Introduction

Jaehyun Park

CS 97SI
Stanford University

January 10, 2015

Welcome to CS 97SI

- Introduction
- Programming Contests
- How to Practice
- Problem Solving Examples
- Grading Policy

Coaches

- Officially: Jerry Cain
- Actually: Jaehyun Park

Why Do Programming Contests?

- You can learn:
  - Many useful algorithms, mathematical insights
  - How to code/debug quickly and accurately
  - How to work in a team
- Then you can rock in classes, job interviews, etc.
- It's also fun!

Prerequisites

- CS 106 level programming experience
  - You'll be coding in either C/C++ or Java
- Good mathematical insight
- Most importantly, eagerness to learn

Topics

1. Introduction
2. Mathematics
3. Data structures
4. Dynamic programming (DP)
5. Combinatorial games
6. Graph algorithms
7. Shortest distance problems
8. Network flow
9. Geometric algorithms
10. String algorithms

Programming Contests

- Stanford Local Programming Contest
- ACM-ICPC
  - Pacific Northwest Regional
  - World Finals
- Online Contests
  - TopCoder, Codeforces
  - Google Code Jam
- And many more...

How to Practice

- USACO Training Program
- Online Judges
- Weekly Practice Contests

USACO Training Program

- `http://ace.delos.com/usacogate`
- Detailed explanation on basic algorithms, problem solving strategies
- Good problems
- Automated judge system

Online Judges

- Websites with automated judges
  - Real contest problems
  - Immediate feedback
- A few good OJs:
  - Codeforces
  - TopCoder
  - Peking OJ
  - Sphere OJ
  - UVa OJ

Weekly Practice Contests

- Every Saturday 11am-4pm at Gates B08
  - Free food!
- Open to anyone interested
- Real contest problems from many sources
- Subscribe to the `stanford-acm-icpc` email list to get announcements

Example

1. Read the problem statement
   - Check the input/output specification!
2. Make the problem abstract
3. Design an algorithm
   - Often the hardest step
4. Implement and debug
5. Submit
6. AC!
   - If not, go back to 4

Problem Solving Example

- POJ 1000: A+B Problem
  - Input: Two space-separated integers a, b
  - Constraints: $0 \leq a, b \leq 10$
  - Output: $a + b$

POJ 1000 Code in C/C++

```
#include<stdio.h>
int main()
{
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d\n", a + b);
    return 0;
}
```

Another Example

- POJ 1004: Financial Management
  - Input: 12 floating point numbers on separate lines
  - Output: Average of the given numbers
- Just a few more bytes than POJ 1000...

POJ 1004 Code in C/C++

```
#include<stdio.h>
int main()
{
    double sum = 0, buf;
    for(int i = 0; i < 12; i++) {
        scanf("%lf", &buf);
        sum += buf;
    }
    printf("$%.2lf\n", sum / 12.0);
    return 0;
}
```

Something to think about

- What if the given numbers are HUGE?
- Not all the input constraints are explicit
  - Hidden constraints are generally "reasonable"
- Always think about the worst case scenario, edge cases, etc.

Grading Policy

- You can either:
  - Solve a given number of POJ problems on the course webpage
  - OR, participate in 5 or more weekly practice contests
- If you have little experience, solving POJ problems is recommended
  - Of course, doing both of them is better

Stanford ACM Team Notebook

- `http://stanford.edu/~liszt90/acm/notebook.html`
- Implementations of many algorithms we'll learn
- Policy on notebook usage:
  - Don't copy-paste anything from the notebook!
  - At least type everything yourself
  - Let me know of any error or suggestion

Links

- Course website: `http://cs97si.stanford.edu`
- Stanford ACM Team Notebook: `http://stanford.edu/~liszt90/acm/notebook.html`
- Peking Online Judge: `http://poj.org`
- USACO Training Gate: `http://ace.delos.com/usacogate`
- Online discussion board: `http://piazza.com/class#winter2012/cs97si/`

Mathematics

Jaehyun Park

CS 97SI
Stanford University

January 10, 2015

Outline

Sum of Powers

$$\sum_{k=1}^{n} k^2 = \frac{1}{6}n(n+1)(2n+1)$$

$$\sum k^3 = \left(\sum k\right)^2 = \left(\frac{1}{2}n(n+1)\right)^2$$

- Pretty useful in many random situations
- Memorize above!

Fast Exponentiation

- Recursive computation of a^n:

$$a^n = \begin{cases} 1 & n = 0 \\ a & n = 1 \\ (a^{n/2})^2 & n \text{ is even} \\ a(a^{(n-1)/2})^2 & n \text{ is odd} \end{cases}$$

Implementation (recursive)

```
double pow(double a, int n) {
    if(n == 0) return 1;
    if(n == 1) return a;
    double t = pow(a, n/2);
    return t * t * pow(a, n%2);
}
```

- Running time: $O(\log n)$

Implementation (non-recursive)

```
double pow(double a, int n) {
    double ret = 1;
    while(n) {
        if(n%2 == 1) ret *= a;
        a *= a; n /= 2;
    }
    return ret;
}
```

- You should understand how it works

Linear Algebra

- Solve a system of linear equations
- Invert a matrix
- Find the rank of a matrix
- Compute the determinant of a matrix
- All of the above can be done with Gaussian elimination

Outline

Algebra

Number Theory

Combinatorics

Geometry

Greatest Common Divisor (GCD)

- $\gcd(a,b)$: greatest integer divides both a and b
- Used very frequently in number theoretical problems
- Some facts:
  - $\gcd(a,b) = \gcd(a, b-a)$
  - $\gcd(a,0) = a$
  - $\gcd(a,b)$ is the smallest positive number in $\{ax + by \mid x, y \in \mathbf{Z}\}$

Euclidean Algorithm

- Repeated use of $\gcd(a, b) = \gcd(a, b - a)$
- Example:

$$\begin{aligned}
\gcd(1989, 867) &= \gcd(1989 - 2 \times 867, 867) \\
&= \gcd(255, 867) \\
&= \gcd(255, 867 - 3 \times 255) \\
&= \gcd(255, 102) \\
&= \gcd(255 - 2 \times 102, 102) \\
&= \gcd(51, 102) \\
&= \gcd(51, 102 - 2 \times 51) \\
&= \gcd(51, 0) \\
&= 51
\end{aligned}$$

Implementation

```
int gcd(int a, int b) {
    while(b){int r = a % b; a = b; b = r;}
    return a;
}
```

- Running time: $O(\log(a + b))$
- Be careful: `a % b` follows the sign of `a`
  - `5 % 3 == 2`
  - `-5 % 3 == -2`

Congruence & Modulo Operation

- $x \equiv y \pmod{n}$ means x and y have the same remainder when divided by n
- Multiplicative inverse
  - x^{-1} is the inverse of x modulo n if $xx^{-1} \equiv 1 \pmod{n}$
  - $5^{-1} \equiv 3 \pmod{7}$ because $5 \cdot 3 \equiv 15 \equiv 1 \pmod{7}$
  - May not exist (*e.g.*, inverse of 2 mod 4)
  - Exists if and only if $\gcd(x, n) = 1$

Multiplicative Inverse

- All intermediate numbers computed by Euclidean algorithm are integer combinations of a and b
  - Therefore, $\gcd(a, b) = ax + by$ for some integers x, y
  - If $\gcd(a, n) = 1$, then $ax + ny = 1$ for some x, y
  - Taking modulo n gives $ax \equiv 1 \pmod{n}$
- We will be done if we can find such x and y

Extended Euclidean Algorithm

- Main idea: keep the original algorithm, but write all intermediate numbers as integer combinations of a and b
- Exercise: implementation!

Chinese Remainder Theorem

- Given a, b, m, n with $\gcd(m, n) = 1$
- Find x with $x \equiv a \pmod{m}$ and $x \equiv b \pmod{n}$
- Solution:
  - Let n^{-1} be the inverse of n modulo m
  - Let m^{-1} be the inverse of m modulo n
  - Set $x = ann^{-1} + bmm^{-1}$ (check this yourself)
- Extension: solving for more simultaneous equations

Outline

Algebra

Number Theory

Combinatorics

Geometry

Binomial Coefficients

- $\binom{n}{k}$ is the number of ways to choose k objects out of n distinguishable objects
- same as the coefficient of $x^k y^{n-k}$ in the expansion of $(x+y)^n$
  - Hence the name "binomial coefficients"
- Appears everywhere in combinatorics

Computing Binomial Coefficients

- Solution 1: Compute using the following formula:

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!}$$

- Solution 2: Use Pascal's triangle
- Can use either if both n and k are small
- Use Solution 1 carefully if n is big, but k or $n-k$ is small

Fibonacci Sequence

- Definition:
  - $F_0 = 0$, $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$, where $n \geq 2$
- Appears in many different contexts

Closed Form

- $F_n = (1/\sqrt{5})(\varphi^n - \overline{\varphi}^n)$
  - $\varphi = (1 + \sqrt{5})/2$
  - $\overline{\varphi} = (1 - \sqrt{5})/2$
- Bad because φ and $\sqrt{5}$ are irrational
- Cannot compute the exact value of F_n for large n
- There is a more stable way to compute F_n
  - ... and any other recurrence of a similar form

Better "Closed" Form

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

- Use fast exponentiation to compute the matrix power
- Can be extended to support any linear recurrence with constant coefficients

Outline

Geometry

- In theory: not that hard
- In programming contests: more difficult than it looks
- Will cover basic stuff today
  - Computational geometry in week 9

When Solving Geometry Problems

- Precision, precision, precision!
  - If possible, don't use floating-point numbers
  - If you have to, always use `double` and never use `float`
  - Avoid division whenever possible
  - Introduce small constant ϵ in (in)equality tests
    - e.g., Instead of `if(x == 0)`, write `if(abs(x) < EPS)`
- No hacks!
  - In most cases, randomization, probabilistic methods, small perturbations won't help

2D Vector Operations

- Have a vector (x, y)
- Norm (distance from the origin): $\sqrt{x^2 + y^2}$
- Counterclockwise rotation by θ:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

  - Make sure to use correct units (degrees, radians)
- Normal vectors: $(y, -x)$ and $(-y, x)$
- Memorize all of them!

Line-Line Intersection

- Have two lines: $ax + by + c = 0$ and $dx + ey + f = 0$
- Write in matrix form:

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = -\begin{bmatrix} c \\ f \end{bmatrix}$$

- Left-multiply by matrix inverse

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix}^{-1} = \frac{1}{ae - bd} \begin{bmatrix} e & -b \\ -d & a \end{bmatrix}$$

  - Memorize this!
- Edge case: $ae = bd$
  - The lines coincide or are parallel

Circumcircle of a Triangle

- Have three points A, B, C
- Want to compute P that is equidistance from A, B, C
- Don't try to solve the system of quadratic equations!
- Instead, do the following:
  - Find the (equations of the) bisectors of AB and BC
  - Compute their intersection

Area of a Triangle

- Have three points A, B, C
- Want to compute the area S of triangle ABC
- Use cross product: $2S = |(B - A) \times (C - A)|$
- Cross product:

$$(x_1, y_1) \times (x_2, y_2) = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1 y_2 - x_2 y_1$$

  – Very important in computational geometry. Memorize!

Area of a Simple Polygon

- Given vertices $P_1, P_2, \ldots, P_n$ of polygon P
- Want to compute the area S of P
- If P is convex, we can decompose P into triangles:

$$2S = \left| \sum_{i=2}^{n-1} (P_{i+1} - P_1) \times (P_i - P_1) \right|$$

- It turns out that the formula above works for non-convex polygons too
  - Area is the absolute value of the sum of "signed area"
- Alternative formula (with $x_{n+1} = x_1, y_{n+1} = y_1$):

$$2S = \left| \sum_{i=1}^{n} (x_i y_{i+1} - x_{i+1} y_i) \right|$$

Conclusion

- No need to look for one-line closed form solutions
- Knowing "how to compute" (algorithms) is good enough
- Have fun with the exercise problems
  - ... and come to the practice contest if you can!

Data Structures

Jaehyun Park

CS 97SI
Stanford University

January 10, 2015

Typical Quarter at Stanford

```
void quarter() {
    while(true) { // no break :(
        task x = GetNextTask(tasks);
        process(x);
        // new tasks may enter
    }
}
```

- GetNextTask() decides the order of the tasks

Deciding the Order of the Tasks

- Possible behaviors of GetNextTask():
  - Returns the newest task (stack)
  - Returns the oldest task (queue)
  - Returns the most urgent task (priority queue)
  - Returns the easiest task (priority queue)

- `GetNextTask()` should run fast
  - We do this by storing the tasks in a clever way

Outline

Stack

- Last in, first out (LIFO)
- Supports three constant-time operations
  - `Push(x)`: inserts `x` into the stack
  - `Pop()`: removes the newest item
  - `Top()`: returns the newest item
- Very easy to implement using an array

Stack Implementation

- Have a large enough array `s[]` and a counter `k`, which starts at zero
  - `Push(x)`: set `s[k] = x` and increment `k` by 1
  - `Pop()`: decrement `k` by 1
  - `Top()`: returns `s[k - 1]` (error if `k` is zero)
- C++ and Java have implementations of stack
  - `stack` (C++), `Stack` (Java)
- But you should be able to implement it from scratch

Queue

- First in, first out (FIFO)
- Supports three constant-time operations
  - `Enqueue(x)`: inserts `x` into the queue
  - `Dequeue()`: removes the oldest item
  - `Front()`: returns the oldest item
- Implementation is similar to that of stack

Queue Implementation

- Assume that you know the total number of elements that enter the queue
  - ... which allows you to use an array for implementation
- Maintain two indices `head` and `tail`
  - `Dequeue()` increments `head`
  - `Enqueue()` increments `tail`
  - Use the value of `tail - head` to check emptiness
- You can use `queue` (C++) and `Queue` (Java)

Outline

Priority Queue

- Each element in a PQ has a priority value
- Three operations:
  - `Insert(x, p)`: inserts `x` into the PQ, whose priority is `p`
  - `RemoveTop()`: removes the element with the highest priority
  - `Top()`: returns the element with the highest priority
- All operations can be done quickly if implemented using a heap
- `priority_queue` (C++), `PriorityQueue` (Java)

Heap

- Complete binary tree with the heap property:
  - The value of a node $\geq$ values of its children
- The root node has the maximum value
  - Constant-time `top()` operation
- Inserting/removing a node can be done in $O(\log n)$ time without breaking the heap property
  - May need rearrangement of some nodes

Heap Example

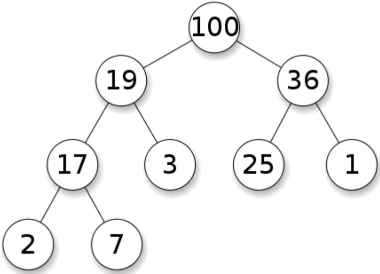
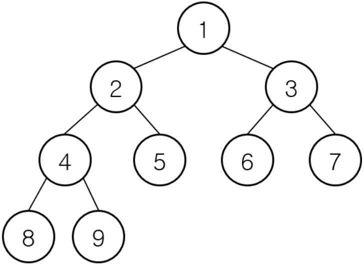


Figure from Wikipedia

Indexing the Nodes



- Start from the root, number the nodes $1, 2, \ldots$ from left to right
- Given a node k easy to compute the indices of its parent and children
  - Parent node: $\lfloor k/2 \rfloor$
  - Children: $2k, 2k+1$

Inserting a Node

1. Make a new node in the last level, as far left as possible
   – If the last level is full, make a new one
2. If the new node breaks the heap property, swap with its parent node
   – The new node moves up the tree, which may introduce another conflict
3. Repeat 2 until all conflicts are resolved

- Running time = tree height = $O(\log n)$

Implementation: Node Insertion

- Inserting a new node with value v into a heap H

```
void InsertNode(int v) {
    H[++n] = v;
    for(int k = n; k > 1; k /= 2) {
        if(H[k] > H[k / 2])
            swap(H[k], H[k / 2]);
        else break;
    }
}
```

Deleting the Root Node

1. Remove the root, and bring the last node (rightmost node in the last level) to the root
2. If the root breaks the heap property, look at its children and swap it with the larger one
   - Swapping can introduce another conflict
3. Repeat 2 until all conflicts are resolved

- Running time $= O(\log n)$
- Exercise: implementation
  - Some edge cases to consider

Outline

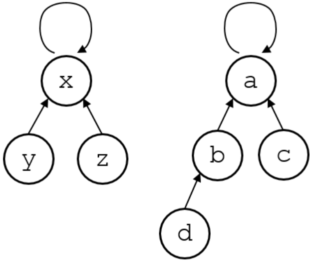
Union-Find Structure

- Used to store disjoint sets
- Can support two types of operations efficiently
  - `Find(x)`: returns the "representative" of the set that `x` belongs
  - `Union(x, y)`: merges two sets that contain `x` and `y`

- Both operations can be done in (essentially) constant time
- Super-short implementation!

