Final Course Topics

- *Computability*
- Computational Complexity
- DNA Computing
- Quantum Computing
- Review partially done in Lab on Friday

Course evaluations done on Wednesday and in lab on Friday

Computational Complexity

- Programs written during the semester were based on given algorithms
  - often used libraries
  - did not analyze programs with respect to efficiency
  - did some run-time comparisons
- Computational complexity of a problem
  - refers to the time needed to solve the problem in terms of its input size n, independent of the machine used
  - Measured as O(f(n)) – (big-Oh-notation): time is bounded by c*f(n) for some constant c

Problems seen ....

- Finding the closest point of a given point in a set of n points
  - Unsorted list : O(n) time
  - KD-trees: faster, but harder to analyze
- Walking a tree structure consisting of n nodes
  - O(n) time
- Graph problems solved in Project 4
  - With the exception of mcl-clustering, algorithms used use linear time

In an earlier lab you wrote code for mutliplying two n×n matrices

The number of operations (multiplication, additions) done was proportional to

A. n^4

B. n^3

C. n^2

D. n

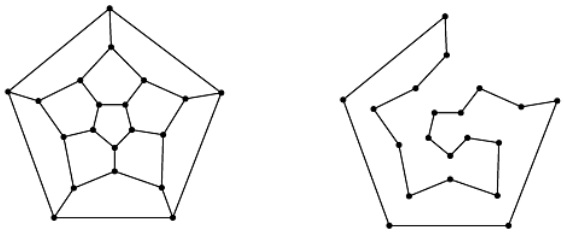
Dictionaries support operations on key-value pairs. Which operations are not provided and would be inefficient to implement?

1. Update the value associated with a given key
2. Delete the entry with the minimum key
3. Determine that a given key is in the dictionary
4. Determine the next largest key in the dictionary

A. **None**
B. **4**
C. **1, 2, and 3**
D. **2 and 4**

Hamiltonian Cycle Problem

Given an n node graph, does there exist a cycle that visits all nodes and every node exactly once?



- Easy to solve in exponential time
- No polynomial time solution is known

Formula Satisfiability

Given a Boolean formula in Conjunctive Normal Form (CNF), does there exist a true/false assignment of the variables making the formula true?

$(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee D) \wedge (C \vee \neg D) \wedge (\neg A \vee B \vee D) \wedge (\neg A \vee \neg C)$

Formula can be satisfied setting A=true, C=false, D=false, B=true)

- Easy to solve in exponential time (check all possible true/false assignments
- No polynomial time solution is known

What do Formula Satisfiability and Hamiltonian Cycle have in common?

1) For both we can *verify* a solution easily:

Given a cycle/a true-false assignment, is it a solution to the problem?

- Is the cycle a Hamiltonian cycle?
- Does the assignment make the formula true?

2) The two problems have been shown to be *equally hard*

- If there exists a polynomial time solution for one, then there exists a polynomial time solution for the other one
  …. as well as for many other problems

How easy is it to verify?

- Given an integer p, verify that it is a composite number (by giving two factors a and b)
- Given an integer p, verify that it is prime (by giving what?)
- Given a graph G, verify that it has no Hamiltonian cycle (by giving what?)

Rule: Verification process needs to run in polynomial time

Classes P, NP, NP-Complete

P = set of all problems that can be solved in polynomial time

NP= set of all problems whose solution can be verified in polynomial time

Open question: Is P=NP?

Problems in NP, but not in P, have a known exponential time solution

NP-complete = set of all problems in NP with the property that a polynomial time solution for them implies a polynomial time solution for all problems in NP

How to deal with NP-complete problems?

- NP-complete problems arise in almost all applications
- Thousands of interesting and relevant problems have been shown to be NP-complete
- They are solved
  - Using heuristic algorithms that generate good, but not optimal solutions (the approach in the bio project was a heuristic)
  - Some problems have heuristics that work quite well, others don't
  - Probabilistic methods (Monte Carlo, Simulated annealing)
  - Looking for special cases that can be solved efficiently

DNA Computing

- Existing genetic engineering tools can slice, create, and rebuild DNA sequences:
  - Custom sequences can be made "to order": We can specify the particular sequences to be created.
  - Duplication, replication, and selection operations exist for DNA sequences.
- Many ($\approx 10^{15}$) different DNA sequences can be operated on at the same time in a single test tube.

How can DNA manipulations lead to computation?

