Recap: Simple vs smart recursion

Simple recursion of fibonacci numbers

```c
int fibonacci(int n) {
    if ( n < 2 ) return n;
    else return fibonacci(n-1)+fibonacci(n-2);
}
```
Top-down dynamic programming

```c
int fibonacci(int n) {
    int* fibs = new int[n+1];
    fibs[0] = 0;
    fibs[1] = 1;
    for(int i=2; i <= n; ++i) {
        fibs[i] = fibs[i-1]+fibs[i-2];
    }
    int ret = fibs[n];
    delete [] fibs;
    return ret;
}
```
int fibonacci(int* fibs, int n) {
    if ( fibs[n] > 0 ) {
        return fibs[n];  // reuse stored solution if available
    }
    else if ( n < 2 ) {
        return n;        // terminal condition
    }
    fibs[n] = fibonacci(n-1) + fibonacci(n-2); // store the solution once computed
    return fibs[n];
}
Recap: The Manhattan tourist problem
Introduction

A "dynamic" structure of the solution

- Let $C(r, c)$ be the optimal cost from $(0, 0)$ to $(r, c)$
- Let $h(r, c)$ be the weight from $(r, c)$ to $(r, c + 1)$
- Let $v(r, c)$ be the weight from $(r, c)$ to $(r + 1, c)$
- We can recursively define the optimal cost as

$$C(r, c) = \begin{cases} 
\min \left\{ C(r - 1, c) + v(r - 1, c), C(r, c - 1) + h(r, c - 1) \right\} & r > 0, c > 0 \\
C(r, c - 1) + h(r, c - 1) & r > 0, c = 0 \\
C(r - 1, c) + v(r - 1, c) & r = 0, c > 0 \\
0 & r = 0, c = 0 
\end{cases}$$

- Once $C(r, c)$ is evaluated, it must be stored to avoid redundant computation.
Recap: Edit distance

![Edit distance diagram]
Today

- Boost library
- Graph algorithms
  - Dijkstra’s algorithm
  - All-pair shortest path
Using boost C++ libraries

Boost C++ library

- An extensive set of libraries for C++
- Supports many additional classes and functions beyond STL
- Useful for increasing productivity
Using boost C++ libraries

Boost C++ library

- An extensive set of libraries for C++
- Supports many additional classes and functions beyond STL
- Useful for increasing productivity

Examples of useful libraries

- Math/Statistical Distributions
- Graph
- Regular expressions
- Tokenizer
Getting started with boost libraries

Download, Compile, and Install

- Follow instructions at
  http://www.boost.org/users/download/

- Note that compile takes a REALLY LONG time - up to hours!

- Everyone should try to install it, and let me know if it does not work for your environment.
Quick boost installation guide

Check whether your system already has boost installed

- Type `ls /usr/include/boost` or `ls /usr/local/include/boost`
- If you get a non-error message, you are in luck!
Quick boost installation guide

Check whether your system already has boost installed

- Type `ls /usr/include/boost` or `ls /usr/local/include/boost`
- If you get a non-error message, you are in luck!

Otherwise, in Linux or MacOS X

```
user@host:~/$ tar xzvf boost_1_45_0.tar.gz
user@host:~/$ cd boost_1_45_0
user@host:~/$ mkdir --p /home/[user]/devel
user@host:~/$ ./bootstrap.sh --prefix=/home/[user]/devel
    (exclude --prefix when you have superuser permission)
user@host:~/$ ./bjam install (or sudo ./bjam install if you are a superuser)
user@host:~/$ g++ -I/home/[user]/devel/include -o boostExample boostExample.cpp
```

In Windows with Visual Studio

http://www.boost.org/doc/libs/1_45_0/more/getting_started/windows.html
#include <iostream>
#include <boost/math/distributions/chi_squared.hpp>

int main(int argc, char** argv) {
    if ( argc != 5 ) {
        std::cerr << "Usage: chisqTest [a] [b] [c] [d]" << std::endl;
        return -1;
    }
    int a = atoi(argv[1]); // read 2x2 table from command line arguments
    int b = atoi(argv[2]);
    int c = atoi(argv[3]);
    int d = atoi(argv[4]);