Union-Find Structure

- Main idea: represent each set by a rooted tree
  - Every node maintains a link to its parent
  - A root node is the "representative" of the corresponding set
  - Example: two sets $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ and $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$



Implementation Idea

- `Find(x)`: follow the links from `x` until a node points itself
  - This can take $O(n)$ time but we will make it faster

- `Union(x, y)`: run `Find(x)` and `Find(y)` to find corresponding root nodes and direct one to the other

Implementation

- We will assume that the links are stored in `L[]`

```
int Find(int x) {
    while(x != L[x]) x = L[x];
    return x;
}
void Union(int x, int y) {
    L[Find(x)] = Find(y);
}
```

Path Compression

- In a bad case, the trees can become too deep
  - ... which slows down future operations
- Path compression makes the trees shallower every time `Find()` is called
- We don't care how a tree looks like as long as the root stays the same
  - After `Find(x)` returns the root, backtrack to `x` and reroute all the links to the root

Path Compression Implementations

```
int Find(int x) {
    if(x == L[x]) return x;
    int root = Find(L[x]);
    L[x] = root;
    return root;
}

int Find(int x) {
    return x == L[x] ? x : L[x] = Find(L[x]);
}
```

Outline

Binary Search Tree (BST)

- A binary tree with the following property: for each node ,
  - value of $v \geq$ values in v's left subtree
  - value of $v \leq$ dvalues in v's right subtree

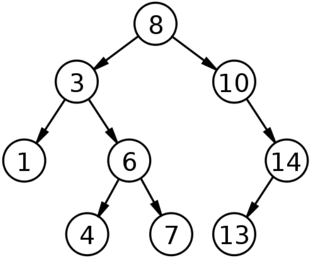


Figure from Wikipedia

What BSTs can do

- Supports three operations
  - `Insert(x)`: inserts a node with value `x`
  - `Delete(x)`: deletes a node with value `x`, if there is any
  - `Find(x)`: returns the node with value `x`, if there is any
- Many extensions are possible
  - `Count(x)`: counts the number of nodes with value less than or equal to `x`
  - `GetNext(x)`: returns the smallest node with value $\geq$ `x`

BSTs in Programming Contests

- Simple implementation cannot guarantee efficiency
  - In worst case, tree height becomes n (which makes BST useless)
  - Guaranteeing $O(\log n)$ running time per operation requires balancing of the tree (hard to implement)
  - We will skip the implementation details
- Use the standard library implementations
  - `set`, `map` (C++)
  - `TreeSet`, `TreeMap` (Java)

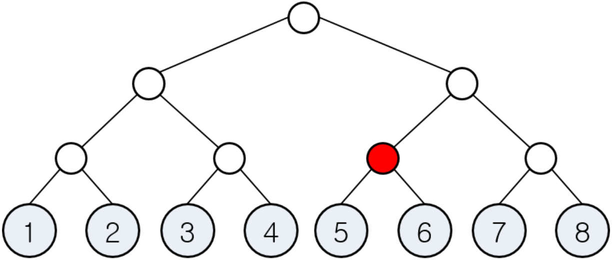
Outline

Fenwick Tree

- A variant of segment trees
- Supports very useful interval operations
  - `Set(k, x)`: sets the value of `k`th item equal to `x`
  - `Sum(k)`: computes the sum of items $1, \ldots, \texttt{k}$ (prefix sum)
    - Note: sum of items $\texttt{i}, \ldots, \texttt{j} = \texttt{Sum}(\texttt{j}) - \texttt{Sum}(\texttt{i} - 1)$
- Both operations can be done in $O(\log n)$ time using $O(n)$ space

Fenwick Tree Structure

- Full binary tree with at least n leaf nodes
  - We will use $n = 8$ for our example
- kth leaf node stores the value of item k
- Each internal node stores the sum of values of its children
  - *e.g.*, Red node stores `item[5]` + `item[6]`

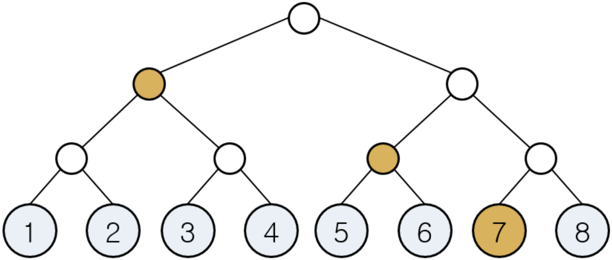


Summing Consecutive Values

- Main idea: choose the minimal set of nodes whose sum gives the desired value
- We will see that
  - at most 1 node is chosen at each level so that the total number of nodes we look at is $\log_2 n$
  - and this can be done in $O(\log n)$ time
- Let's start with some examples

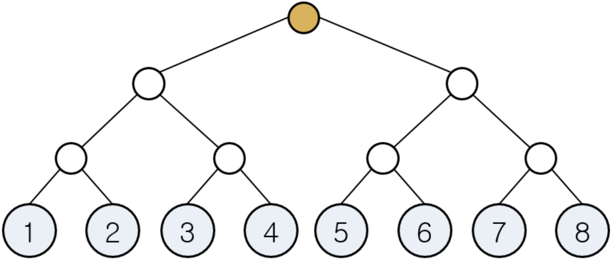
Summing: Example 1

- $\texttt{Sum}(7)$ = sum of the values of gold-colored nodes



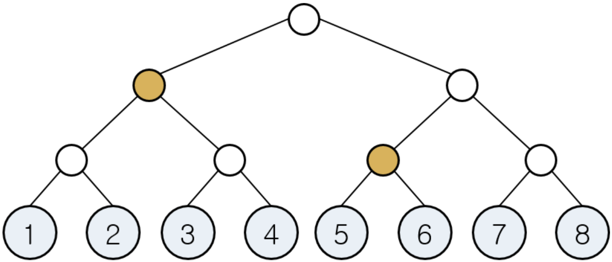
Summing: Example 2

- `Sum(8)`



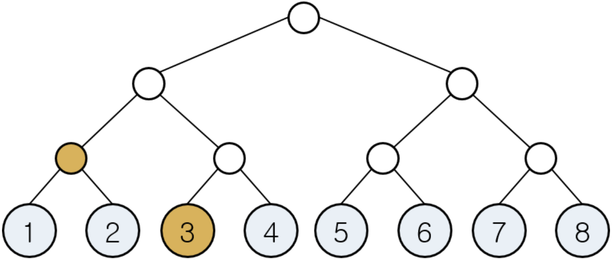
Summing: Example 3

- `Sum(6)`



Summing: Example 4

- `Sum(3)`



Computing Prefix Sums

- Say we want to compute `Sum(k)`
- Maintain a pointer `P` which initially points at leaf `k`
- Climb the tree using the following procedure:
  - If `P` is pointing to a left child of some node:
    - Add the value of `P`
    - Set `P` to the parent node of `P`'s left neighbor
    - If `P` has no left neighbor, terminate
  - Otherwise:
    - Set `P` to the parent node of `P`
- Use an array to implement (review the heap section)

Updating a Value

- Say we want to do `Set(k, x)` (set the value of leaf `k` as `x`)
- This part is a lot easier
- Only the values of leaf `k` and its ancestors change

1. Start at leaf `k`, change its value to `x`
2. Go to its parent, and recompute its value
3. Repeat 2 until the root

Extension

- Make the `Sum()` function work for any interval
  - ... not just ones that start from item 1

- Can support more operations with the new `Sum()` function
  - `Min(i, j)`: Minimum element among items $\mathtt{i}, \ldots, \mathtt{j}$
  - `Max(i, j)`: Maximum element among items $\mathtt{i}, \ldots, \mathtt{j}$

Outline

Lowest Common Ancestor (LCA)

- Input: a rooted tree and a bunch of node pairs
- Output: lowest (deepest) common ancestors of the given pairs of nodes

- Goal: preprocessing the tree in $O(n \log n)$ time in order to answer each LCA query in $O(\log n)$ time

Preprocessing

- Each node stores its depth, as well as the links to every 2^kth ancestor
  - $O(\log n)$ additional storage per node
  - We will use `Anc[x][k]` to denote the 2^kth ancestor of node `x`

- Computing `Anc[x][k]`:
  - `Anc[x][0] = x`'s parent
  - `Anc[x][k] = Anc[ Anc[x][k-1] ][ k-1 ]`

Answering a Query

- Given two node indices $\mathtt{x}$ and $\mathtt{y}$
  - Without loss of generality, assume $\mathtt{depth}(\mathtt{x}) \leq \mathtt{depth}(\mathtt{y})$
- Maintain two pointers p and q, initially pointing at $\mathtt{x}$ and $\mathtt{y}$
- If $\mathtt{depth}(\mathtt{p}) < \mathtt{depth}(\mathtt{q})$, bring q to the same depth as $\mathtt{p}$
  - using `Anc` that we computed before
- Now we will assume that $\mathtt{depth}(\mathtt{p}) = \mathtt{depth}(\mathtt{q})$

Answering a Query

- If `p` and `q` are the same, return `p`
- Otherwise, initialize `k` as $\lceil \log_2 n \rceil$ and repeat:
  - If `k` is 0, return `p`'s parent node
  - If `Anc[p][k]` is undefined, or if `Anc[p][k]` and `Anc[q][k]` point to the same node:
    - Decrease `k` by 1
  - Otherwise:
    - Set `p = Anc[p][k]` and `q = Anc[q][k]` to bring `p` and `q` up by 2^k levels

Conclusion

- We covered LOTS of stuff today
  - Try many small examples with pencil and paper to completely internalize the material
  - Review and solve relevant problems

- Discussion and collaboration are strongly recommended!

Dynamic Programming

Jaehyun Park

CS 97SI
Stanford University

January 12, 2015

Outline

What is DP?

- Wikipedia definition: “method for solving complex problems by breaking them down into simpler subproblems”

- This definition will make sense once we see some examples
  - Actually, we’ll only see problem solving examples today

Steps for Solving DP Problems

1. Define subproblems
2. Write down the recurrence that relates subproblems
3. Recognize and solve the base cases

- Each step is very important!

Outline

1-dimensional DP Example

- Problem: given n, find the number of different ways to write n as the sum of 1, 3, 4
- Example: for $n = 5$, the answer is 6

$$
\begin{aligned}
5 &= 1+1+1+1+1 \\
&= 1+1+3 \\
&= 1+3+1 \\
&= 3+1+1 \\
&= 1+4 \\
&= 4+1
\end{aligned}
$$

1-dimensional DP Example

- Define subproblems
  - Let D_n be the number of ways to write n as the sum of 1, 3, 4
- Find the recurrence
  - Consider one possible solution $n = x_1 + x_2 + \cdots + x_m$
  - If $x_m = 1$, the rest of the terms must sum to $n - 1$
  - Thus, the number of sums that end with $x_m = 1$ is equal to D_{n-1}
  - Take other cases into account ($x_m = 3$, $x_m = 4$)

1-dimensional DP Example

- Recurrence is then

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

- Solve the base cases
  - $D_0 = 1$
  - $D_n = 0$ for all negative n
  - Alternatively, can set: $D_0 = D_1 = D_2 = 1$, and $D_3 = 2$

- We're basically done!

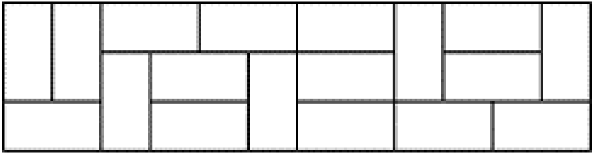
Implementation

```
D[0] = D[1] = D[2] = 1; D[3] = 2;
for(i = 4; i <= n; i++)
    D[i] = D[i-1] + D[i-3] + D[i-4];
```

- Very short!
- Extension: solving this for huge n, say $n \approx 10^{12}$
  - Recall the matrix form of Fibonacci numbers

POJ 2663: Tri Tiling

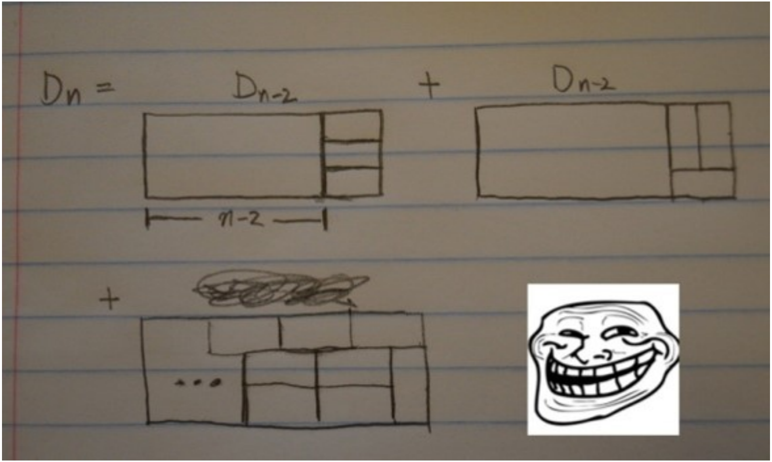
- Given n, find the number of ways to fill a $3 \times n$ board with dominoes
- Here is one possible solution for $n = 12$



POJ 2663: Tri Tiling

- Define subproblems
  - Define D_n as the number of ways to tile a $3 \times n$ board
- Find recurrence
  - Uuuhhhhh...

Troll Tiling

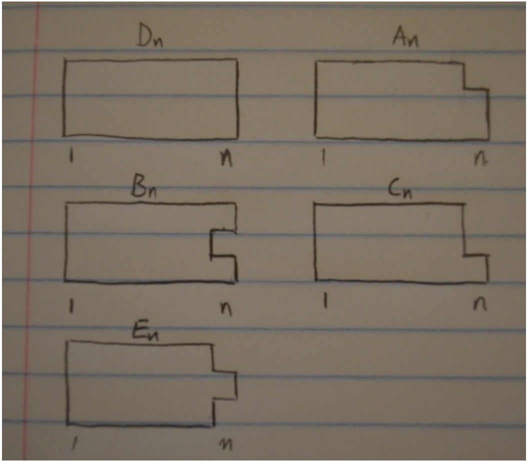


Defining Subproblems

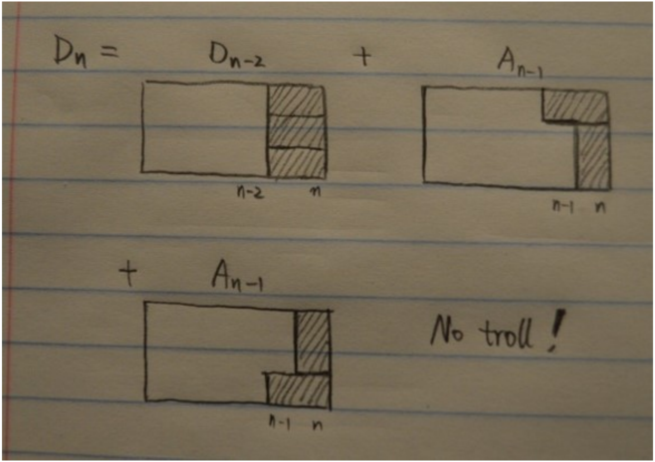
- Obviously, the previous definition didn't work very well
- D_n's don't relate in simple terms

- What if we introduce more subproblems?

Defining Subproblems



Finding Recurrences



Finding Recurrences

- Consider different ways to fill the nth column
  - And see what the remaining shape is
- Exercise:
  - Finding recurrences for A_n, B_n, C_n
  - Just for fun, why is B_n and E_n always zero?

- Extension: solving the problem for $n \times m$ grids, where n is small, say $n \leq 10$
  - How many subproblems should we consider?

