Biostatistics 615/815 Lecture 9: Dynamic Programming

Hyun Min Kang

October 4th, 2011

Direct address tables

Direct address table : a constant-time continaer

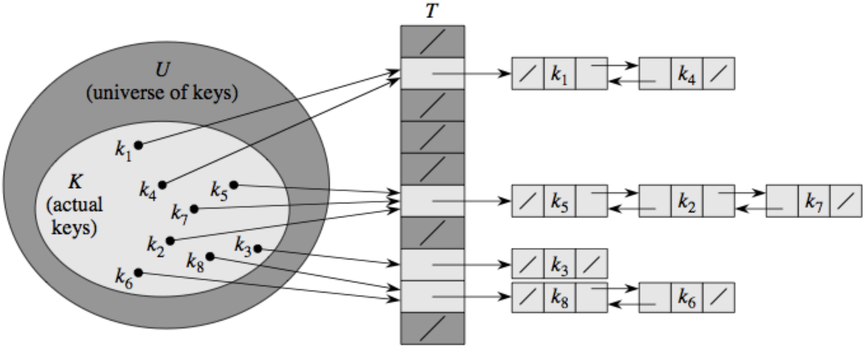
Let $T[0, \cdots, N-1]$ be an array space that can contain N objects

- $\textsc{Insert}(T, x)$: $T[x.key] = x$
- $\textsc{Search}(T, k)$: $\textsc{return}$ $T[k]$
- $\textsc{Remove}(T, x)$: $T[x.key] = \textsc{Nil}$

Time and memory cost

- $O(1)$ - constant time complexity
- Requires to pre-allocate memory space for any possible input value

Recap - Illustration of ChainedHash



Open Addressing

Chained Hash - Pros and Cons

△ Easy to understand

△ Behavior at collision is easy to track

▽ Every slots maintains pointer - extra memory consumption

▽ Inefficient to dereference pointers for each access

▽ Larger and unpredictable memory consumption

Open Addressing

- Store all the elements within an array
- Resolve conflicts based on predefined probing rule.
- Avoid using pointers - faster and more memory efficient.
- Implementation of Remove can be very complicated

Probing in open hash

Modified hash functions

- $h : K \times H \rightarrow H$
- For every $k \in K$, the probe sequence $< h(k, 0), h(k, 1), \cdots, h(k, m-1) >$ must be a permutation of $< 0, 1, \cdots, m-1 >$.

Recap: Divide and conquer algorithms

Good examples of divide and conquer algorithms

- TowerOfHanoi
- MergeSort
- QuickSort
- BinarySearchTree algorithms

These algorithms divide a problem into smaller and disjoint subproblems until they become trivial.

A divide-and-conquer algorithms for Fibonacci numbers

Fibonacci numbers

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & n > 1 \\ 1 & n = 1 \\ 0 & n = 0 \end{cases}$$

A recursive implementation of fibonacci numbers

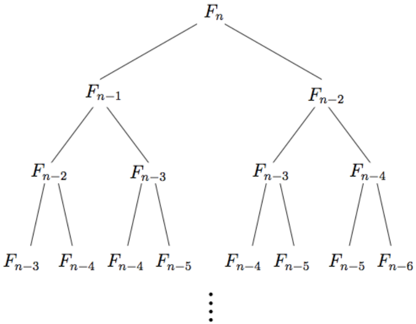
```
int fibonacci(int n) {
  if ( n < 2 ) return n;
  else return fibonacci(n-1)+fibonacci(n-2);
}
```

Performance of recursive FIBONACCI

Computational time

- 4.4 seconds for calculating F_{40}
- 49 seconds for calculating F_{45}
- ∞ seconds for calculating F_{100}!

What is happening in the recursive FIBONACCI



Time complexity of redundant FIBONACCI

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ T(1) &= 1 \\ T(0) &= 1 \\ T(n) &= F_{n+1} \end{aligned}$$

The time complexity is exponential

A non-redundant FIBONACCI

```
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;
}
```

Key idea in non-redundant FIBONACCI

- Each F_n will be reused to calculate F_{n+1} and F_{n+2}
- Store F_n into an array so that we don't have to recalculate it

A recursive, but non-redundant FIBONACCI

```
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];
}
```

Dynamic programming

Key components of dynamic programing

- Problems that can be divided into subproblems
- Overlapping subproblems - subproblems share subsubproblems
- Solves each subsubproblem just once and then saves its answer

Why *dynamic* programming?