    // calculate chi-squared statistic and p-value
    double chisq = (double)(a*d-b*c)*(a*d-b*c)*(a+b+c+d)/(a+b)/(c+d)/(a+c)/(b+d);
    boost::math::chi_squared chisqDist(1); // chi-squared statistic
    double p = boost::math::cdf(chisqDist, chisq); // calculate cdf
    std::cout << "Chi-square statistic = " << chisq << std::endl;
    std::cout << "p-value = " << 1-p << std::endl; // output p-value
    return 0;
}
Running examples of chisqTest

user@host~:/$ ./chisqTest 2 7 8 2
Chi-square test statistic = 6.34272
p-value = 0.0117864

user@host~:/$ ./chisqTest 20 70 80 20
Chi-square test statistic = 63.4272
p-value = 1.66533e-15

user@host~:/$ ./chisqTest 200 700 800 200
Chi-square test statistic = 634.272
p-value = 0 (not very robust to small p-values)
#include <iostream>
#include <boost/math/distributions/chi_squared.hpp>
using namespace std;
using namespace boost::math;
int main(int argc, char** argv) {
  ...
  // calculate chi-squared statistic and p-value
  double chisq = (double)((a*d-b*c)*(a*d-b*c)*(a+b+c+d)/(a+b)/(c+d)/(a+c)/(b+d));
  chi_squared chisqDist(1);  // instead of boost::math::chi_squared
  double p = cdf(chisqDist, chisq);  // instead of boost::math::cdf
  cout << "Chi-square statistic = " << chisq << endl;  // instead of std::cout
  cout << "p-value = " << 1-p << endl;  // and std::endl;
  return 0;
}
#include <iostream>
#include <boost/tokenizer.hpp>
#include <string>
using namespace std;
using namespace boost;

int main(int argc, char** argv) {
    // default delimiters are spaces and punctuations
    string s1 = "Hello, boost library";
    tokenizer<> tok1(s1);
    for(tokenizer<>::iterator i=tok1.begin(); i != tok1.end() ; ++i) {
        cout << *i << endl;
    }
    // you can parse csv-like format
    string s2 = "Field 1,""putting quotes around fields, allows commas"",Field 3";
    tokenizer<escaped_list_separator<char> > tok2(s2);
    for(tokenizer<escaped_list_separator<char> >::iterator i=tok2.begin(); i != tok2.end() ; ++i) {
        cout << *i << endl;
    }
    return 0;
}
A running example of tokenizerTest

user@host~:/$ ./tokenizerTest
Hello
boost
library
Field 1
putting quotes around fields, allows commas
Field 3
Introducing graphs

Graph is useful for representing

- Bayesian network
- Biological network
- Dependency between processes
- Phylogenetic tree
Introducing graphs

Graph is useful for representing

- Bayesian network
- Biological network
- Dependency between processes
- Phylogenetic tree

Key components of a graph

- Vertices
- Edges
- Directionality (directed, undirected, bidirectional)
- Vertex properties (e.g. colors)
- Edge properties (e.g. weights)
Algorithmic problems with graphs

- Vertex coloring (k-coloring) problem
  - Minimum number of colors required to color all pairs of adjacent vertices with different colors
  - An $NP$-complete problem - no known polynomial time solution.
Algorithmic problems with graphs

- Vertex coloring (k-coloring) problem
  - Minimum number of colors required to color all pairs of adjacent vertices with different colors
  - An *NP-complete* problem - no known polynomial time solution.

- Traveling salesman problem
  - Determine whether there is a path to visit each vertex exactly once.
  - Another *NP-complete* problem
Algorithmic problems with graphs

- Vertex coloring (k-coloring) problem
  - Minimum number of colors required to color all pairs of adjacent vertices with different colors
  - An $NP$-complete problem - no known polynomial time solution.

- Traveling salesman problem
  - Determine whether there is a path to visit each vertex exactly once.
  - Another $NP$-complete problem

- Shortest path finding problem
  - Find shortest path from a source to destination
  - A polynomial time solution exists
Single-source shortest paths problem

Given

- A directed graph $G = (V, E)$
- With weight function $w : E \to \mathbb{R}$
- $(u, v)$: source and destination vertices.