Outline

2-dimensional DP Example

- Problem: given two strings x and y, find the longest common subsequence (LCS) and print its length
- Example:
  - x: `ABCBDAB` (A**BC**BD**AB**)
  - y: `BDCABC` (**B**D**CAB**C)
  - "BCAB" is the longest subsequence found in both sequences, so the answer is 4

Solving the LCS Problem

- Define subproblems
  - Let D_{ij} be the length of the LCS of $x_{1\dots i}$ and $y_{1\dots j}$
- Find the recurrence
  - If $x_i = y_j$, they both contribute to the LCS
    - $D_{ij} = D_{i-1,j-1} + 1$
  - Otherwise, either x_i or y_j does not contribute to the LCS, so one can be dropped
    - $D_{ij} = \max\{D_{i-1,j}, D_{i,j-1}\}$
  - Find and solve the base cases: $D_{i0} = D_{0j} = 0$

Implementation

```
for(i = 0; i <= n; i++) D[i][0] = 0;
for(j = 0; j <= m; j++) D[0][j] = 0;
for(i = 1; i <= n; i++) {
    for(j = 1; j <= m; j++) {
        if(x[i] == y[j])
            D[i][j] = D[i-1][j-1] + 1;
        else
            D[i][j] = max(D[i-1][j], D[i][j-1]);
    }
}
```

Outline

Interval DP Example

- Problem: given a string $x = x_{1...n}$, find the minimum number of characters that need to be inserted to make it a palindrome

- Example:
  - x: `Ab3bd`
  - Can get "`dAb3bAd`" or "`Adb3bdA`" by inserting 2 characters (one '`d`', one '`A`')

Interval DP Example

- Define subproblems
  - Let D_{ij} be the minimum number of characters that need to be inserted to make $x_{i\ldots j}$ into a palindrome
- Find the recurrence
  - Consider a shortest palindrome $y_{1\ldots k}$ containing $x_{i\ldots j}$
  - Either $y_1 = x_i$ or $y_k = x_j$ (why?)
  - $y_{2\ldots k-1}$ is then an optimal solution for $x_{i+1\ldots j}$ or $x_{i\ldots j-1}$ or $x_{i+1\ldots j-1}$
    - Last case possible only if $y_1 = y_k = x_i = x_j$

Interval DP Example

- Find the recurrence

$$D_{ij} = \begin{cases} 1 + \min\{D_{i+1,j}, D_{i,j-1}\} & x_i \neq x_j \\ D_{i+1,j-1} & x_i = x_j \end{cases}$$

- Find and solve the base cases: $D_{ii} = D_{i,i-1} = 0$ for all i

- The entries of D must be filled in increasing order of $j - i$

Interval DP Example

```
// fill in base cases here
for(t = 2; t <= n; t++)
    for(i = 1, j = t; j <= n; i++, j++)
        // fill in D[i][j] here
```

- Note how we use an additional variable `t` to fill the table in correct order
- And yes, for loops can work with multiple variables

An Alternate Solution

- Reverse x to get x^R
- The answer is $n - L$, where L is the length of the LCS of x and x^R

- Exercise: Think about why this works

Outline

Dynamic Programming

1-dimensional DP

2-dimensional DP

Interval DP

Tree DP

Subset DP

Tree DP Example

- Problem: given a tree, color nodes black as many as possible without coloring two adjacent nodes
- Subproblems:
  - First, we arbitrarily decide the root node r
  - B_v: the optimal solution for a subtree having v as the root, where we color v black
  - W_v: the optimal solution for a subtree having v as the root, where we don't color v
  - Answer is $\max\{B_r, W_r\}$

Tree DP Example

- Find the recurrence
  - Crucial observation: once v's color is determined, subtrees can be solved independently
  - If v is colored, its children must not be colored

$$B_v = 1 + \sum_{u \in \text{children}(v)} W_u$$

  - If v is not colored, its children can have any color

$$W_v = 1 + \sum_{u \in \text{children}(v)} \max\{B_u, W_u\}$$

- Base cases: leaf nodes

Outline

Subset DP Example

- Problem: given a weighted graph with n nodes, find the shortest path that visits every node exactly once (Traveling Salesman Problem)

- Wait, isn't this an NP-hard problem?
  - Yes, but we can solve it in $O(n^2 2^n)$ time
  - Note: brute force algorithm takes $O(n!)$ time

Subset DP Example

- Define subproblems
  - $D_{S,v}$: the length of the optimal path that visits every node in the set S exactly once and ends at v
  - There are approximately $n2^n$ subproblems
  - Answer is $\min_{v \in V} D_{V,v}$, where V is the given set of nodes

- Let's solve the base cases first
  - For each node v, $D_{\{v\},v} = 0$

Subset DP Example

- Find the recurrence
  - Consider a path that visits all nodes in S exactly once and ends at v
  - Right before arriving v, the path comes from some u in $S - \{v\}$
  - And that subpath has to be the optimal one that covers $S - \{v\}$, ending at u
  - We just try all possible candidates for u

$$D_{S,v} = \min_{u \in S - \{v\}} \left(D_{S - \{v\}, u} + \mathrm{cost}(u, v) \right)$$

Working with Subsets

- When working with subsets, it's good to have a nice representation of sets
- Idea: Use an integer to represent a set
  - Concise representation of subsets of small integers $\{0, 1, \ldots\}$
  - If the ith (least significant) digit is 1, i is in the set
  - If the ith digit is 0, i is not in the set
  - *e.g.*, $19 = \texttt{010011}_{(2)}$ in binary represent a set $\{0, 1, 4\}$

Using Bitmasks

- Union of two sets `x` and `y`: `x | y`
- Intersection: `x & y`
- Symmetric difference: `x ^ y`
- Singleton set $\{i\}$: `1 << i`
- Membership test: `x & (1 << i) != 0`

Conclusion

- Wikipedia definition: "a method for solving complex problems by breaking them down into simpler subproblems"
  - Does this make sense now?

- Remember the three steps!
  1. Defining subproblems
  2. Finding recurrences
  3. Solving the base cases

Combinatorial Games

Jaehyun Park

CS 97SI
Stanford University

January 13, 2015

Combinatorial Games

- Turn-based competitive multi-player games
- Can be a simple win-or-lose game, or can involve points
- Everyone has perfect information
- Each turn, the player changes the current "state" using a valid "move"
- At some states, there are no valid moves
  - The current player immediately loses at these states

Outline

Simple Games

Minimax Algorithm

Nim Game

Grundy Numbers (Nimbers)

Combinatorial Game Example

- Settings: There are n stones in a pile. Two players take turns and remove 1 or 3 stones at a time. The one who takes the last stone wins. Find out the winner if both players play perfectly
- State space: Each state can be represented by the number of remaining stones in the pile
- Valid moves from state x: $x \to (x-1)$ or $x \to (x-3)$, as long as the resulting number is nonnegative
- State 0 is the losing state

Example (continued)

- No cycles in the state transitions
  - Can solve the problem bottom-up (DP)
- A player wins if there is a way to force the opponent to lose
  - Conversely, we lose if there is no such a way
- State x is a winning state (W) if
  - $(x-1)$ is a losing state,
  - OR $(x-3)$ is a losing state
- Otherwise, state x is a losing state (L)

Example (continued)

- DP table for small values of n:

n	0	1	2	3	4	5	6	7
W/L	L	W	L	W	L	W	L	W

- See a pattern?

- Let's prove our conjecture

Example (continued)

- Conjecture: If n is odd, the first player wins. If n is even, the second player wins.

- Holds true for the base case $n = 0$
- In general,
  - If n is odd, we can remove one stone and give the opponent an even number of stones
  - If n is even, no matter what we choose, we have to give an odd number of stones to the opponent

Outline

More Complex Games

- Settings: a competitive zero-sum two-player game
- Zero-sum: if the first player's score is x, then the other player gets $-x$
- Each player tries to maximize his/her own score
- Both players play perfectly

- Can be solved using a *minimax* algorithm

Minimax Algorithm

- Recursive algorithm that decides the best move for the current player at a given state
- Define $f(S)$ as the optimal score of the current player who starts at state S
- Let $T_1, T_2, \ldots, T_m$ be states can be reached from S using a single move
- Let T be the state that minimizes $f(T_i)$
- Then, $f(S) = -f(T)$
  - Intuition: minimizing the opponent's score maximizes my score

Memoization

- (Not *memorization* but *memoization*)
- A technique used to avoid repeated calculations in recursive functions
- High-level idea: take a note (memo) of the return value of a function call. When the function is called with the same argument again, return the stored result
- Each subproblem is solved at most once
  - Some may not be solved at all!

Recursive Function without Memoization

```
int fib(int n)
{
    if(n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

- How many times is `fib(1)` called?

Memoization using `std::map`

```
map<int, int> memo;
int fib(int n)
{
    if(memo.count(n)) return memo[n];
    if(n <= 1) return n;
    return memo[n] = fib(n - 1) + fib(n - 2);
}
```

- How many times is `fib(1)` called?

Minimax Algorithm Pseudocode

- Given state S, want to compute $f(S)$

- If we know $f(S)$ already, return it
- Set return value $x \leftarrow -\infty$
- For each valid next state T:
  - Update return value $x \leftarrow \max\{x, -f(T)\}$
- Write a memo $f(S) = x$ and return x

Possible Extensions

- The game is not zero-sum
  - Each player wants to maximize his own score
  - Each player wants to maximize the difference between his score and the opponent's
- There are more than two players

- All of above can be solved using a similar idea

Outline

Nim Game

- Settings: There are n piles of stones. Two players take turns. Each player chooses a pile, and removes any number of stones from the pile. The one who takes the last stone wins. Find out the winner if both players play perfectly

- Can't really use DP if there are many piles, because the state space is huge

Nim Game Example

- Starts with heaps of 3, 4, 5 stones
  - We will call them heap A, heap B, and heap C

- Alice takes 2 stones from A: $(1, 4, 5)$
- Bob takes 4 from C: $(1, 4, 1)$
- Alice takes 4 from B: $(1, 0, 1)$
- Bob takes 1 from A: $(0, 0, 1)$
- Alice takes 1 from C and wins: $(0, 0, 0)$

Solution to Nim

- Given heaps of size $n_1, n_2, \ldots, n_m$
- The first player wins if and only if the *nim-sum* $n_1 \oplus n_2 \oplus \cdots \oplus n_m$ is nonzero ($\oplus$ is bitwise XOR operator)

- Why?
  - If the nim-sum is zero, then whatever the current player does, the nim-sum of the next state is nonzero
  - If the nim-sum is nonzero, it is possible to force it to become zero (not obvious, but true)

Outline

Playing Multiple Games at Once

- Suppose that multiple games are played at the same time. At each turn, the player chooses a game and make a move. You lose if there is no possible move. We want to determine the winner

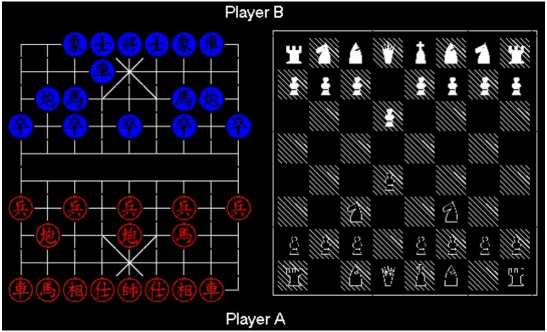


Figure from `http://sps.nus.edu.sg/~limchuwe/cgt/`

Grundy Numbers (Nimbers)

- For each game, we compute its *Grundy number*
- The first player wins if and only if the XOR of all the Grundy numbers is nonzero
  - For example, the Grundy number of a one-pile version of the nim game is equal to the number of stones in the pile (we will see this again later)
- Let's see how to compute the Grundy numbers for general games

Grundy Numbers

- Let S be a state, and $T_1, T_2, \ldots, T_m$ be states can be reached from S using a single move

- The Grundy number $g(S)$ of S is the smallest nonnegative integer that doesn't appear in $\{g(T_1), g(T_2), \ldots, g(T_m)\}$
  - Note: the Grundy number of a losing state is 0
  - Note: I made up the notation $g(\cdot)$. Don't use it in other places

Grundy Numbers Example

- Consider a one-pile nim game
- $g(0) = 0$, because it is a losing state
- State 0 is the only state reachable from state 1, so $g(1)$ is the smallest nonnegative integer not appearing in $\{g(0)\} = \{0\}$. Thus, $g(1) = 1$
- Similarly, $g(2) = 2$, $g(3) = 3$, and so on
- Grundy numbers for this game is then $g(n) = n$
  - That's how we got the nim-sum solution

Another Example

- Let's consider a variant of the game we considered before; only 1 or 2 stones can be removed at each turn
- Now we're going to play many copies of this game at the same time
- Grundy number table:

n	0	1	2	3	4	5	6	7
$g(n)$	0	1	2	0	1	2	0	1

Another Example (continued)

- Grundy number table:

n	0	1	2	3	4	5	6	7
$g(n)$	0	1	2	0	1	2	0	1

- Who wins if there are three piles of stones $(2, 4, 5)$?
- What if we start with $(5, 11, 13, 16)$?
- What if we start with $(10^{100}, 10^{200})$?

Tips for Solving Game Problems

- If the state space is small, use memoization
- If not, print out the result of the game for small test data and look for a pattern
  - This actually works really well!
- Try to convert the game into some nim-variant
- If multiple games are played at once, use Grundy numbers

CS 97SI: INTRODUCTION TO PROGRAMMING CONTESTS

Jaehyun Park

Today's Lecture: Graph Algorithms

- What are graphs?
- Adjacency Matrix and Adjacency List
- Special Graphs
- Depth-First Search and Breadth-First Search
- Topological Sort
- Eulerian Circuit
- Minimum Spanning Tree (MST)
- Strongly Connected Components (SCC)

What are graphs?

- An abstract way of representing connectivity using nodes (or vertices) and edges
- We will label the nodes from 1 to n
- m edges connect some pairs of nodes
  - Edges can be either one-directional (directed) or bidirectional
- Nodes and edges can have some auxiliary information

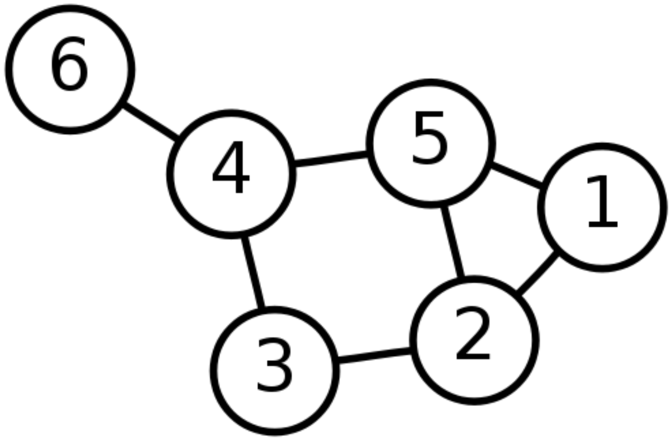


Figure from Wikipedia

Why study graphs?

- Lots of problems formulated and solved in terms of graphs
  - Shortest path problems
  - Network flow problems
  - Matching problems
  - 2-SAT problem
  - Graph coloring problem
  - Traveling Salesman Problem (TSP): *still unsolved!*
  - and many more…

Storing Graphs

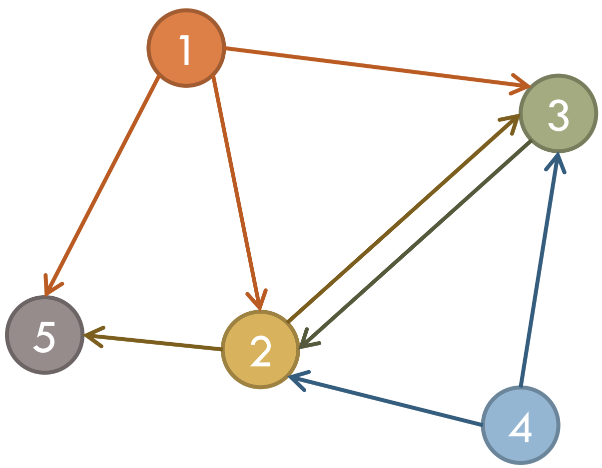
- We need to store both the set of nodes V and the set of edges E
  - Nodes can be stored in an array
  - Edges must be stored in some other way
- We want to support the following operations
  - Retrieving all edges incident to a particular node
  - Testing if given two nodes are directly connected
- Use either adjacency matrix or adjacency list to store the edges

Adjacency Matrix

- An easy way to store connectivity information
  - Checking if two nodes are directly connected: $O(1)$ time
- Make an $n \times n$ matrix A
  - $a_{ij} = 1$ if there is an edge from i to j
  - $a_{ij} = 0$ otherwise
- Uses $\Theta(n^2)$ memory
  - Only use when n is less than a few thousands,
  - AND when the graph is dense

Adjacency List

- Each node has its own list of edges
  - The lists have variable lengths
  - Space usage: $\Theta(n + m)$

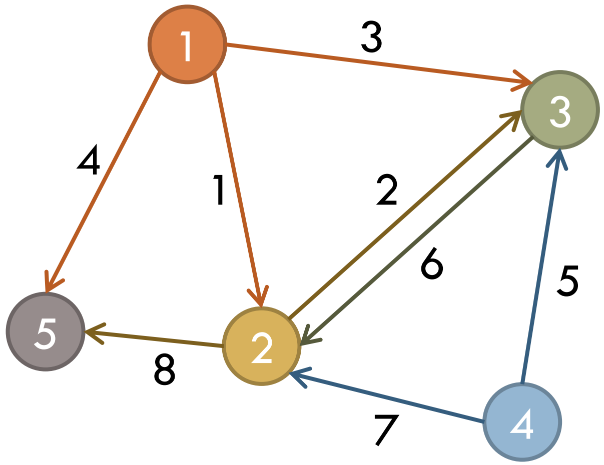


From	To		
1	2	3	5
2	3	5	
3	2		
4	2	5	
5			

Implementing Adjacency List

- Solution 1. Using linked lists
  - Too much memory/time overhead
  - Using dynamic allocated memory or pointers is bad
- Solution 2. Using an array of `vector`s
  - Easier to code, no bad memory issues
  - But very slow
- Solution 3. Using arrays (!)
  - Assuming the total number of edges is known
  - Very fast and memory-efficient

Implementation Using Arrays



ID	To	Next Edge ID
1	2	-
2	3	-
3	3	1
4	5	3
5	3	-
6	2	-
7	2	5
8	5	2

From	1	2	3	4	5
Last Edge ID	4	8	6	7	-

Implementation Using Arrays

- Have two arrays `E` of size m and `LE` of size n
  - `E` contains the edges
  - `LE` contains the starting pointers of the edge lists
- Initialize `LE[i] = -1` for all `i`
  - `LE[i] = 0` is also fine if the arrays are 1-indexed
- Inserting a new edge from `u` to `v` with ID `k`
  - `E[k].to = v`
  - `E[k].nextID = LE[u]`
  - `LE[u] = k`

Implementation Using Arrays

- Iterating over all edges starting at u:
  - ```
    for(ID = LE[u]; ID != -1; ID = E[ID].nextID)
      // E[ID] is an edge starting from u
    ```

- It's pretty hard to modify the edge lists
  - The graph better be static!
    ```