According to wikipedia... *"The word 'dynamic' was chosen because it sounded impressive, not because how the method works"*

Examples of dynamic programming

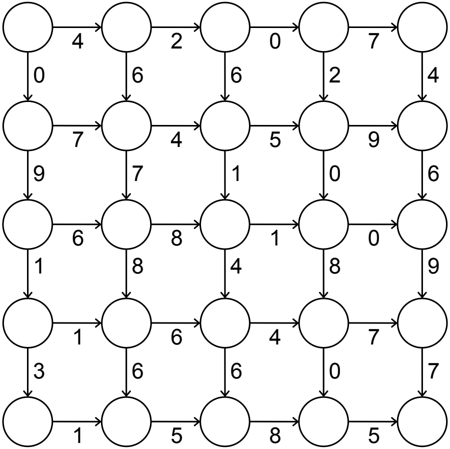
- Shortest path finding algorithms
- DNA sequence alignment
- Hidden markov models

Steps of dynamic programming

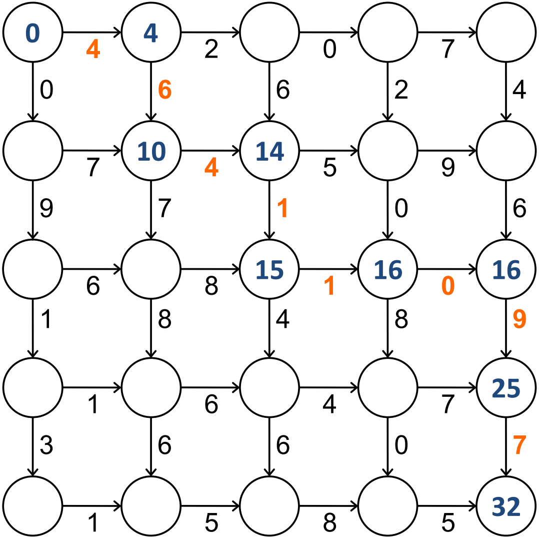
- Characterize the structure of an (optimal) solution
- Recursively define the value of an (optimal) solution
- Compute the value of an (optimal) solution, typically in a bottom-up fashion
- Construct an optimal solution from computed information.

The Manhattan tourist problem

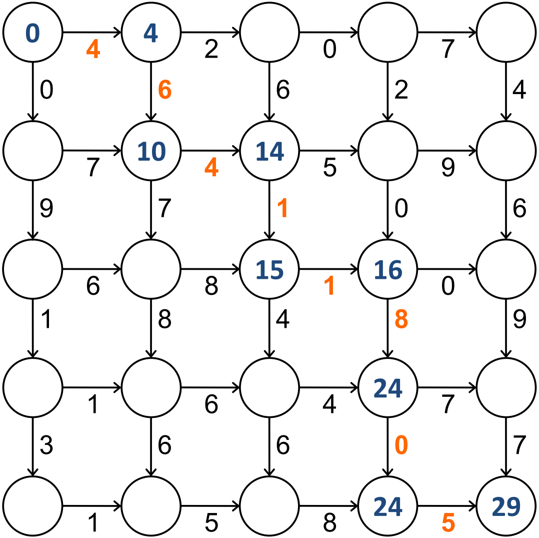
Find the cost-optimal path from left-top corner to right-bottom corner



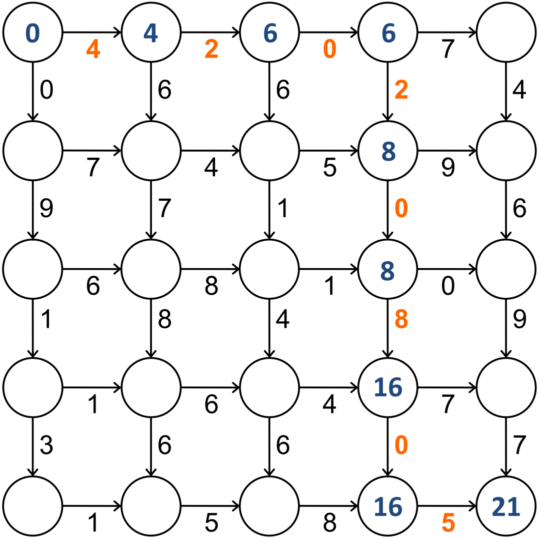
One possible (but not optimal) solution



A slightly better, but still not an optimal solution



And here comes an optimal solution



A brute-force algorithm

Algorithm BruteForceMTP

1. Enumerate all the possible paths
2. Calculate the cost of each possible path
3. Pick the path that produces a minimum cost

