CSE 326: Data Structures
Lecture #20
Problem Solving

Bart Niswonger

Summer Quarter 2001

Today's Outline

- Algorithm Design
  - Dynamic Programming
  - Randomized
  - Backtracking

Dynamic Programming (Memoizing)

- Define problem in terms of smaller subproblems
- Solve and record solution for base cases
- Build solutions for subproblems up from solutions to smaller subproblems

Can improve runtime of divide & conquer algorithms that have shared subproblems with *optimal substructure*.

Usually involves a table of subproblem solutions.

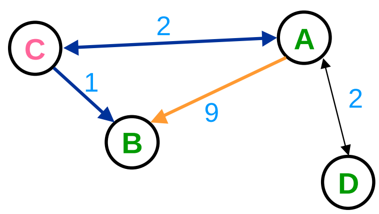
Dynamic Programming in Action

- Sequence Alignment
- Optimal Binary Search Tree
- Fibonacci numbers
- Many, many optimization problems
  - Databases: finding the optimal way to answer a query
  - Workflow: the optimal order of operations to construct some complex object
- *All* pairs shortest path

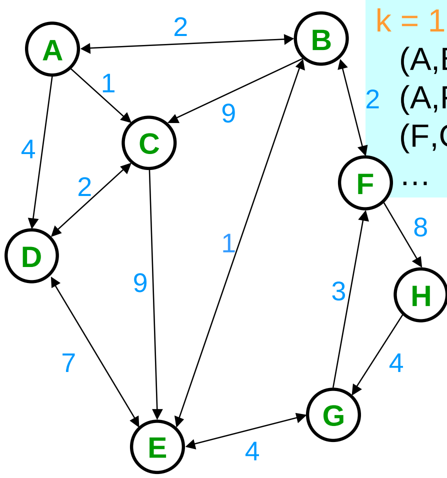
Recursive All-Pairs Shortest Path

Observation:

- The shortest path from A to B is either
  - Non-existent (if the graph is not connected)
  - Direct
  - The shortest path from A to some node n plus the shortest path from n to B



Idea



k = 1	k = 2	k = 3
(A,B) 2	(A,B) 2	(A,B) 2
(A,F) x	(A,F) 4	(A,F) 4
(F,G) x	(F,G) 12	(F,G) 7
…	…	…

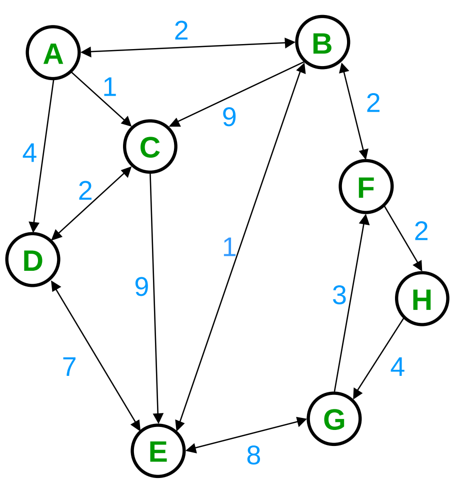
d(i,j,1) = cost of edge(i,j)

d(i,j,k) = min(d(i,j,k – 1),
  d(i,n,k – 1) + d(n,j,k – 1))

Pseudocode

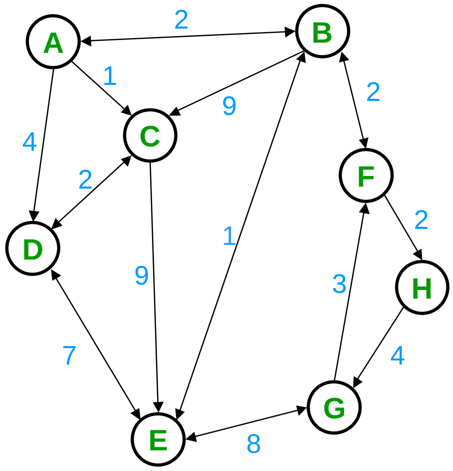
```
int dist( node* i, node *j, int k)
  int distance;
  if ( k <= 1 ) distance = weight(i,j)
  else {
    distance = dist(i,j,k-1)
    foreach node n st path(i,n,k-1) & path(n,j,k-1) {
      i2n2jDistance = dist(i,n,k-1) + dist(n,j,k-1)
      if ( distance < i2n2jDistance )
      distance = i2n2jDistance
    }
  }
  return distance
```