Special Graphs

- Tree: a connected acyclic graph
  - The most important type of graph in CS
  - Alternate definitions (all are equivalent!)
    - A connected graph with $n-1$ edges
    - An acyclic graph with $n-1$ edges
    - There is exactly one path between every pair of nodes
    - An acyclic graph but adding any edge results in a cycle
    - A connected graph but removing any edge disconnects it

Special Graphs

- Directed Acyclic Graph (DAG): the name says what it is
  - Equivalent to a partial ordering of nodes

- Bipartite Graph
  - Nodes can be separated into two groups S and T such that edges exist between S and T only (no edges within S or within T)

Graph Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms

- We will cover two algorithms
  - Depth-First Search (DFS): uses recursion (stack)
  - Breadth-First Search (BFS): uses queue

Depth-First Search Pseudocode

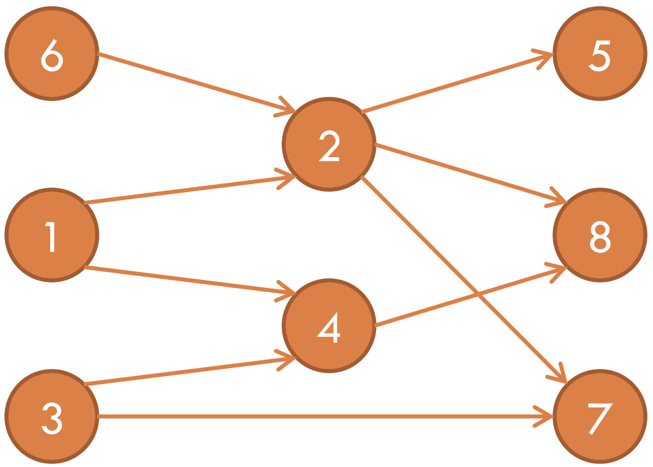
- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited
  - For each edge $v \to u$:
    - If u is not visited, call DFS(u)
- Use non-recursive version if recursion depth is too big (over a few thousands)
  - Replace recursive calls with a stack

Breadth-First Search Pseudocode

- BFS(v): visits all the nodes reachable from v in breadth-first order
  - Initialize a queue Q
  - Mark v as visited and push it to Q
  - While Q is not empty:
    - Take the front element of Q and call it w
    - For each edge $w \rightarrow u$:
      - If u is not visited, mark it as visited and push it to Q

Topological Sort

- Input: a DAG $G = (V, E)$
- Output: an ordering of nodes such that for each edge $u \rightarrow v$, u comes before v
- There can be many answers
  - e.g. {6, 1, 3, 2, 7, 4, 5, 8} and {1, 6, 2, 3, 4, 5, 7, 8} are valid orderings for the graph on the right



Topological Sort

- Any node without an incoming edge can be the first element
- After deciding the first node, remove outgoing edges from it
- Repeat!

- Time complexity: $O(n^2 + m)$
  - Ugh, too slow…

Topological Sort (faster version)

- Precompute the number of incoming edges $\deg(v)$ for each node v
- Put all nodes with zero $\deg(\cdot)$ into a queue Q
- Repeat until Q becomes empty:
  - Take v from Q
  - For each edge $v \to u$
    - Decrement $\deg(u)$ (essentially removing the edge $v \to u$)
    - If $\deg(u)$ becomes zero, push u to Q
- Time complexity: $\Theta(n + m)$

Eulerian Circuit

- Given an undirected graph G
- We want to find a sequence of nodes that visits every edge exactly once and comes back to the starting point
- Eulerian circuits exist if and only if
  - G is connected
  - and each node has an even degree

Constructive Proof of Existence

- Pick any node in G and walk randomly(!) without using the same edge more than once
- Each node is of even degree, so when you enter a node, there will be an unused edge you exit through
  - Except at the starting point, at which you can get stuck
- When you get stuck, what you have is a cycle
  - Remove the cycle and repeat the process in each connected component
  - Glue the cycles together to finish!

Related Problems

- Eulerian path: exists if and only if the graph is connected and the number of nodes with odd degree is 0 or 2.

- Hamiltonian path/cycle: a path/cycle that visits every *node* in the graph exactly once. Looks similar but still unsolved!

Minimum Spanning Tree (MST)

- Given an undirected weighted graph $G = (V, E)$
- Want to find a subset of E with the minimum total weight that connects all the nodes into a tree

- We will cover two algorithms:
  - Kruskal's algorithm
  - Prim's algorithm

Kruskal's Algorithm

- Main idea: the edge $e^\star$ with the smallest weight has to be in the MST
  - Simple proof:
    - Assume not. Take the MST T that doesn't contain $e^\star$.
    - Add $e^\star$ to T, which results in a cycle.
    - Remove the edge with the highest weight from the cycle.
      - The removed edge cannot be $e^\star$ since it has the smallest weight.
    - Now we have a better spanning tree than T
    - Contradiction!

Kruskal's Algorithm

- Another main idea: after an edge is chosen, the two nodes at the ends can be merged and considered as a single node (supernode)
- Pseudocode:
  - Sort the edges in increasing order of weight
  - Repeat until there is one supernode left:
    - Take the minimum weight edge $e^\star$
    - If $e^\star$ connects two different supernodes:
      - Connect them and merge the supernodes (use union-find)
    - Otherwise, ignore $e^\star$ and go back

Prim's Algorithm

- Main idea:
  - Maintain a set S that starts out with a single node s
  - Find the smallest weighted edge $e^\star = (u, v)$ that connects $u \in S$ and $v \notin S$
  - Add $e^\star$ to the MST, add v to S
  - Repeat until $S = V$
- Differs from Kruskal's in that we grow a single supernode S instead of growing multiple ones here and there

Prim's Algorithm Pseudocode

- Initialize S to $\{s\}$, D_v to $\text{cost}(s, v)$ for every v
  - If there is no edge between s and v, $\text{cost}(s, v) = \infty$
- Repeat until $S = V$:
  - Find $v \notin S$ with smallest D_v
    - Use a priority queue or a simple linear search
  - Add v to S, add D_v to the total weight of the MST
  - For each edge (v, w):
    - Update D_w to $\min(D_w, \text{cost}(v, w))$
- Can be modified to compute the actual MST along with the total weight

Kruskal's vs Prim's

- Kruskal's Algorithm
  - Takes $O(m \log m)$ time
  - Pretty easy to code
  - Generally slower than Prim's
- Prim's Algorithm
  - Time complexity depends on the implementation:
    - Can be $O(n^2 + m)$, $O(m \log n)$, $O(n \log n)$
  - A bit trickier to code
  - Generally faster than Kruskal's

Strongly Connected Components (SCC)

- Given a *directed* graph $G = (V, E)$
- A graph is *strongly connected* if all nodes are reachable from every single node in V
- Strongly connected components of G are maximal strongly connected subgraphs of G
  - The graph on the right has 3 SCCs: {a, b, e}, {c, d, h}, {f, g}

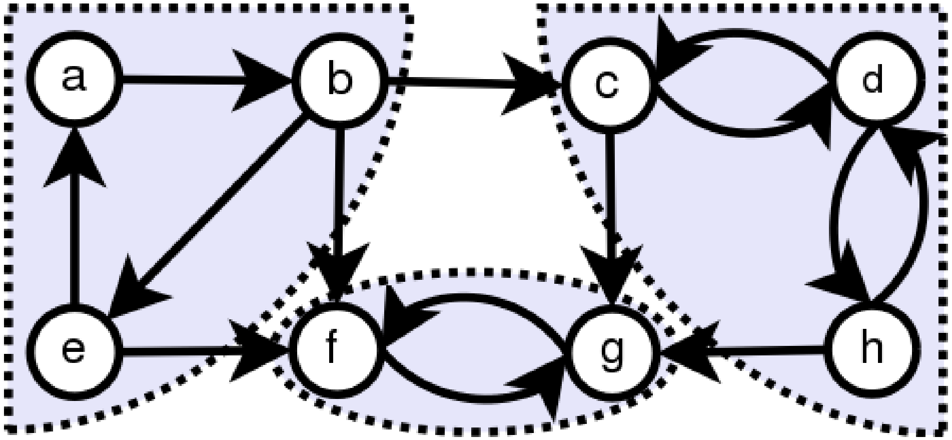


Figure from Wikipedia

Kosaraju's Algorithm

- Initialize counter $c = 0$
- While not all nodes are labeled:
  - Choose an arbitrary unlabeled node v
  - Start DFS from v
    - Check the current node x as visited
    - Recurse on all unvisited neighbors
    - After the DFS calls are finished, increment c and set x's label to c
- Reverse the direction of all the edges
- For node v with label $n \dots 1$
  - Find all reachable nodes from v and group them as an SCC

Kosaraju's Algorithm

- We won't prove why this works ☺
- Two graph traversals are performed
  - Running time: $\Theta(n + m)$

- Other SCC algorithms exist but this one is particularly easy to code
  - and asymptotically optimal

CS 97SI: INTRODUCTION TO PROGRAMMING CONTESTS

Jaehyun Park

Today's Lecture

- Shortest Path Problem
- Floyd-Warshall Algorithm
- Dijkstra's Algorithm
- Bellman-Ford Algorithm
  - System of difference constraints

- Maybe: Problem Discussion

Shortest Path Problem

- Input: a weighted graph $G = (V, E)$
  - The edges can be directed or not
  - Sometimes, we allow negative edge weights
- Output: the path between two given nodes u and v that minimizes the total weight (or cost, length)
  - Sometimes, we want to compute all-pair shortest paths
  - Sometimes, we want to compute shortest paths from u to all other nodes

Floyd-Warshall Algorithm

- Given a directed weighted graph G
- Outputs a matrix D where d_{ij} is the shortest distance from node i to j
- Can detect a negative-weight cycle
- Runs in $\Theta(n^3)$ time
- Extremely easy to code
  - Coding time less than a few minutes

Floyd-Warshall Pseudocode

- Initialize D to the given cost matrix
- For $k = 1 \dots n$:
  - For all i and j:
    - $d_{ij} = \min\left(d_{ij}, d_{ik} + d_{kj}\right)$
- If $d_{ij} + d_{ji} < 0$ for some i and j, then the graph has a negative weight cycle
- Done!
  - But how does this work?

How does Floyd-Warshall work?

- Define $f(i,j,k)$ as the shortest distance from i to j, using $1 \dots k$ as intermediate nodes
  - $f(i,j,n)$ is the shortest distance from i to j
  - $f(i,j,0) = \text{cost}(i,j)$
- The optimal path for $f(i,j,k)$ may or may not have k as an intermediate node
  - If it does, $f(i,j,k) = f(i,k,k-1) + f(k,j,k-1)$
  - Otherwise, $f(i,j,k) = f(i,j,k-1)$
- Therefore, $f(i,j,k)$ is the minimum of the two quantities above

How does Floyd-Warshall work?

- We have the following recurrences and base cases
  - $f(i,j,0) = \text{cost}(i,j)$
  - $f(i,j,k) = \min\{f(i,k,k-1) + f(k,j,k-1),$
    $f(i,j,k-1)\}$
- From the values of $f(\cdot,\cdot,k-1)$, we can calculate $f(\cdot,\cdot,k)$
  - It turns out that we don't need a separate matrix for each k; overwriting the existing values is fine
- That's how we get Floyd-Warshall algorithm

Dijkstra's Algorithm

- Given a directed weighted graph G and a source s
  - Important: The edge weights have to be nonnegative!
- Outputs a vector d where d_i is the shortest distance from s to node i
- Time complexity depends on the implementation:
  - Can be $O(n^2 + m)$, $O(m \log n)$, $O(n \log n)$
- Very similar to Prim's algorithm
- Intuition: Find the closest node to s, and then the second closest one, then the third, etc.

Dijkstra's Algorithm

- Maintain a set of nodes S, the shortest distances to which are decided
- Also maintain a vector d, the shortest distance estimate from s
- Initially, $S = \{s\}$, and $d_v = \text{cost}(s, v)$
- Repeat until $S = V$:
  - Find $v \notin S$ with the smallest d_v, and add it to S
  - For each edge $v \to u$ of cost c:
    - $d_u = \min(d_u, d_v + c)$

Bellman-Ford Algorithm

- Given a directed weighted graph G and a source s
- Outputs a vector d where d_i is the shortest distance from s to node i
- Can detect a negative-weight cycle
- Runs in $\Theta(nm)$ time
- Extremely easy to code
  - Coding time less than a few minutes

Bellman-Ford Pseudocode

- Initialize $d_s = 0$ and $d_v = \infty$ for all $v \neq s$
- For $k = 1 \dots n - 1$:
  - For each edge $u \rightarrow v$ of cost c:
    - $d_v = \min(d_v, d_u + c)$
- For each edge $u \rightarrow v$ of cost c:
  - If $d_v > d_u + c$:
    - Then the graph contains a negative-weight cycle

Why does Bellman-Ford work?

- A shortest path can have at most $n-1$ edges
- At the kth iteration, all shortest paths of k or less edges are computed
- After $n-1$ iterations, all distances are final: for every edge $u \to v$ of cost c, $d_v \leq d_u + c$ holds
  - Unless there is a negative-weight cycle
  - This is how the negative-weight cycle detection works

System of Difference Constraints

- Given m inequalities of the form $x_i - x_j \leq c$
- Want to find real numbers $x_1, \dots, x_n$ that satisfy all the given inequalities

- Seemingly this has nothing to do with shortest paths
  - But it can be solved using Bellman-Ford

Graph Construction

- Create node i for every variable x_i
- Make an imaginary source node s
- Create zero-weight edges from s to all other nodes
- Rewrite the given inequalities as $x_i \leq x_j + c$
  - For each of these constraint, make an edge from j to i with weight c

- Now we run Bellman-Ford using s as the source

What happens?

- For every edge $j \rightarrow i$ with cost c, the shortest distance d vector will satisfy $d_i \leq d_j + c$
  - Setting $x_i = d_i$ gives a solution!
- What if there is a negative-weight cycle?
  - Assume that $1 \rightarrow 2 \rightarrow \cdots k \rightarrow 1$ is a negative-weight cycle
  - From our construction, the given constraints contain $x_2 \leq x_1 + c_1$, $x_3 \leq x_2 + c_2$, etc.
  - Adding all of them gives $0 \leq (\text{something negative})$
  - i.e. the given constraints were impossible to satisfy

System of Difference Constraints

- It turns out that our solution minimizes the *span* of the variables: $\max x_i - \min x_i$
  - We won't prove it
  - This is a big hint on POJ 3169!

CS 97SI: INTRODUCTION TO PROGRAMMING CONTESTS

Jaehyun Park

Last Lecture on Graph Algorithms

- Network Flow Problems
  - Maximum Flow
  - Minimum Cut
- Ford-Fulkerson Algorithm
- Application: Bipartite Matching
- Min-cost Max-flow Algorithm

Network Flow Problems

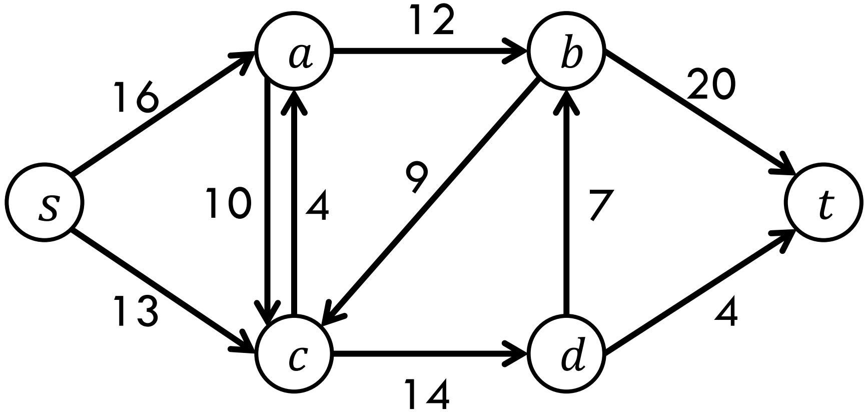
- A type of network optimization problem
- Arise in many different contexts (CS 261):
  - Networks: routing as many packets as possible on a given network
  - Transportation: sending as many trucks as possible, where roads have limits on the number of trucks per unit time
  - Bridges: destroying (?!) some bridges to disconnect s from t, while minimizing the cost of destroying the bridges

Network Flow Problems

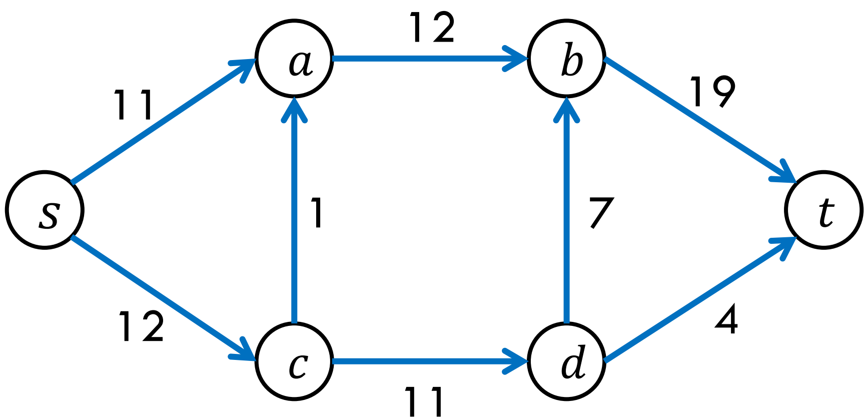
- Settings: Given a directed graph $G = (V, E)$, where each edge e is associated with its capacity $c(e) > 0$. Two special nodes *source* s and *sink* t are given ($s \neq t$)
- Problem: Maximize the total amount of *flow* from s to t subject to two constraints
  - Flow on edge e doesn't exceed $c(e)$
  - For every node $v \neq s, t$, incoming flow is equal to outgoing flow

