does not allow a vertex to loop itself
3. Clique is every vertex is connected to every other vertex
4. Complete graph is whose all of each vertex connects to other vertex including itself (looping itself)

nCk = = =

= ≤ Polynomial?

No, it is still exponential since k is a variable, not constant

Karp Problem #17: **3-Dimension Matching**

Definition: Let *X*, *Y*, and *Z* be finite, disjoint sets, and let *T* be a subset of *X* × *Y* × *Z*. That is, *T* consists of triples (*x*, *y*, *z*) such that *x* ∈ *X*, *y* ∈ *Y*, and *z* ∈ *Z*. Now *M* ⊆ *T* is a 3-dimensional matching if the following holds: for any two distinct triples (*x*1, *y*1, *z*1) ∈ *M* and (*x*2, *y*2, *z*2) ∈ *M*, we have *x*1 ≠ *x*2, *y*1 ≠ *y*2, and *z*1 ≠ *z*2.

T is a finite set of numbers

U ⊆ T3 (T T T)

Choose W where W ⊆ U such that |W| = |T|

(1, T, Z)

(2, . , .)

|T| (3, . , .) |W| Each dimension is a permutation of T

(4, . , .)

(. . , .)

2D matching is in P

aCb = cab where

a = |T3|

b = |T|

Karp Problem #15 **🡪 Hitting Set**

Given a collection Σ of subsets of V , the hitting set problem is to find the smallest subset S ⊆ V which intersects (hits) every set in Σ. If we regard Σ as defining a hypergraph on V (where each set in Σ constituting a hyperedge) then we see that the hitting set problem is equivalent to the vertex cover problem on hypergraphs; this problem is NP-hard.

e.g

Λ

S1 {3, 4, 5}

Λ

S2 {4, 6, 7}

Λ

S3 {1, 2, 4}

0 🡪 2, 5, 6

0 🡪 4

1 thing in common

Karp Problem #16 🡪 **Steiner Tree**

The Steiner tree problem in graphs, called for brevity ST, is defined in decisional form as follows:

Instance:

• an undirected graph G = (V, E);

• a subset of the vertices R ⊆ V , called terminal nodes;

• a number k ∈ N.

Steiner Tree is in NP:

So firstly we want to be sure that the ST problem is actually in NP. Assume <G, R, k> ∈ ST, that is, assume the instance <G, R, k> reserves a yes answer. In this case, given a hypothetic positive solution T ⊆ G, we can check in polynomial time that:

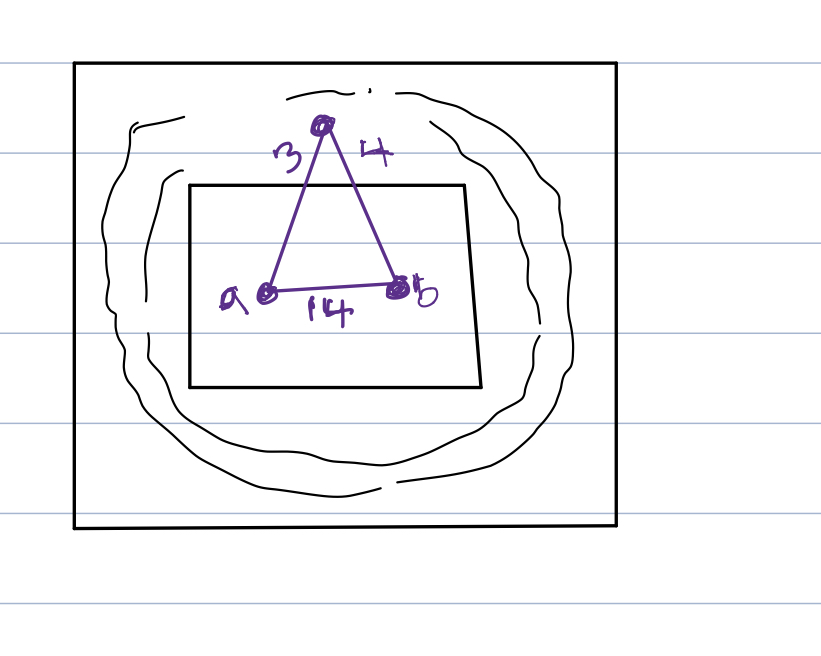
• T is really a tree: it contains no cycles and it is connected;

• the tree T touches all the terminals specified by the set R;

• the number of edges used by the tree is no more than k.