Want

A path $p = \langle x_0, x_1, \ldots, x_k \rangle$ ($x_0 = u, x_k = v$) whose weight $w(p) = \sum_{i=1}^{k} w(x_{i-1}, x_i)$ is minimum among all possible paths
Shortest path algorithms

- Single-source shortest paths problems
  - Bellman-Ford algorithm: allowing negative weights
    - $\Theta(|V||E|)$ complexity
  - Dijkstra's algorithm: non-negative weights only
    - $\Theta(|V| \log |V| + |E|)$ complexity

- All-pair shortest paths algorithms
  - Floyd-Warshall algorithm
    - $\Theta(|V|^3)$ complexity
Elementary functions

**Algorithm `InitializeSingleSource`**

**Data:** $G$: graph, $s$: source

```plaintext
for $v \in G. V$ do

  $v.d = \infty$;

  $v.\pi = \text{NIL}$;

end

$s.d = 0$;
```

**Algorithm `Relax`**

**Data:** $u$: vertex, $v$: vertex, $w$: weights

```plaintext
if $v.d > u.d + w(u, v)$ then

  $v.d = u.d + w(u, v)$;

  $v.\pi = u$;

end
```
Dijkstra’s algorithm

Algorithm DIJKSTRA

Data: $G$ : graph, $w$ : weight, $s$ : source
Result: Each vertex contains the optimal weight from $s$

INITIALIZESINGLESOURCE($G,s$);
$S = \emptyset$;
$Q = G.V$;
while $Q \neq \emptyset$ do
    $u = \text{EXTRACTMIN}(Q)$;
    $S = S \cup \{u\}$;
    for $v \in G.Adj[u]$ do
        RELAX($u,v,w$);
    end
end
Illustration of Dijkstra’s algorithm
Time complexity of Dijkstra’s algorithm

- The total number of while iteration is $|V|$
- \textsc{ExtractMin} takes $\Theta(\log |Q|) \leq \Theta(\log |V|)$ time
- The total number of for iteration if $|E|$ because \textsc{Relax} is called only once per edge
- The total time complexity is $\Theta(|V| \log |V| + |E|)$. 
Using boost library for Manhattan Tourist Problem

```cpp
int main(int argc, char** argv) {
    // defining a graph type
    // 1. edges are stored as std::list internally
    // 2. verticies are stored as std::vector internally
    // 3. the graph is directed (undirectedS, bidirectionalS can be used)
    // 4. vertices do not carry particular properties
    // 5. edges contains weight property as integer value
    typedef adjacency_list< listS, vecS, directedS, no_property,
                             property< edge_weight_t, int> > graph_t;

    // vertex_descriptor is a type for representing vertices
    typedef graph_traits< graph_t >::vertex_descriptor vertex_descriptor;

    // a nodes is represented as an integer, and an edge is a pair of integers
    typedef std::pair<int, int> E;

    // Connect between verticies as in the Manhattan Tourist Problem
    // Each node is labled as a two-digit integer of [#row] and [#col]
    enum { N11, N12, N13, N14, N15,
           N21, N22, N23, N24, N25,
           N31, N32, N33, N34, N35,
           N41, N42, N43, N44, N45,
           N51, N52, N53, N54, N55 };
Using boost library for Manhattan Tourist Problem

// model edges for Manhattan tourist problem
E edges [] = { E(N11,N12), E(N12,N13), E(N13,N14), E(N14,N15),
             E(N21,N22), E(N22,N23), E(N23,N24), E(N24,N25),
             E(N31,N32), E(N32,N33), E(N33,N34), E(N34,N35),
             E(N41,N42), E(N42,N43), E(N43,N44), E(N44,N45),
             E(N51,N52), E(N52,N53), E(N53,N54), E(N54,N55),
             E(N11,N21), E(N12,N22), E(N13,N23), E(N14,N24), E(N15,N25),
             E(N21,N31), E(N22,N32), E(N23,N33), E(N24,N34), E(N25,N35),
             E(N31,N41), E(N32,N42), E(N33,N43), E(N34,N44), E(N35,N45),
             E(N41,N51), E(N42,N52), E(N43,N53), E(N44,N54), E(N45,N55) };

// Assign weights for each edge
int weight [] = { 4, 2, 0, 7,  // horizontal weights
                 7, 4, 5, 9,
                 6, 8, 1, 0,
                 1, 6, 4, 7,
                 1, 5, 8, 5,
                 0, 6, 6, 2, 4,  // vertical weights
                 9, 7, 1, 0, 6,
                 1, 8, 4, 8, 9,
                 3, 6, 6, 0, 7 };}
Using Dijkstra’s algorithm to solve the MTP

// define a graph as an array of edges and weights
graph_t g(edges, edges + sizeof(edges) / sizeof(E), weight, 25);
// vectors to store predecessors and shortest distances from source
std::vector<vertex_descriptor> p(num_vertices(g));
std::vector<int> d(num_vertices(g));
vertex_descriptor s = vertex(N11, g); // specify source vertex
// Run Dijkstra's algorithm and store paths and distances to p and d
dijkstra_shortest_paths(g, s, predecessor_map(&p[0]).distance_map(&d[0]));

graph_traits<graph_t>::vertex_iterator vi, vend;

std::cout << "Backtracking the optimal path from the destination to source" << std::endl;
for(int node = N55; node != N11; node = p[node]) {
    std::cout << "Path: N" << getNodeID(p[node]) << " -> N"
              << getNodeID(node) << ", Distance from origin is " << d[node] << std::endl;
}

return 0;
The remainder - beginning of DijkstraMTP.cpp

// Note that this code would not work with VC++
#include <iostream> // for input/output
#include <boost/graph/adjacency_list.hpp> // for using graph type
#include <boost/graph/dijkstra_shortest_paths.hpp> // for dijkstra algorithm
using namespace std; // allow to omit prefix 'std::'
using namespace boost; // allow to omit prefix 'boost::'

// converts 0,1,2,3,4,5,6,...,25 to 11,12,13,14,15,21,22,.....,55
int getNodeID(int node) {
    return ((node/5)+1)*10+(node%5+1);
}

int main(int argc, char** argv) {
    ...

Running example of DijkstraMTP