Network Flow Example (from CLRS)

- Capacities



- Maximum Flow (of 23 units)



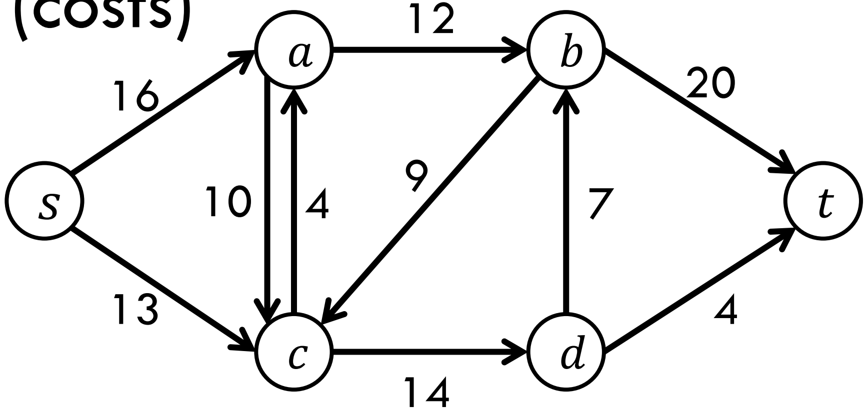
Alternate Formulation: Minimum Cut

- We want to remove some edges from the graph such that after removing the edges, there is no path from s to t
- The cost of removing e is equal to its capacity $c(e)$
- The minimum cut problem is to find a cut with minimum total cost

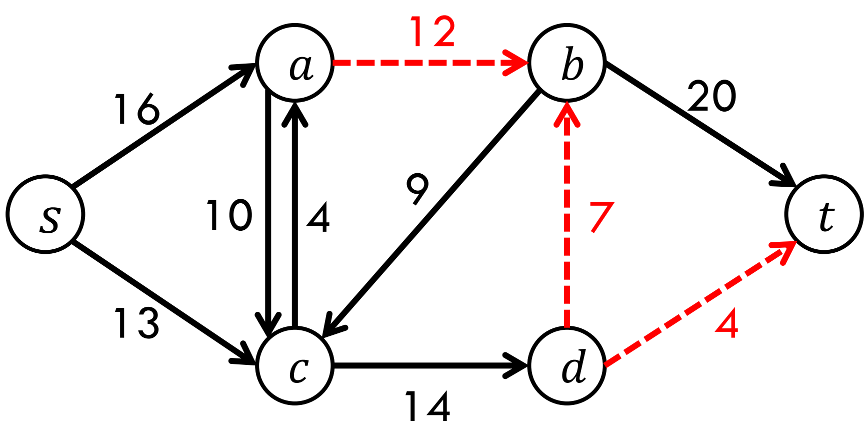
- Theorem: $(\text{maximum flow}) = (\text{minimum cut})$
  - Take CS 261 if you want to see the proof ☺

Minimum Cut Example

- Capacities (costs)

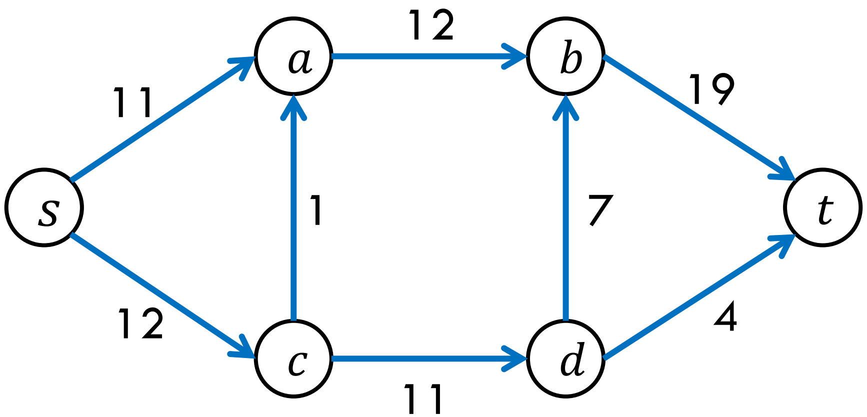


- Minimum Cut (red edges are removed)



Flow Decomposition

- Any valid flow can be decomposed into flow paths and circulations



  - $s \to a \to b \to t$: 11
  - $s \to c \to a \to b \to t$: 1
  - $s \to c \to d \to b \to t$: 7
  - $s \to c \to d \to t$: 4

Ford-Fulkerson Algorithm

- A simple and practical max-flow algorithm
- Main idea: find valid flow paths until there is none left, and add them up
- How do we know if this gives a maximum flow?
  - Proof sketch: Suppose not. Take a maximum flow $f^\star$ and subtract our flow f. It is a valid flow of positive total flow. By the flow decomposition, it can be decomposed into flow paths and circulations. These must have been found by Ford-Fulkerson. Contradiction.

Back Edges

- We don't need to maintain the amount of flow on each edge but work with capacity values directly
- If f amount of flow goes through $u \to v$, then:
  - Decrease $c(u \to v)$ by f
  - Increase $c(v \to u)$ by f
- Why do we need to do this?
  - Sending flow to both directions is equivalent to canceling flow

Ford-Fulkerson Pseudocode

- Set $f_{\text{total}} = 0$
- Repeat until there is no path from s to t:
  - Run DFS from s to find a flow path to t
  - Let f be the minimum capacity value on the path
  - Add f to f_{total}
  - For each edge $u \rightarrow v$ on the path:
    - Decrease $c(u \rightarrow v)$ by f
    - Increase $c(v \rightarrow u)$ by f

Analysis

- Assumption: capacities are integer-valued
- Finding a flow path takes $\Theta(n+m)$ time
- We send at least 1 unit of flow through the path
- If the max-flow is $f^\star$, the time complexity is $O\big((n+m)f^\star\big)$
  - "Bad" in that it depends on the output of the algorithm
  - Nonetheless, easy to code and works well in practice

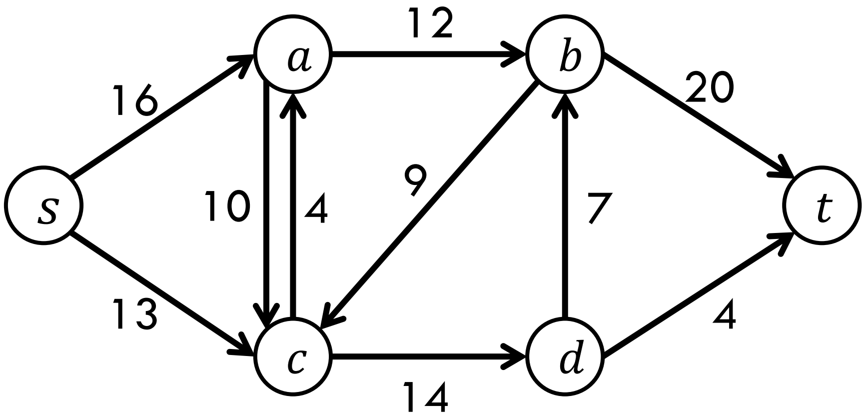
Computing the Min-Cut

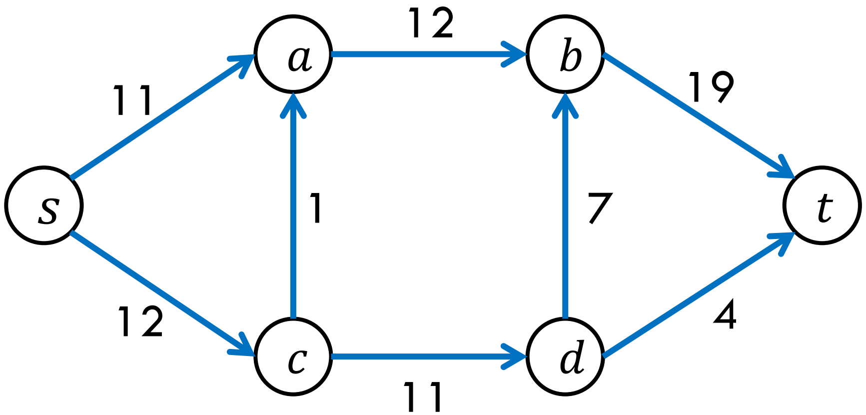
- We know that max-flow is equal to min-cut
- And we now know how to find the max-flow

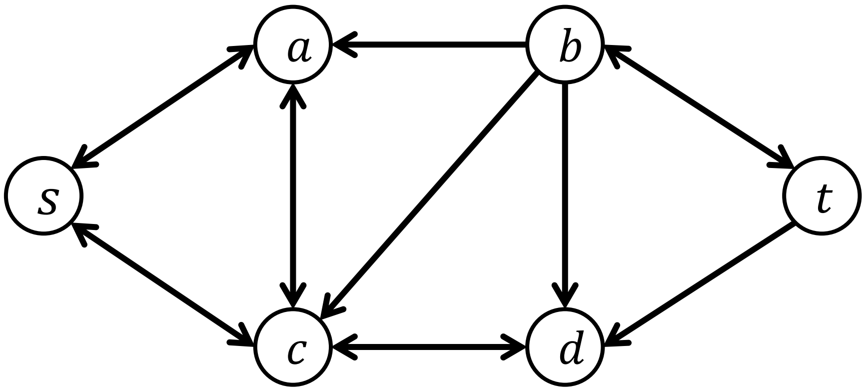
- Question: how do we find the min-cut?
- Answer: use the *residual graph*

Computing the Min-Cut

- "Subtract" the max-flow from the original graph



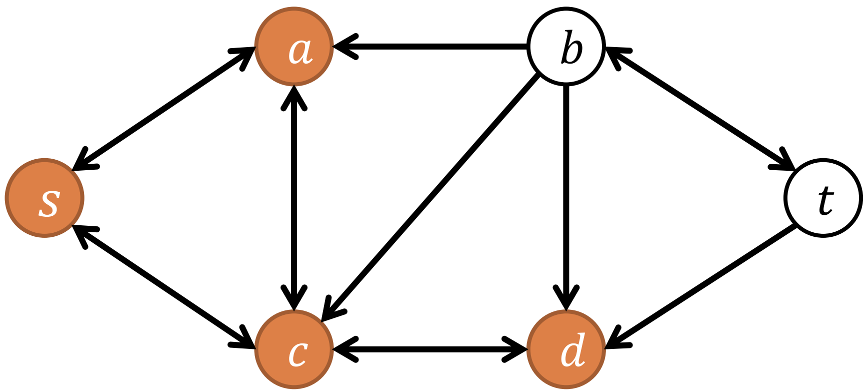




Only the topology of the residual graph is shown.
Don't forget to add the back edges!

Computing the Min-Cut

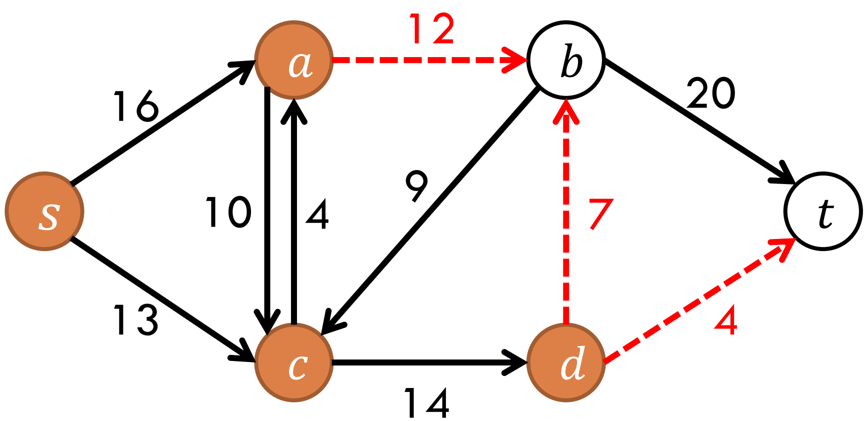
- Mark all nodes reachable from s
  - Call the set of reachable nodes A



- Now separate these nodes from the others
  - Edges go from A to $V - A$ are cut

Computing the Min-Cut

- Look at the original graph and find the cut:



- Why isn't $b \to c$ cut?

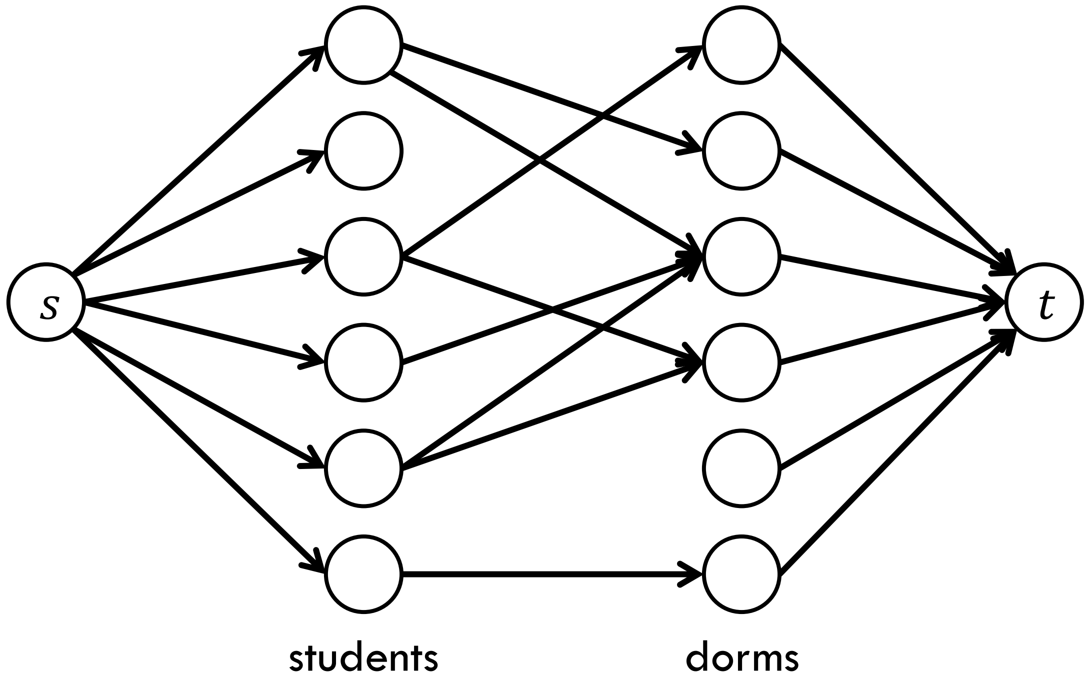
Bipartite Matching

- Settings:
  - n students and d dorms
  - Each student wants to live in one of the dorms of his choice
  - Each dorm can accommodate at most one student (?!)
    - Fine, we will fix this later…

- Problem: find an assignment that maximizes the number of students who get a housing

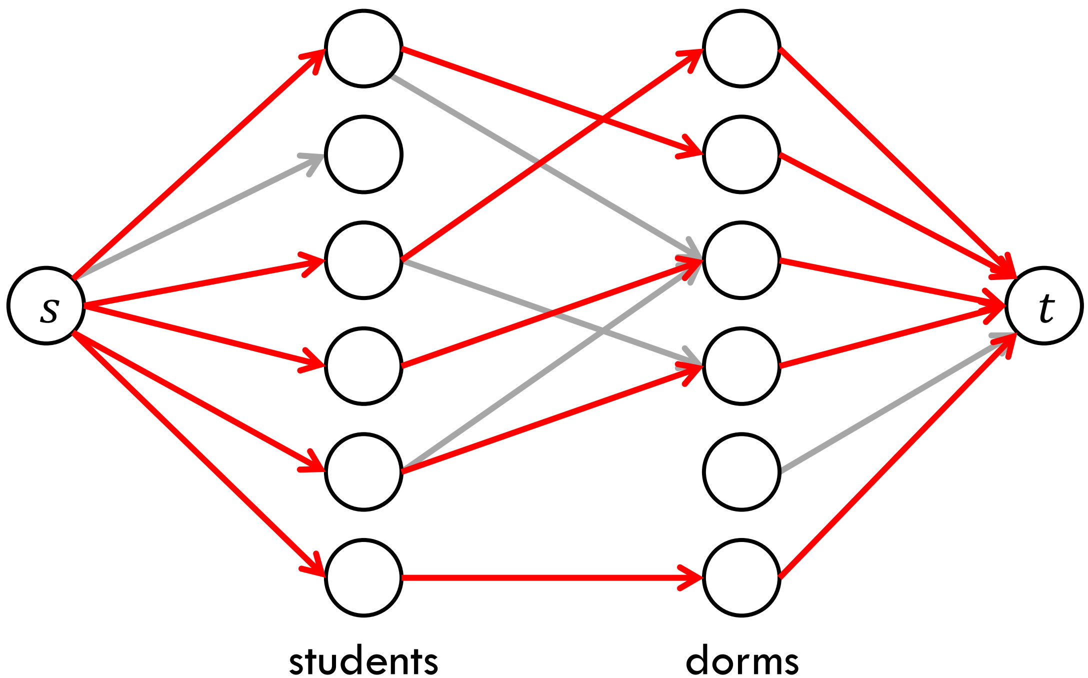
Flow Network Construction

- Add source and sink
- Make edges between students and dorms
  - All the edge weights are 1