Time complexity

- Number of possible paths are $\binom{n_r+n_c}{n_r}$
- Super-exponential growth when n_r and n_c are similar.

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

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

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

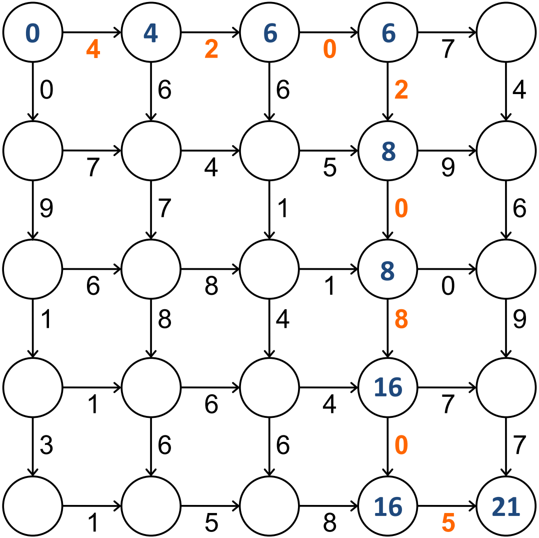
Time complexity of the "dynamic" solution

- Each recursive step takes a constant time
- Each $C(r, c)$ is evaluated at most once.
- Total time complexity is $\Theta(n_r n_c)$.
- Like Fibonacci search, the time complexity would be super exponential if $C(r, c)$ is not stored and redundantly evaluated.

Reconstructing the optimal path

- Optimal cost does not automatically produce optimal path.
- When choosing smaller-cost path between two alternatives, store the decision
- Backtrack from the destination to the source based on the stored decision

Example of backtracking the path



Implementing Manhattan tourist algorithm

```
template <class T>
class Matrix615 {
public:
  std::vector< std::vector<T> > data;
  Matrix615(int nrow, int ncol, T val = 0) {
    data.resize(nrow); // make n rows
    for(int i=0; i < nrow; ++i) {
      data[i].resize(ncol,val); // make n cols with default value val
    }
  }
  int rowNums() { return (int)data.size(); }
  int colNums() { return ( data.size() == 0 ) ? 0 : (int)data[0].size(); }
};
```

Manhattan tourist problem : `main()`

```
int main(int argc, char** argv) {
  int nrows=5, ncols=5;
  // hw stores horizontal weights, vw stores vertical weights
  Matrix615<int> hw(nrows,ncols-1), vw(nrows-1,ncols);

  hw.data[0][0] = 4; hw.data[0][1] = 2; ...
  vw.data[0][0] = 0; vw.data[0][1] = 6; ...

  Matrix615<int> cost(nrows,ncols), move(nrows,ncols);
  // calculate the optimal cost, recording the backtracking info
  int optCost = optimalCost(hw,vw,cost,move,nrows-1,ncols-1);
  std::cout << "Optimal cost is " << optCost << std::endl;
  return 0;
}
```

Calculating optimal cost

```
// hw, vw : horizontal and vertical input weights
// cost : stored optimal cost from (0,0) to (r,c)
// move : stored optimal decision to reach (r,c)
// r,c  : the position of interest
int optimalCost(Matrix615<int>& hw, Matrix615<int>& vw, Matrix615<int>& cost,
  Matrix615<int>& move, int r, int c) {
  // if cost is stored already, skip the cost evaluation
  if ( cost.data[r][c] == 0 ) {
    if ( ( r == 0 ) && ( c == 0 ) ) cost.data[r][c] = 0; // terminal condition
    else if ( r == 0 ) { // only horizontal move is possible
      move.data[r][c] = 0; // 0 means horitontal move to (r,c)
      cost.data[r][c] = optimalCost(hw,vw,cost,move,r,c-1) + hw.data[r][c-1];
    }
    else if ( c == 0 ) { // only vertical move is possible
      move.data[r][c] = 1; // 1 means vertical move to (r,c)
      cost.data[r][c] = optimalCost(hw,vw,cost,move,r-1,c) + vw.data[r-1][c];
    }
```