```
user@host~/$ ./DijkstraMTP
Backtracking the optimal path from the destination to source
Path: N54 -> N55, Distance from origin is 21
Path: N44 -> N54, Distance from origin is 16
Path: N34 -> N44, Distance from origin is 16
Path: N24 -> N34, Distance from origin is 8
Path: N14 -> N24, Distance from origin is 8
Path: N13 -> N14, Distance from origin is 6
Path: N12 -> N13, Distance from origin is 6
Path: N11 -> N12, Distance from origin is 4
```
Dijkstra’s algorithm: summary

- An efficient algorithm for shortest-path finding
- Using boost library
- Transformed Manhattan Tourist Problem (simpler) to a shortest-path finding problem (more complex).
Calculating all-pair shortest-path weights

A dynamic programming formulation

Let \( d_{ij}^{(k)} \) be the weight of shortest path from vertex \( i \) to \( j \), for which intermediate vertices are in the set \( \{1, 2, \cdots, k\} \).

\[
  d_{ij}^{(k)} = \begin{cases} 
    w_{ij} & k = 0 \\
    \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & k = 1 
  \end{cases}
\]
**Floyd-Warshall Algorithm**

**Algorithm** `FLOYDWARSHALL`

**Data:** $W: n \times n$ weight matrix

$D^{(0)} = W$;

for $k = 1$ to $n$ do

  for $i = 1$ to $n$ do

    for $j = 1$ to $n$ do

      $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$;

    end

  end

end

return $D^{(n)}$;
Graphs and Statistical Models

- Graphs are useful in modeling dependency between random variables, especially in Bayesian networks
  - Each node represents a random variable
  - A directed edge can represent conditional dependency
  - A undirected edge can represent joint probability distribution.

- Inference in Bayesian network directly correspond to particular graph algorithms

- For example, Viterbi algorithm in Hidden Markov Models (HMMs) is equivalent to represented as Dijkstra’s algorithm.
Next Lecture

- Random numbers
- Hidden Markov Models