Floyd-Warshall



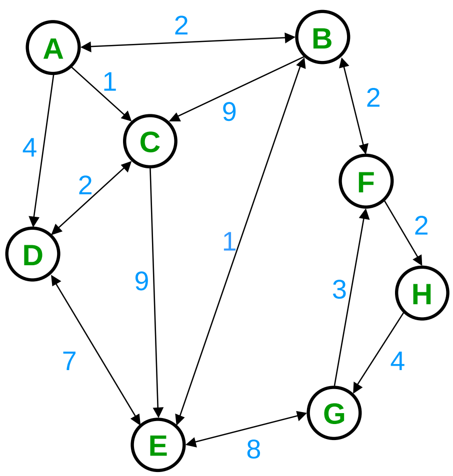
	A	B	C	D	E	F	G	H
A	0	2	1	4				
B	2	0	9		1	2		
C			0	2	9			
D			2	0	7			
E		1		7	0		8	
F		2				0		2
G					8	3	0	
H							4	0

Floyd-Warshall



	A	B	C	D	E	F	G	H
A	0	2	1	4				
B	2	0	9		1	2		
C			0	2	9			
D			2	0	7			
E		1		7	0		8	
F		2				0		2
G					8	3	0	
H							4	0

Floyd-Warshall



	A	B	C	D	E	F	G	H
A		A	A	A				
B	B		B		B	B		
C				C	C			
D			D		D			
E		E		E			E	
F		F						F
G					G	G		
H							G	

Backtracking (a.k.a. Systematic Search)

1. Incrementally establish a solution
2. If complete solution is constructed, succeed!
3. If solution fails, roll back and alter recent choices

- Usually *asymptotically no better* than brute force.
- Key to success is pruning the search space.
- Key to pruning is domain knowledge and learning!

Backtracking in Action

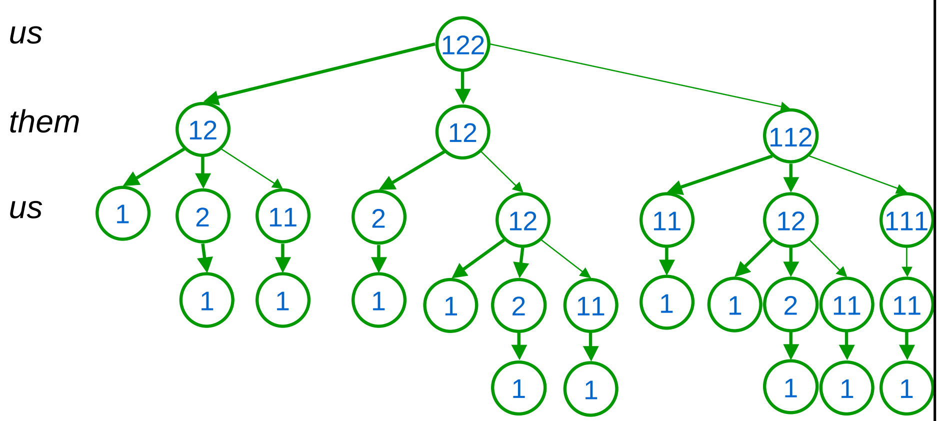
- Depth First Search
- DPLL: Satisfiability Solving
- α-β Search (Game Search)

Game Search

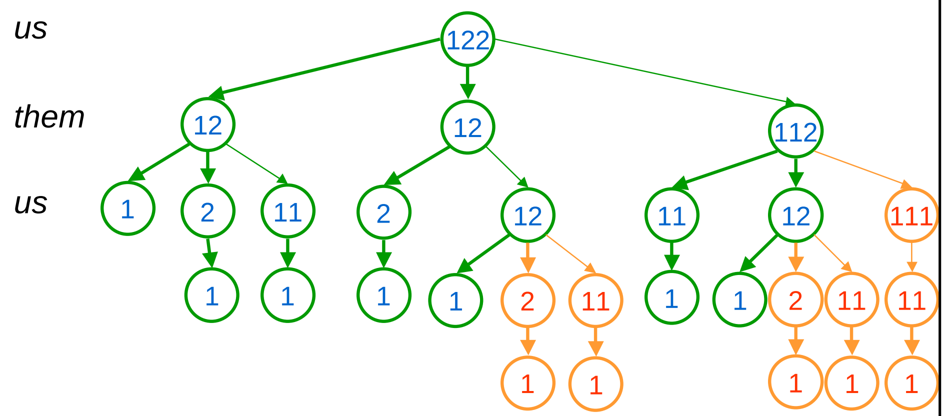
- Search space is composed of board configurations
- Transitions are moves
- Levels alternate between *us* and *them*
- We can evaluate any given board configuration according to a scoring heuristic