Calculating optimal cost (cont'd)

```
   else { // evaluate the cumulative cost of horizontal and vertical move
      int hcost = optimalCost(hw,vw,cost,move,r,c-1) + hw.data[r][c-1];
      int vcost = optimalCost(hw,vw,cost,move,r-1,c) + vw.data[r-1][c];
      if ( hcost > vcost ) { // when vertical move is optimal
move.data[r][c] = 1; // store the decision
cost.data[r][c] = vcost; // and store the optimal cost
      }
      else {
move.data[r][c] = 0;
cost.data[r][c] = hcost;
      }
    }
  }

  // when horizontal move is optimal
  return cost.data[r][c]; // return the optimal cost }
}
```

Dynamic programming : A smart recursion

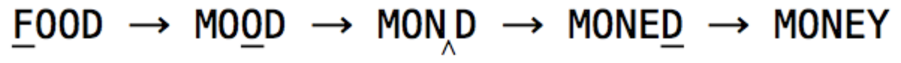
- Dynamic programming is recursion without repetition
  1. Formulate the problem recursively
  2. Build solutions to your recurrence from the bottom up
- Dynamic programming is not about filling in tables; it's about smart recursion (Jeff Erickson)

Minimum edit distance problem

Edit distance

Minimum number of letter insertions, deletions, substitutions required to transform one word into another

An example



$\underline{F}OOD \rightarrow MO\underline{O}D \rightarrow MON_{\wedge}D \rightarrow MONE\underline{D} \rightarrow MONEY$

Edit distance is 4 in the example above

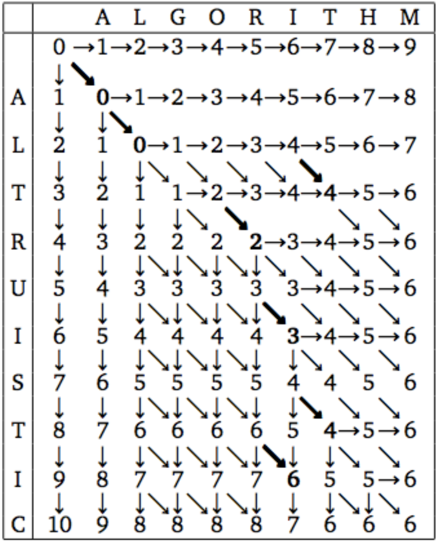
More examples of edit distance

```
F O O   D
M O N E Y
```

```
A L G O R   I   T H M
A L   T R U I S T I C
```

- Similar representation to DNA sequence alignment
- Does the above alignment provides an optimal edit distance?

A dynamic programming solution



		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Recursively formulating the problem

- Input strings are $x[1, \cdots, m]$ and $y[1, \cdots, n]$.
- Let $x_i = x[1, \cdots, i]$ and $y_j = y[1, \cdots, j]$ be substrings of x and y.
- Edit distance $d(x, y)$ can be recursively defined as follows

$$d(x_i, y_j) = \begin{cases} i & j = 0 \\ j & i = 0 \\ \min \left\{ \begin{array}{l} d(x_{i-1}, y_j) + 1 \\ d(x_i, y_{j-1}) + 1 \\ d(x_{i-1}, y_{i-1}) + I(x[i] \neq y[j]) \end{array} \right\} & otherwise \end{cases}$$

- Similar to the Manhattan tourist problem, but with 3-way choice.
- Time complexity is $\Theta(mn)$.