Flow Network Construction

- Find the max-flow
- Find the optimal assignment from the chosen edges



Related Problems

- A more reasonable variant of the previous problem: dorm j can accommodate c_j students
  - Make an edge with capacity c_j from dorm j to the sink
- Decomposing a DAG into nonintersecting paths
  - Split each vertex v into v_{left} and v_{right}
  - For each edge $u \to v$ in the DAG, make an edge from u_{left} to v_{right}
- And many others…

Min-Cost Max-Flow

- A variant of the max-flow problem
- Each edge e has capacity $c(e)$ and cost $\text{cost}(e)$
- You have to pay $\text{cost}(e)$ amount of money per unit flow flowing through e
- Problem: find the maximum flow that has the minimum total cost
- A lot harder than the regular max-flow
  - But there is an easy algorithm that works for small graphs

Simple (?) Min-Cost Max-Flow

- Forget about the costs and just find *a* max-flow
- Repeat:
  - Take the residual graph
  - Find a negative-cost cycle using Bellman-Ford
    - If there is none, finish
  - Circulate flow through the cycle to decrease the total cost, until one of the edges is saturated
    - The total amount of flow doesn't change!
- Time complexity: very slow

Notes on Max-Flow Problems

- Remember different formulations of the max-flow problem
  - Again, $(\text{maximum flow}) = (\text{minimum cut})$!
- Often the crucial part is to construct the flow network
- We didn't cover fast max-flow algorithms
  - Refer to the Stanford Team notebook for efficient flow algorithms

CS 97SI: INTRODUCTION TO PROGRAMMING CONTESTS

Jaehyun Park

Computational Geometry

- Cross Product
  - Segment-Segment Intersection
- Convex Hull Problem
  - Graham Scan
- Sweep Line Algorithm
- Intersecting Half-planes
- A Useful Note on Binary/Ternary Search

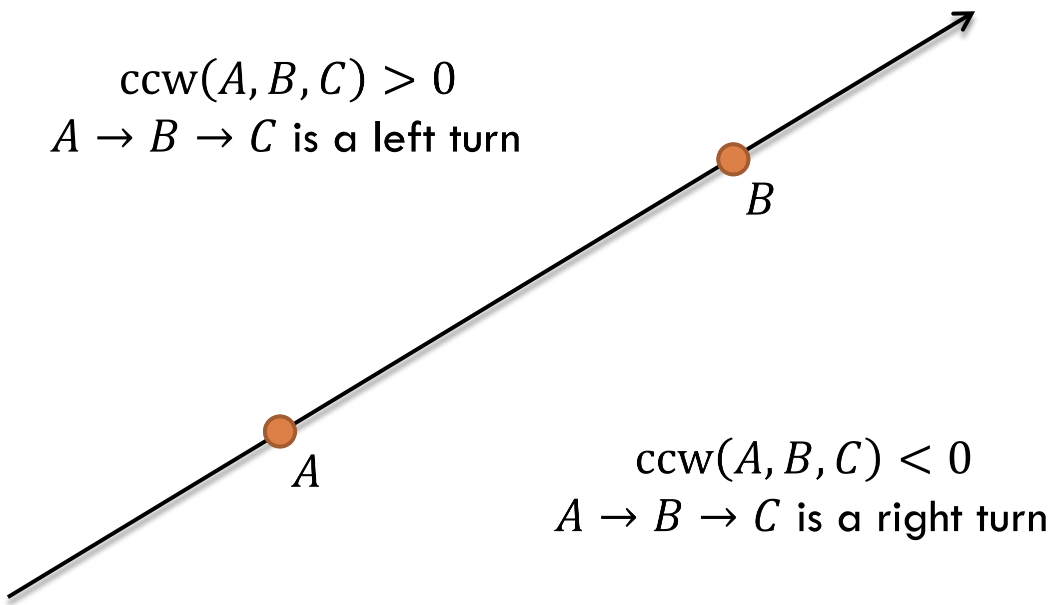
Cross Product

- Arguably the most important operation in 2D geometry
  - We'll use it all the time

- Applications:
  - Determining the (signed) area of a triangle
  - Testing if three points are collinear
  - Determining the orientation of three points
  - Testing if two line segments intersect

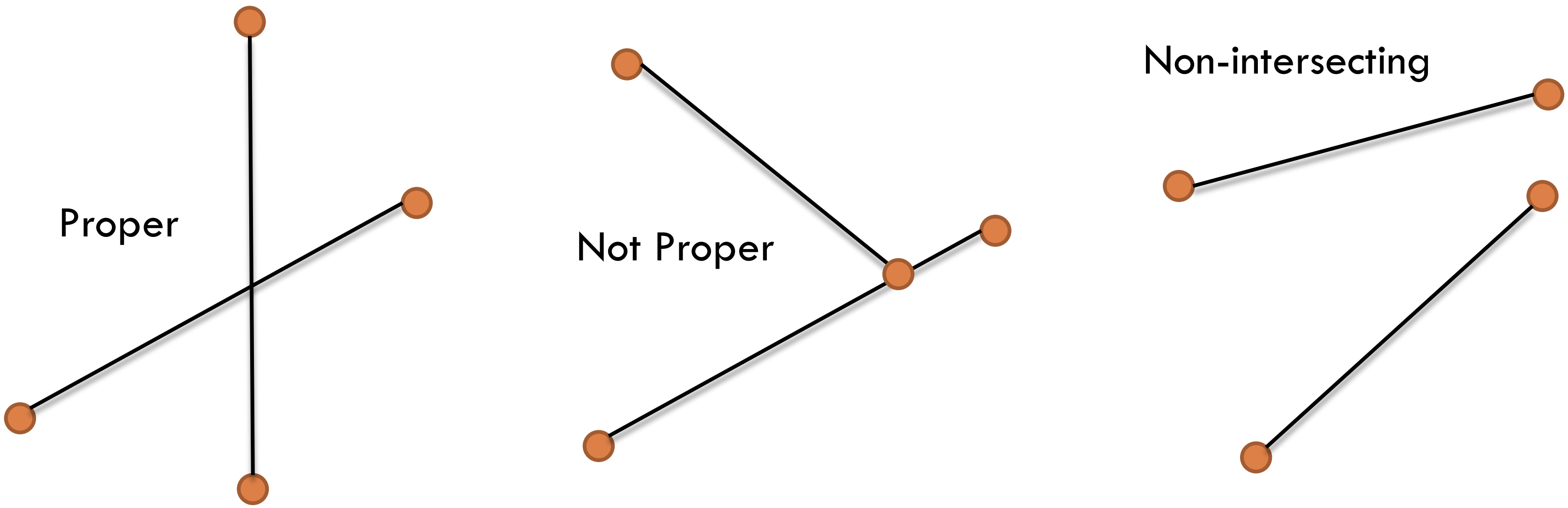
Cross Product

- Define $\mathrm{ccw}(A, B, C) = (B - A) \times (C - A)$



Segment-Segment Intersection Test

- Given two segments AB and CD
- Want to determine if they intersect properly: two segments meet at a single point that are strictly inside both segments



Segment-Segment Intersection Test

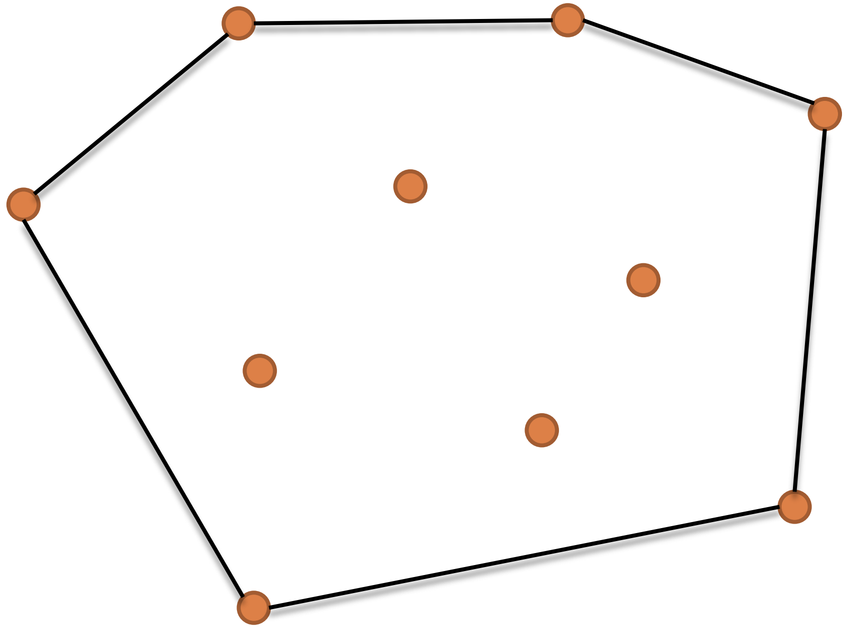
- Assume that the segments intersect
  - From A's point of view, looking straight to B, C and D must lie on different sides
  - Holds true for the other segment as well
- The intersection exists and is proper if:
  - $\mathrm{ccw}(A, B, C) \times \mathrm{ccw}(A, B, D) < 0$
  - AND $\mathrm{ccw}(C, D, A) \times \mathrm{ccw}(C, D, B) < 0$

Segment-Segment Intersection Test

- Determining non-proper intersections
  - We need more special cases to consider!
  - e.g. If $\mathrm{ccw}(A, B, C)$, $\mathrm{ccw}(A, B, D)$, $\mathrm{ccw}(C, D, A)$, $\mathrm{ccw}(C, D, B)$ are all zeros, then two segments are collinear
  - Very careful implementation is required

Convex Hull Problem

- Given n points on the plane, find the smallest convex polygon that contains all the given points
  - For simplicity, assume that no three points are collinear



Simple $O(n^3)$ algorithm

- AB is an edge of the convex hull iff $\mathrm{ccw}(A, B, C)$ have the same sign for all other given points C
  - This gives us a simple algorithm
- For each A and B:
  - If $\mathrm{ccw}(A, B, C) > 0$ for all $C \neq A, B$:
    - Record the edge $A \rightarrow B$
- Walk along the recorded edges to recover the convex hull

Faster Algorithm: Graham Scan

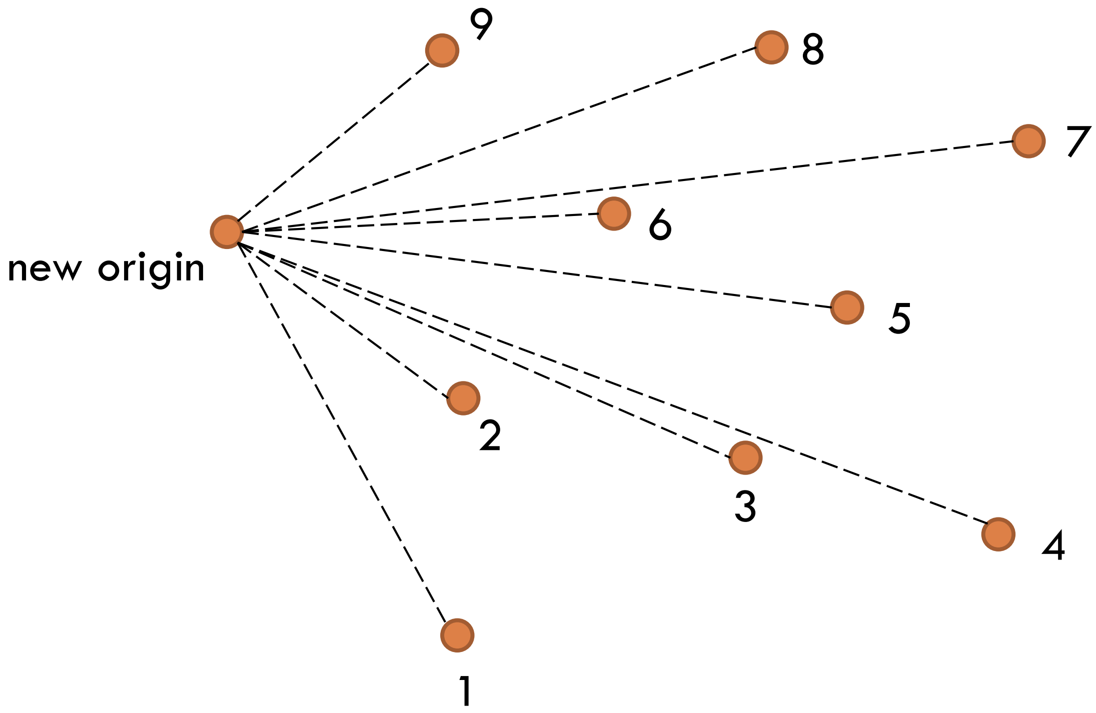
- We know that the leftmost given point has to be in the convex hull
  - We assume that there is a unique leftmost point
- Make the leftmost point the origin
  - So that all other points have positive x coordinates
- Sort the points in increasing order of y/x
  - Increasing order of angle, whatever you like to call it
- Incrementally construct the convex hull using a stack

Incremental Construction

- We maintain a *convex chain* of the given points
- For each i, we do the following:
  - Append point i to the current chain
  - If the new point causes a concave corner, remove the bad vertex from the chain that causes it
  - Repeat until the new chain becomes convex