In 1993, Leonard Adleman (USC) proposed DNA computing

- DNA has a 4-letter alphabet:
  - A (Adenine), C (Cytosine), G (Guanine), T (Thymine)
- Computers have a 2-letter alphabet (0 and 1)
- We can synthesize any strand of DNA
- We can generate any binary sequence
- DNA and binary strings
  - encode information
  - we can apply operations like cut, merge, copy, and paste.

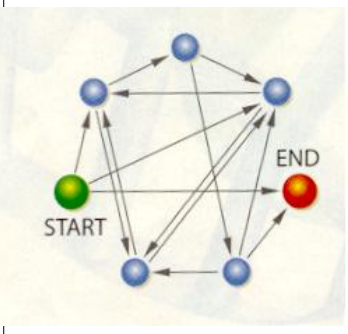
How can DNA strings carry out computation?

DNA strings attach to each other (twist into a double helix) if they have complementary elements in corresponding positions.

- A and T are complements
- C and G are complements

Adleman devised a DNA manipulation technique to solve the Hamiltonian Path (HP) problem.

- DNA sequences were created
- They allowed the execution of trillions of operations in parallel



Adleman's 98 Scientific American paper is very readable

Adleman's first DNA computation operated on a 7-vertex, 14-edge graph.

The algorithm is probabilistic.

- With high probability it will generate a Hamiltonian path, if one exists.

It is a hard-wired, not a programmed solution.

- The setup is for one graph. It needs to be repeated from scratch for another graph.

Demonstrated feasibility and started a new area:
DNA Computing

Overview of the HP DNA algorithm

Given is a directed graph G and vertices s and t,

1. Generate DNA sequences to represent vertices and edges.
2. Generate the DNA strands to represent paths.
3. From among the paths, select a path that:
   - begins at s and ends at t
   - has length n−1
   - is a simple path.

Encoding of vertices and edges

For every vertex v_i:

generate a random DNA strand R_i of length 20

For every edge (v_i, v_j):

create the string S(i,j), which consists of the first half of R_i and the second half of R_j.

Example:

v_2 as R_2 = ATCGAACGTTTTAACGTAGT

v_3 as R_3 = *TCGAATTACG*TAGAACGTTT

edge (v_2,v_3) as TTAACGTAGT*TCGAATTACG*

Initial conditions

Start vertex s has strand R_s and end vertex t has strand R_t

For every edge (s, v_i), create the edge strand consisting of R_s and the first half of R_i

For every edge (v_i, t), create the edge strand consisting of the second half of R_i followed by R_t

Consider a path $< s, v_1, v_2 >$:

R_s = TTCATTCGCCTTAAAGGACT

R_1 = GTAGTACGTT**TCGGCGTAGA**

R_2 = ATCGAACGTATTAACGTAGT

TTCATTCGCCTTAAAGGACTGTAGTACGTT**TCGGCGTAGA**ATCGAACGTATTA …

What goes on in the test tube?

- Complement of the vertex strands – lots of them
- Edge strands – lots of them

DNA strands attach to form double strands if they have complementary elements in the corresponding positions.

R_s = TTCATTCGCCTTAAAGGACT

R_1 = GTAGTACGTTTCGGCGTAGA

R_2 = ATCGAACGTATTAACGTAGT

In the tube:

c(R_s) = AAGTAAGCGGAATTTCCTGA

c(R_1) = CATCATGCAAAGCCGCAACA

c(R_2) = TAGCTTGCATAATTGCATCA

edge (s, v_1): TTCATTCGCCTTAAAGGACTGTAGTACGTT

edge (v_1, v_2): TCGGCGTAGAATCGAACGTA

Reaction:

TTCATTCGCCTTAAAGGACTGTAGTACGTTTCGGCGTAGAATCGAACGTA
AAGTAAGCGGAATTTCCTGACATCATGCAAAGCCGCAACATAGCTTGCATAATTGCATC

Putting it all together

1. Generate DNA sequences representing vertices and edges.
2. Mix to generate the paths.
3. From among the paths, select a path that
   begins at s and ends at t and
   has length n−1 and is a simple path.

Step 3 involves operations like merge, amplify, test-if-empty, separate, separate-by-length, separate-by-positions.

Progress since 1993

Other problems were solved using Adleman's approach (satisfiability, factoring numbers, combinatorial problems with a large solution space). Some solutions were fast.

It was shown that DNA computation does not allow provide additional computational power beyond that of a Turing machine.

Between 2002-04, researchers from the Weizmann Institute of Science (Israel) developed a programmable molecular computing machine composed of enzymes (hardware) and DNA molecules (data).