Edit Distance Implementation

```
#include <iostream>
#include <climits>
#include <string>
#include <vector>

template <class T>
class Matrix615 {
public:
  std::vector< std::vector<T> > data;
  Matrix615(int nrow, int ncol, T val = 0) {
    data.resize(nrow); // make n rows
    for(int i=0; i < nrow; ++i) {
      data[i].resize(ncol,val); // make n cols with default value val
    }
  }
  int rowNums() { return (int)data.size(); }
  int colNums() { return ( data.size() == 0 ) ? 0 : (int)data[0].size(); }
};
```

editDistance.cpp: main() function

```
int main(int argc, char** argv) {
  if ( argc != 3 ) {
    std::cerr << "Usage: editDistance [str1] [str2]" << std::endl;
    return -1;
  }
  std::string s1(argv[1]);
  std::string s2(argv[2]);

  Matrix615<int> cost(s1.size()+1, s2.size()+1, INT_MAX);
  Matrix615<int> move(s1.size()+1, s2.size()+1, -1);

  int optDist = editDistance(s1, s2, cost,move, cost.rowNums()-1,
                             cost.colNums()-1);

  std::cout << "EditDistance is " << optDist << std::endl;
  printEdits(s1, s2, move);

  return 0;
}
```

editDistance() algorithm

```
int editDistance(std::string& s1, std::string& s2, Matrix615<int>& cost,
                 Matrix615<int>& move, int r, int c) {
  int iCost = 1, dCost = 1, mCost = 1; // insertion, deletion, mismatch cost

  if ( cost.data[r][c] == INT_MAX ) {
    if ( r == 0 && c == 0 ) { cost.data[r][c] = 0; }
    else if ( r == 0 ) {
      move.data[r][c] = 0;  // only insertion is possible
      cost.data[r][c] = editDistance(s1,s2,cost,move,r,c-1) + iCost;
    }
    else if ( c == 0 ) {
      move.data[r][c] = 1;  // only deletion is possible
      cost.data[r][c] = editDistance(s1,s2,cost,move,r-1,c) + dCost;
    }
```

editDistance() algorithm

```
else {  // compare 3 different possible moves and take the optimal one
  int iDist = editDistance(s1,s2,cost,move,r,c-1) + iCost;
  int dDist = editDistance(s1,s2,cost,move,r-1,c) + dCost;
  int mDist = editDistance(s1,s2,cost,move,r-1,c-1) +
              (s1[r-1] == s2[c-1] ? 0 : mCost);
  if ( iDist < dDist ) {
    if ( iDist < mDist ) { // insertion is optima
      move.data[r][c] = 0;
      cost.data[r][c] = iDist;
    }
    else {
      move.data[r][c] = 2; // match is optimal
      cost.data[r][c] = mDist;
    }
  }
```

editDistance() algorithm

```
      else {
        if ( dDist < mDist ) {
          move.data[r][c] = 1; // deletion is optimal
          cost.data[r][c] = dDist;
        }
        else {
          move.data[r][c] = 2; // match is optimal
          cost.data[r][c] = mDist;
        }
      }
    }
  }
  return cost.data[r][c];
}
```

editDistance.cpp: printEdits()

```
int printEdits(std::string& s1, std::string& s2, Matrix615<int>& move) {
  std::string o1, o2, m;    // output string and alignments
  int r = move.rowNums()-1;
  int c = move.colNums()-1;
  while( r >= 0 && c >= 0 && move.data[r][c] >= 0) { // back from the last character
    if ( move.data[r][c] == 0 ) {  // insertion
      o1 = "-" + o1;  o2 = s2[c-1] + o2;  m = "I" + m;
      --c;
    }
    else if ( move.data[r][c] == 1 ) { // delettion
      o1 = s1[r-1] + o1;  o2 = "-" + o2;  m = "D" + m;
      --r;
    }
    else if ( move.data[r][c] == 2 ) { // match or mismatch
      o1 = s1[r-1] + o1; o2 = s2[c-1] + o2;
      m = (s1[r-1] == s2[c-1] ? "-" : "*") + m;
      --r; --c;
    }
    else std::cout << r << " " << c << " " << move.data[r][c] << std::endl;
  }
  std::cout << m << std::endl << o1 << std::endl << o2 << std::endl;
}
```

Running example

```
$ ./editDistance FOOD MONEY
EditDistance is 4
*-I**
FO-OD
MONEY
```

Summary

Today

- Dynamic programming is a smart recursion avoiding redundancy
- Divide a problem into subproblems that can be shared
- Examples of dynamic programming
    - Fibonacci numbers
    - Manhattan tourist problem
    - Edit distance problem

Next lecture

- Edit Distance
- Introduction to Hidden Markov model