We can now proceed to the next step: select an (appropriate) known NPcomplete problem for the reduction.

Steiner Tree: reflection



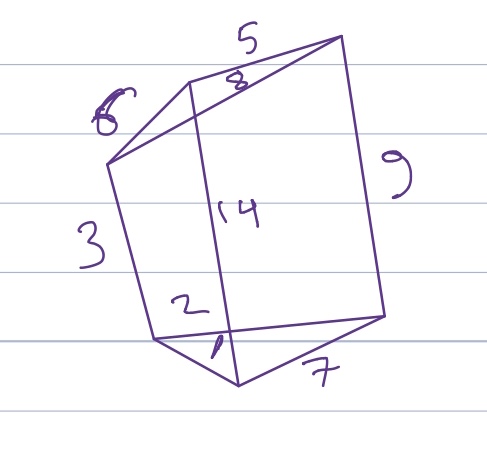
All possible root outside the box called GHOST POINT, point that you never consider  
So Steiner tree is a minimum spanner tree with go through points are not part of your area

Greedy Algorithm: **Minimum Spanning Tree (MST)**

Definitions: G (V, E, w) is an edge weighted graph if there exists a weight function w: E → R that assigns a weight to every edge e ∈ E.

G (V, E) is connected if there exists a path between any two vertices. G (V, E) is a tree if it is connected and has no cycles. It is easy to see that if G (V, E) is a tree and |V | = n, then |E| = n − 1.

Let H (V 0, E0) be a subgraph of G (V, E). We say that H is a spanning tree of G if H is a tree and V 0 = V.



Greedy Algorithm

Tree ≡ No Cycle

Spanning ≡ every node

**Keywords:** Minimum Spanning Tree (MST), Steiner Tree, Hitting Set, 3-Dimension Matching, clique, clique cover, job sequencing

**DATE: Oct. 7, 2019**

Today is the review of our upcoming quiz which is on Wednesday October 16th and there will be no classes on October 8 and 9 and 14. I will also give you guys a day off for our video day, which is on October 21, though collage is open.

**Partition**: Given a set of integers A, split all of the elements of A into two groups B and C, such that the sum of all of the elements in B is equal to the sum of all of the elements in C. For example: Suppose A was the set {1,2,3,4,6}.  Then A’ could be {1,3,4}, forming a partition (A’ and A-A’ both sum to 8).  If instead, A was the set {1,2,3,4,5}, then no possible partition exists.

By using the base 2 conversion algorithm, we can store an integer in a bit string of length n using 0s and 1s. For example:

e0 e1 ….....................en-1

0 1 0 1 1 0 1 1 0 0 0 1



n-bits

number of requirements or criterion of Partition:



1. No box is empty: 0⊂ Si ⊂S



1. No box has left behind: S= S



1. **Mutually Exclusive**: Si ∩ S =0, i≠j



4-**Set Packing**: Set packing problems is that no elements are permitted to be covered by more than one. We seek a large subset of vertices such that each edge is adjacent to at most one of the selected vertices. This problem arises in partitioning applications, where we need to partition elements under strong constraints on what is an allowable partition.

For example: let say you want to go Florida and you rent a truck that can fit L boxes, but you have 100 boxes. Therefore, in order to fit the boxes in the truck, we will search only for the items that has no duplicate, only that items. It focuses on mutually exclusive or criterion 3.