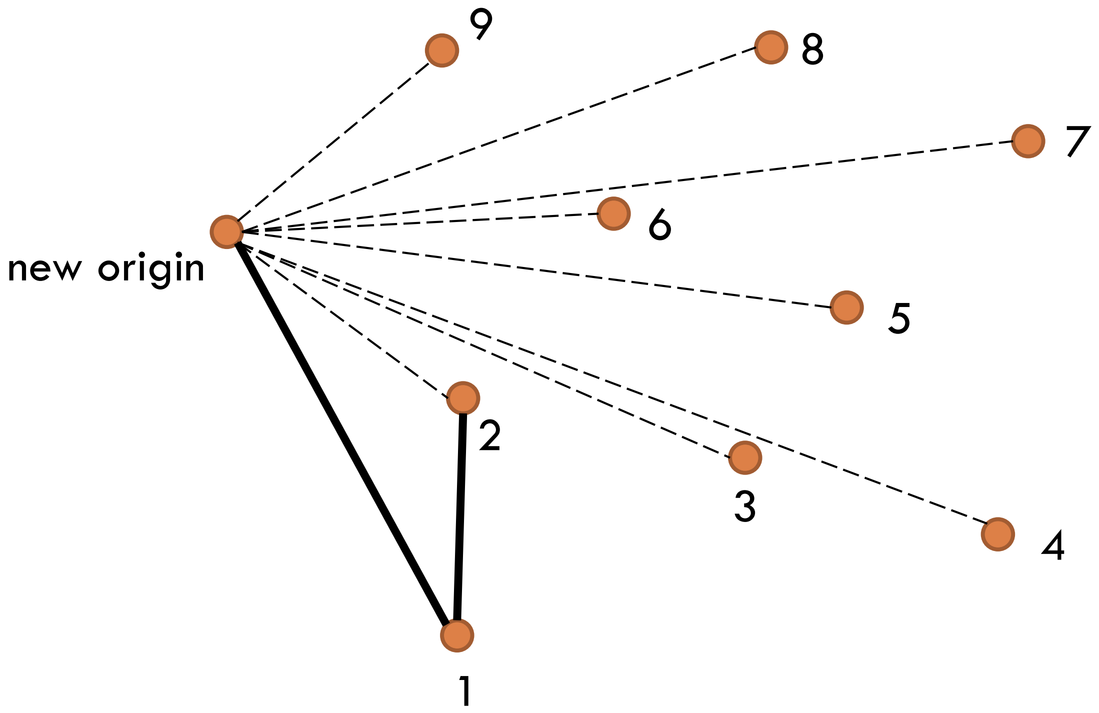
Example

- Points are numbered in increasing order of y/x



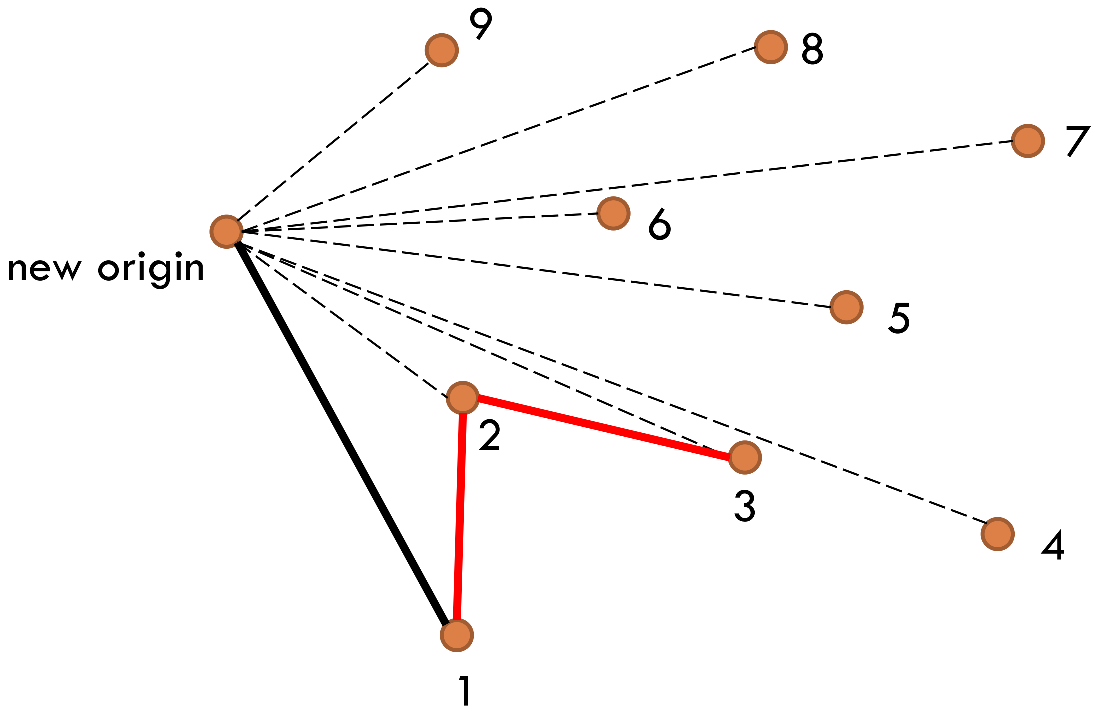
Example

- Add the first two points in the chain



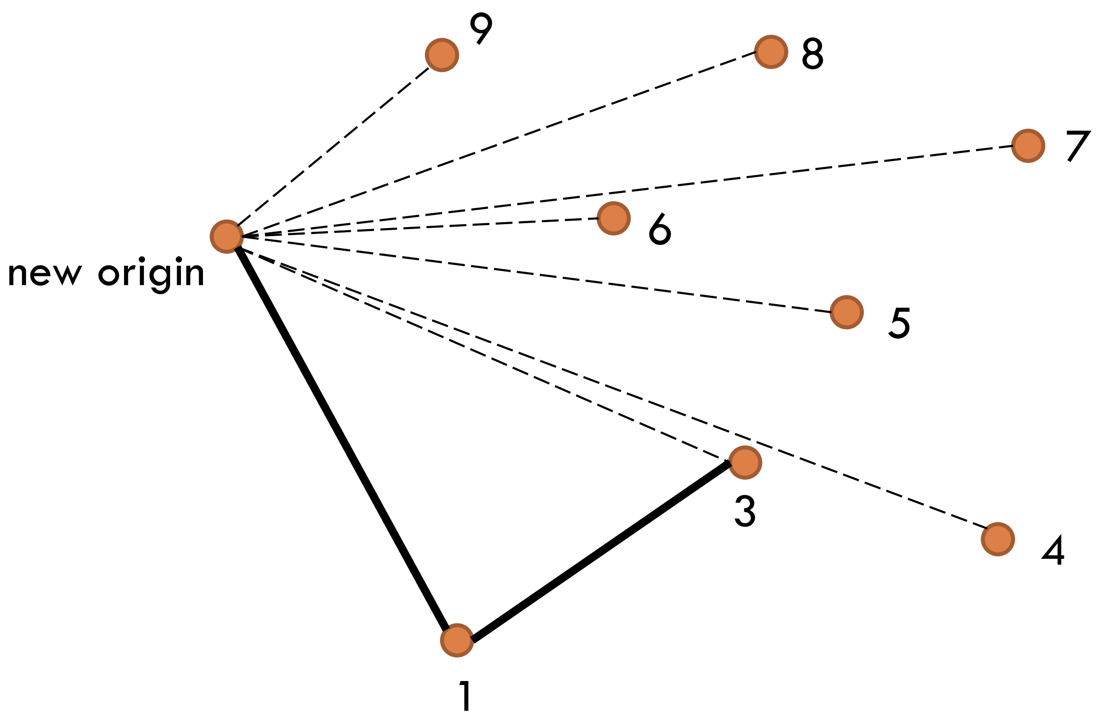
Example

- Adding point 3 causes a concave corner 1-2-3
  - Remove 2



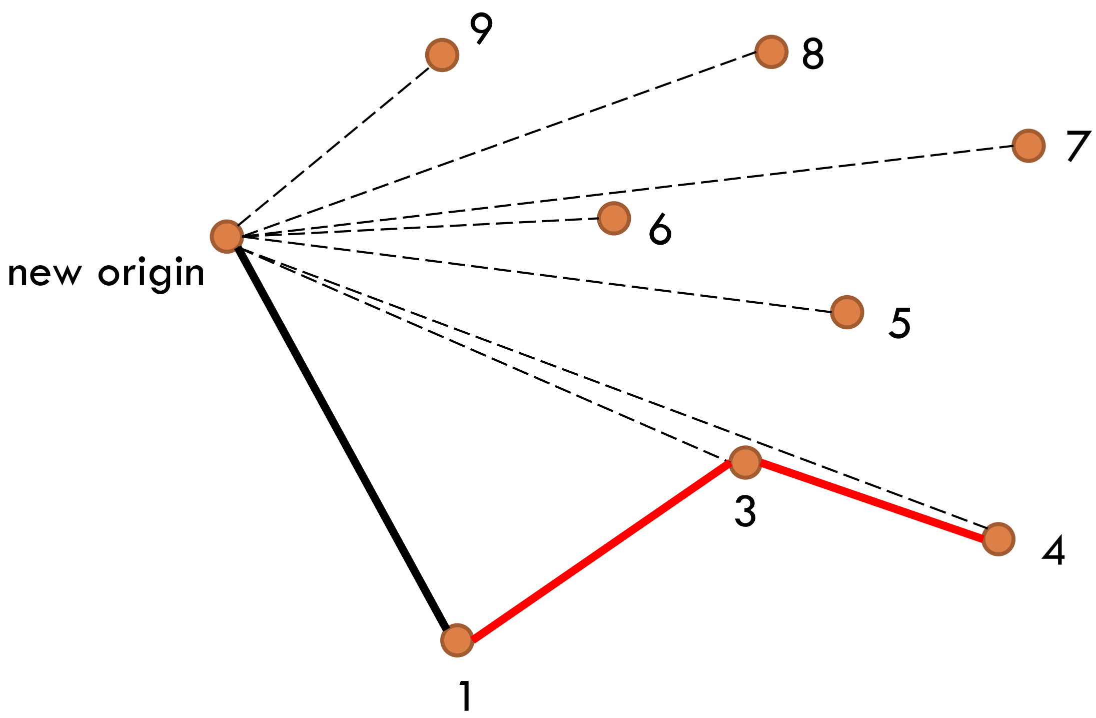
Example

- That's better…



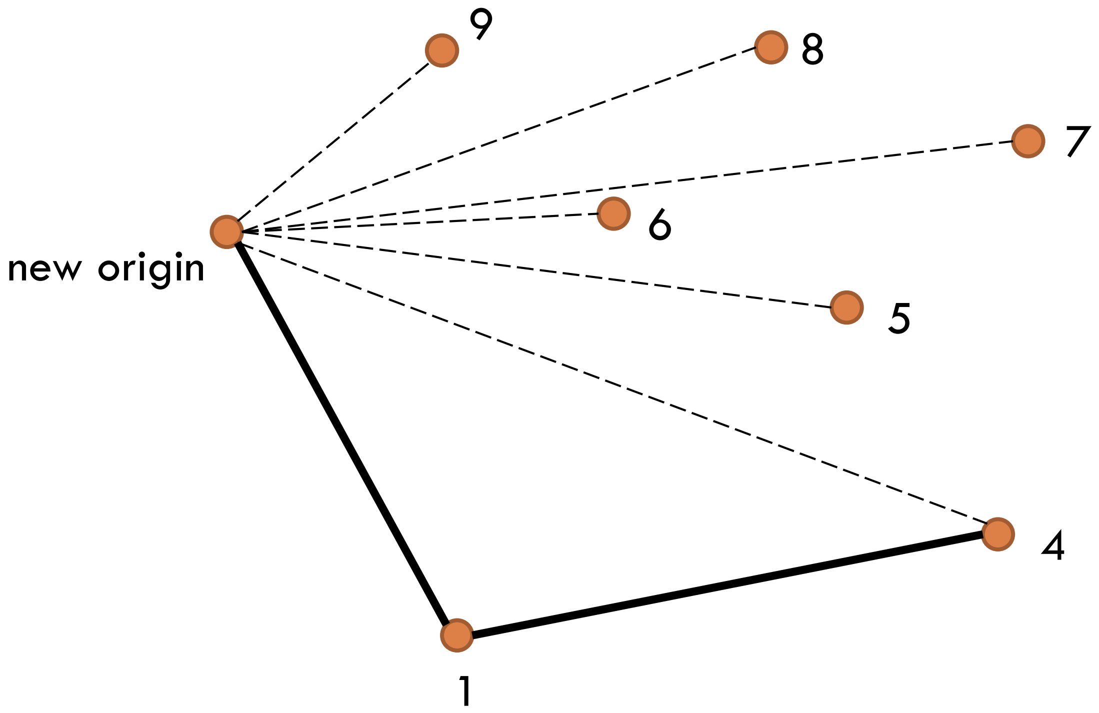
Example

- Adding 4 to the chain causes a problem
  - Remove 3



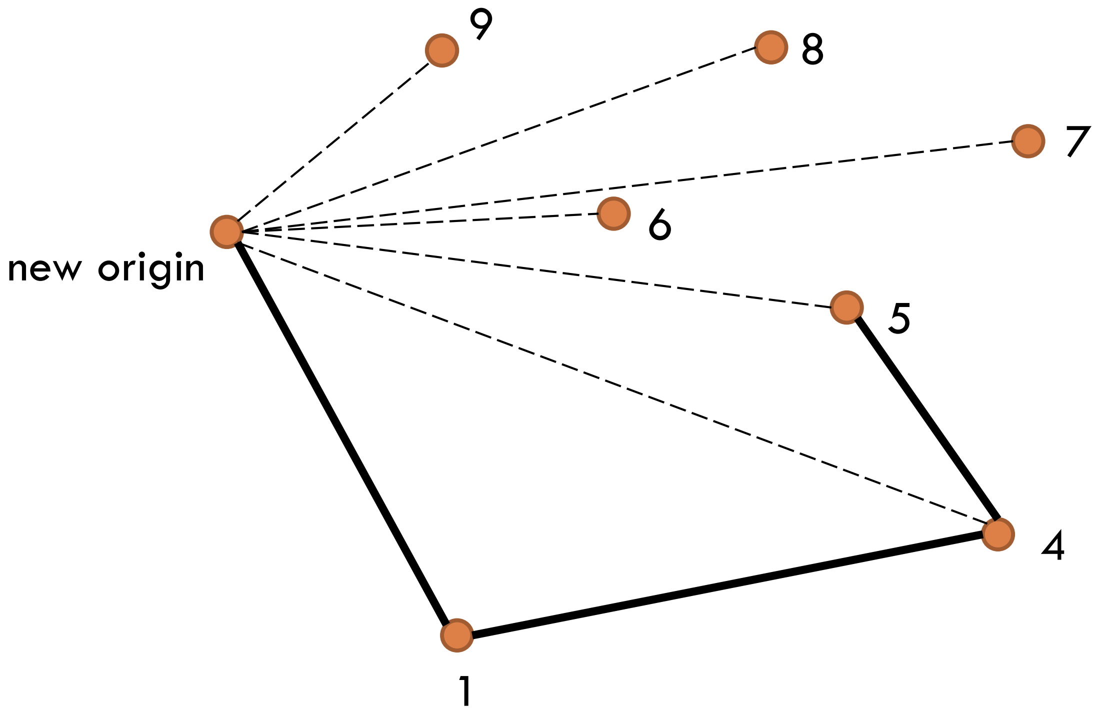
Example

- Continue adding points…



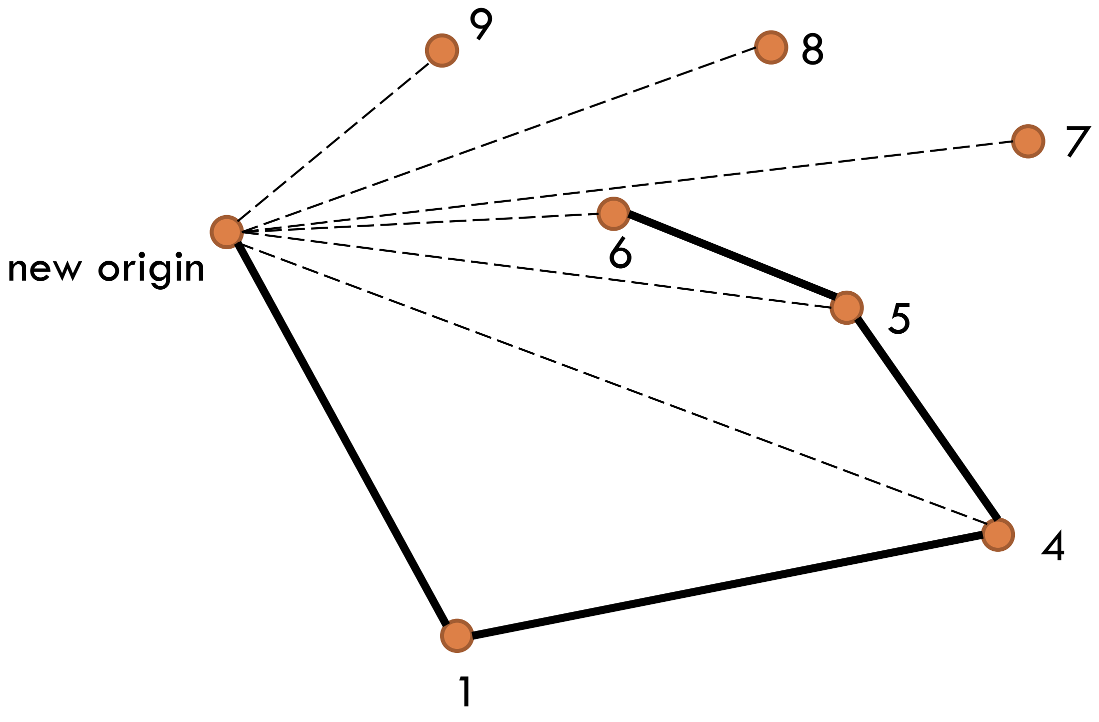
Example

- Continue adding points…



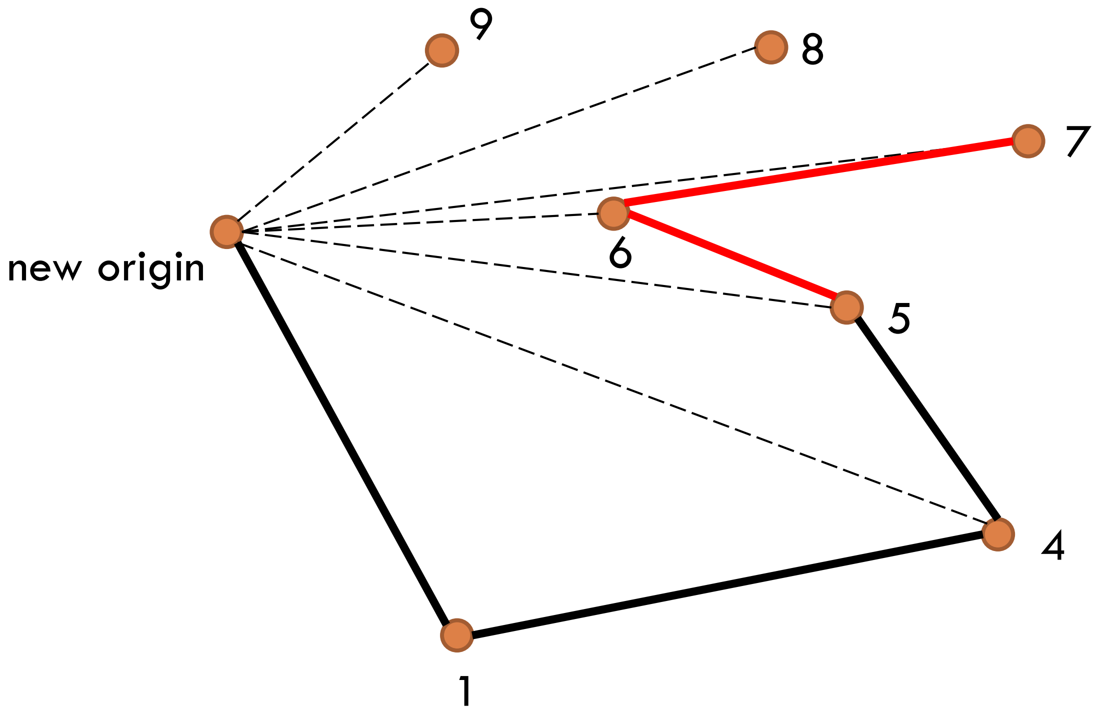
Example

- Continue adding points…



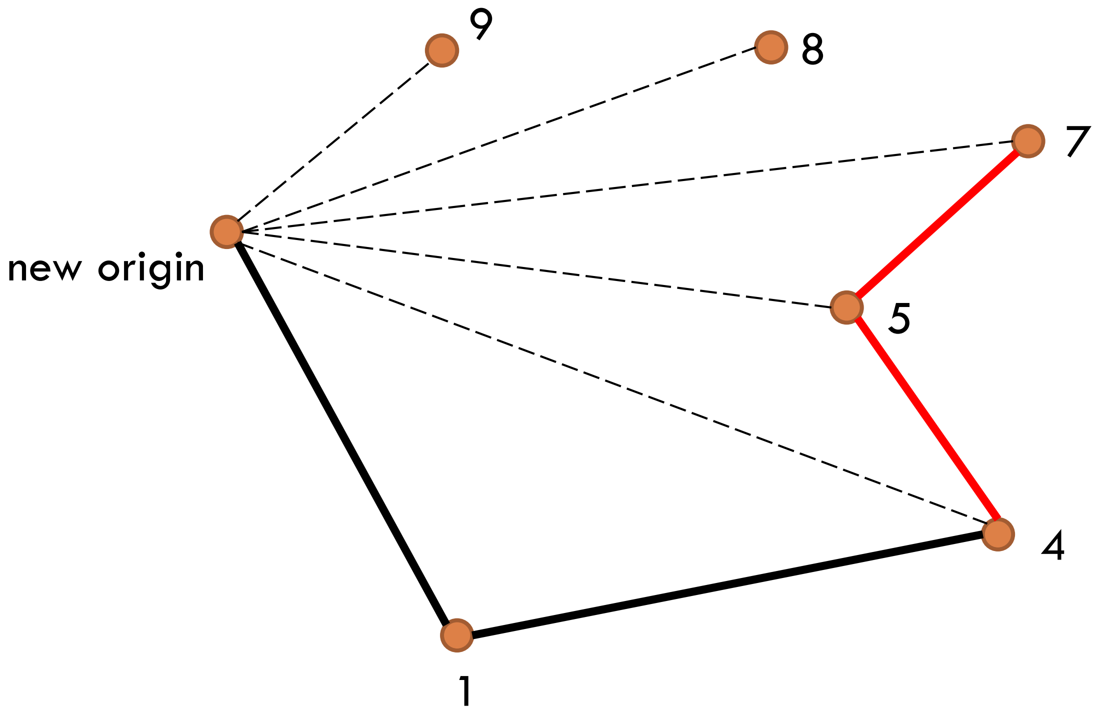
Example

- Bad corner!