How should we decide the next move?

Backtracking Game Search (MiniMax)



α-β Pruned Game Search



Randomized Algorithms

- Define a property (or subroutine) in an algorithm
- Sample or randomly modify the property
- Use altered property as if it were the true property

Can transform average case runtimes into expected runtimes (remove input dependency)

Sometimes allows substantial speedup in exchange for probabilistic unsoundness

Randomization in Action

- Treaps
- Quicksort
- Randomized back-off
- Primality testing

Properties of Primes

P is a prime **0 < A < P** and **0 < X < P**

Then:

$$A^{P-1} = 1 \pmod{P}$$

And, the only solutions to $X^2 = 1 \pmod{P}$ are:

$X = 1$ and $X = P - 1$

Calculating Powers

```
HugeInt pow(HugeInt x, HugeInt n, HugeInt modulo)
{
  if (n == 0)
    return 1;
  if (n == 1)
    return x;                          // If 1 < x < modulo - 1
  HugeInt squared = x * x % modulo;    // but squared == 1,
  if (isEven(n))                       // then modulo isn't prime!
    return pow(squared, n / 2, modulo);
  else
    return (pow(squared, n/2, modulo) * x) % modulo;
}
```

Checking Primality

Systematic algorithm:

- For prime P, for all A such that 0 < A < P
- Calculate A^{P-1} mod P using pow
- Check at each step of pow and at end for primality conditions

Randomized algorithm: use just one random A

If the randomized algorithm reports failure, then P really isn't prime.

If the randomized algorithm reports success, then P *might be* prime.

- P is prime with probability > ¾
- Each new A has independent probability of false positive

Evaluating Randomized Primality Testing

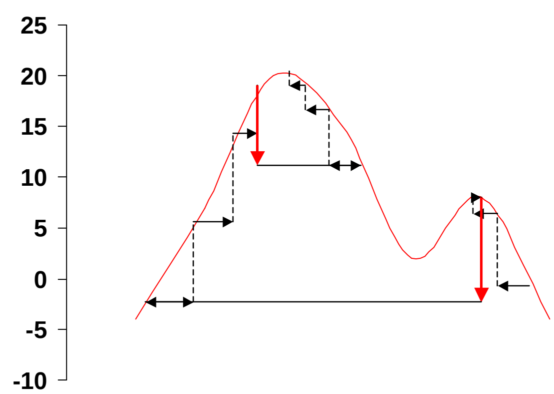
Your probability of being struck by lightning this year: 0.00004%

Your probability that a number that tests prime 11 times in a row is actually not prime: 0.00003%

Your probability of winning a lottery of 1 million people five times in a row: 1 in 2^{100}

Your probability that a number that tests prime 50 times in a row is actually not prime: 1 in 2^{100}

Randomized Greedy Algorithms: Simulated Annealing



Randomized Backtracking: Heavy-Tailed Distributions

Some backtracking algorithms have a few (fruitless) branches that are very large, both deep and broad.

Algorithms which choose randomly at a split point will have a small probability of getting caught in one of these branches.

Therefore, some runs finish very quickly, most runs take some time, and a few runs take orders of magnitude more time than the median.

Solution: cut off long runs and reseed the randomizer.

To Do

- Project IV
  - Create a fearsome runner strategy… and implement it!
- Finish reading Chapter 10
- Start reading Chapter 12
- Study for the final!
- Come to the movie TONIGHT

Coming Up

- “Advanced” Data Structures
- Final – Friday, one week!
- Movie!! (& pizza)