6-**Set Covering**: Set covering is decision problem Given a set of elements {{1, 2..., n} and a collection S of m sets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of S whose union equals the universe. For example, consider the universe U= {1,2,3,4,5}and the collection of sets S= {{1,2,3}, {2,4}, {3,4}, {4,5}}. Clearly the union of S is U. However, we can cover all of the elements with the following, smaller number of sets: {{1,2,3}, {4,5}}. It focuses on criterion 2 which is no box has left behind. Therefore, you may have duplication now but you have to have one of everything of your stuff in order to put the boxes in the truck.

14- **Exact Cover**: Given a collection S of subsets of set X, an exact cover is the subset S\* of S such that each element of X is contained is exactly one subset of S\*. It should satisfy following two conditions –

The **Intersection** of any two subsets in S\* should be empty. That is, each element of X should be contained in at most one subset of S\*

**Union** of all subsets in S\* is X. That means union should contain all the elements in set X. So, we can say that S\* covers X.

Example (standard representation) –  
Let S = {A, B, C, D, E, F} and X = {1, 2, 3, 4, 5, 6, 7} such that –

A = {1, 4, 7}

B = {1, 4}

C = {4, 5, 7}

D = {3, 5, 6}

E = {2, 3, 6 7}

F = {2, 7}

Then S\* = {B, D, F} is an exact cover, because each element in X is contained exactly once in subsets {B, D, F}. If we union subsets then we will get all the elements of X –  
F = {1,2,3,4,5,6,7}.

The Exact cover problem is a decision problem to determine if exact cover exists or not. It focuses on both criterion 2 and 3. So if we consider that truck and box problem then now we have to have one of everything and we have everything.

**Satisfiability**: Satisfiability implies that the problem is hard to solve in the worst case, but certain instances of the problem are not necessarily so tough. Suppose that each clause contains exactly one literal. To satisfy such a clause, we have to appropriately set that literal, so we can repeat this argument for every clause in the problem instance. Only when we have two clauses that directly contradict each other, such as  C ={{ v1},{ v1}} will the set not be satisfiable.

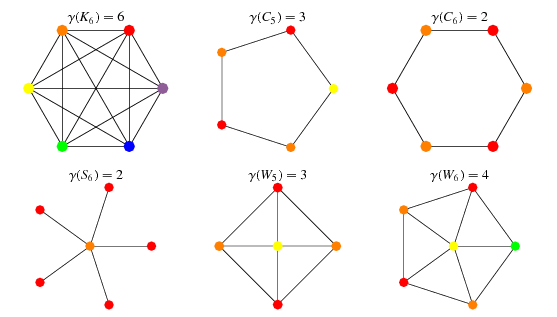
Satisfiability has to do with the **Boolean predicate** either 0 or 1

P: x —> {0,1}

(xi1 V xj1 V xk1) ^ (xi2 V xj2 V xk2) ^ ...

Clause 1 clause 2 …

**Chromatic Number:**



The chromatic number of a graph G is the smallest number of colors needed to color the vertices of G so that no two adjacent vertices share the same color (Skiena 1990, p. 210), the smallest value of K possible to obtain a k-coloring.

Max Cut:



nn



17 + 1 + 4+ 3 + 5 = 30



Given a graph G = (V, E) with non-negative weights wij on the edges, find a set S ⊂ V for which cut(S) is maximal. Goemans and Williamson [GW95] introduced an approximation algorithm that runs in polynomial time and has a randomized component to it, and is able to obtain a cut whose expected value is guaranteed to be no smaller than a particular constant αGW times the optimum cut. The constant αGW is referred to as the approximation ratio. Let V = {1, . . . , n}. In a summary Max-Cut defines as

max )

s.t. ||=1

Note that NP- Complete Problems 7,8,9,10 all of them are cycles and more specifically 9 and 10 are permutations and 7 and 8 are combinations.

And the last thing there will be programming project on np-complete. You could use source code from website, but you must have to attach documentation for that.

**Keywords:** Chromatic Number, Boolean predicate, Satisfiability, Union, intersection, Exact Cover, set covering, Set Packing, mutually exclusive, partition