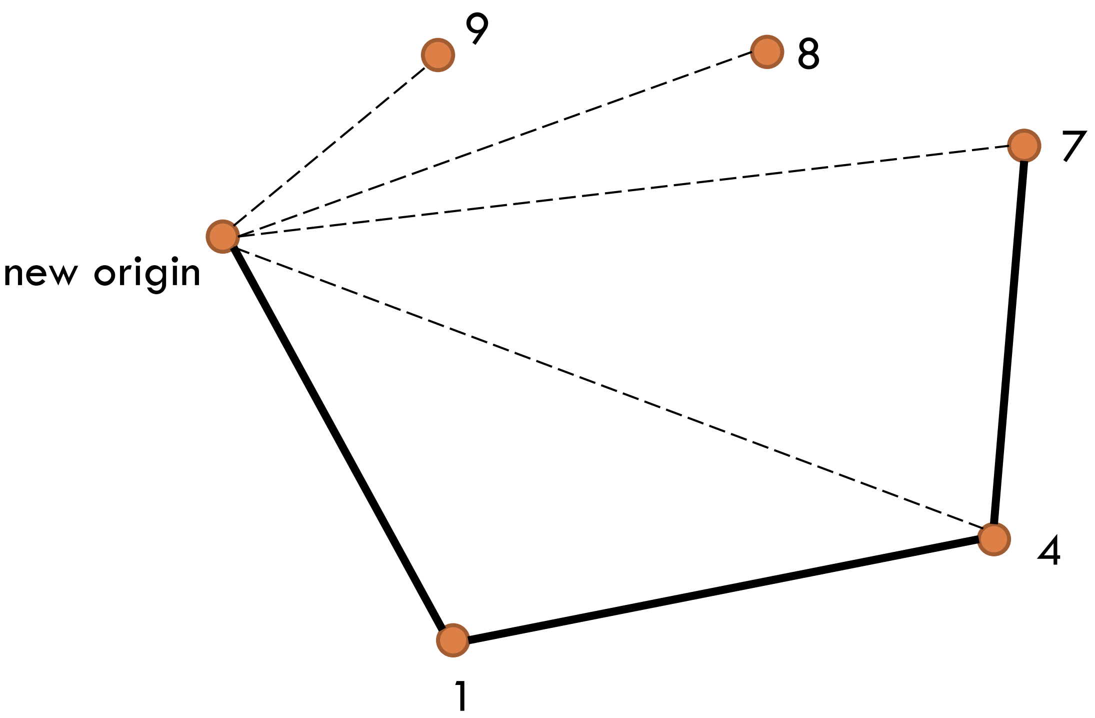
Example

- Bad corner again!



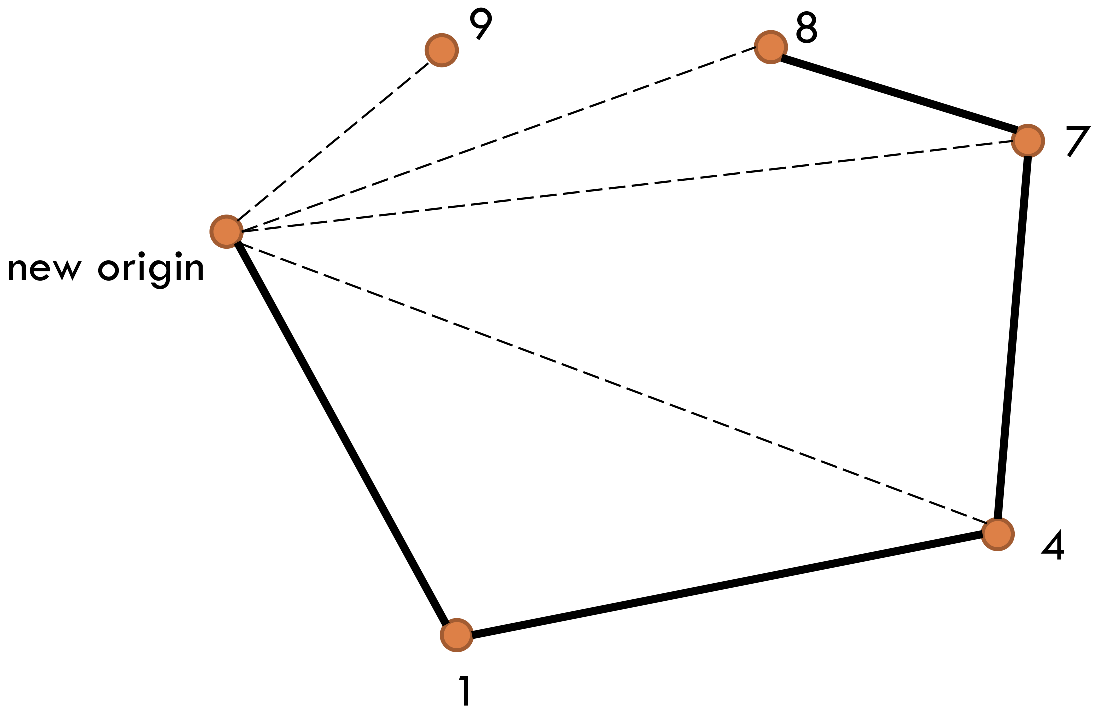
Example

- Continue adding points…



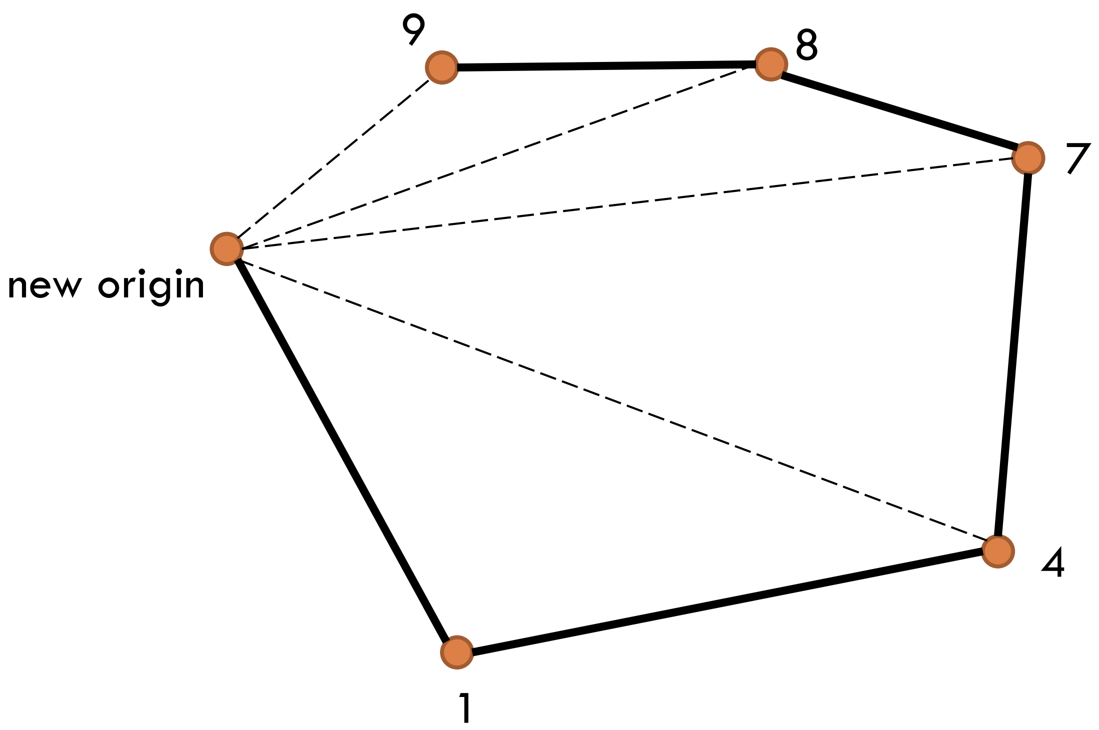
Example

- Continue adding points…



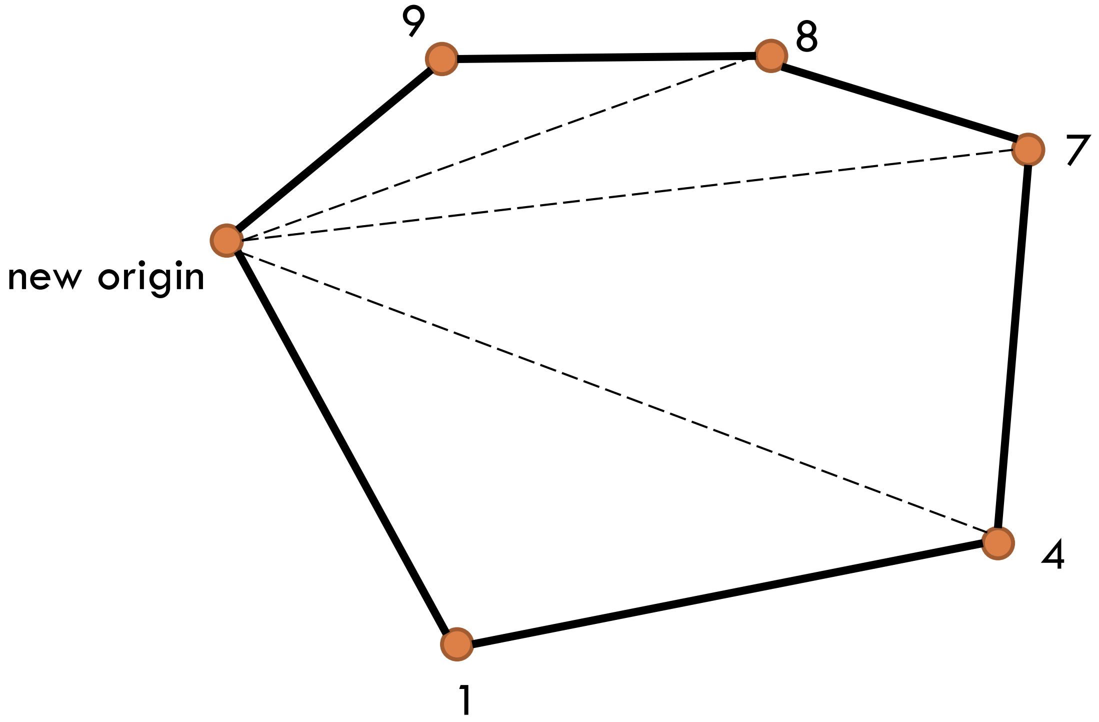
Example

- Continue adding points…



Example

- Done!



Pseudocode

- Set the leftmost point $(0,0)$, and sort the rest of the points in increasing order of y/x
- Initialize stack S
- For $i = 1 \dots n$:
  - Let A be the second topmost element of S, B be the topmost element of S, C be the ith point
  - If $\mathrm{ccw}(A, B, C) < 0$, pop S and go back
  - Push C to S
- Points in S form the convex hull

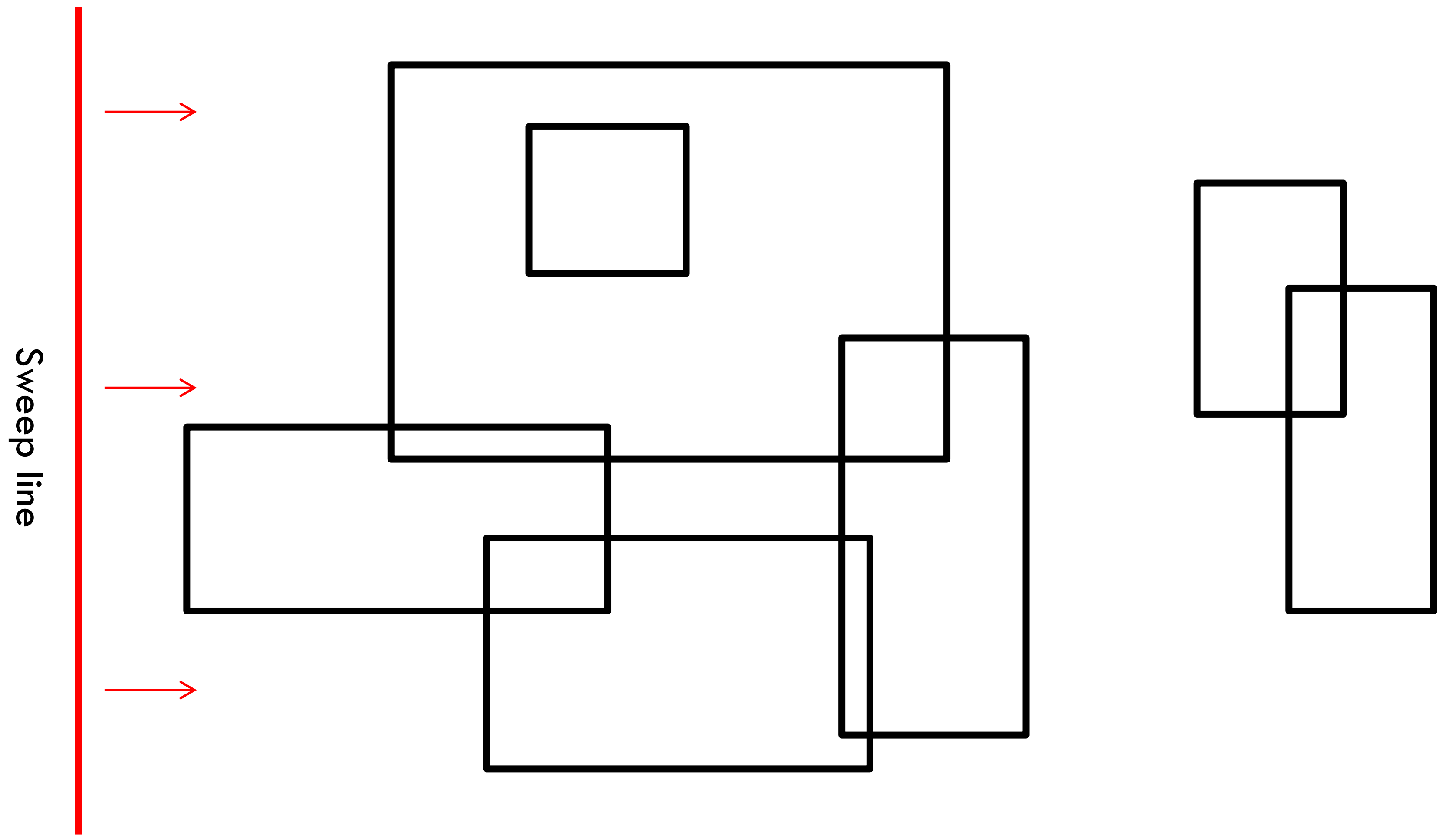
Sweep Line Algorithm

- A problem solving strategy for geometry problems
- The main idea is to maintain a line (with some auxiliary data structure) that sweeps through the entire plane and solve the problem locally
- We can't simulate a continuous process, (e.g. sweeping a line) so we define *events* that causes certain changes in our data structure
  - And process the events in the order of occurrence
- We'll cover one sweep line algorithm

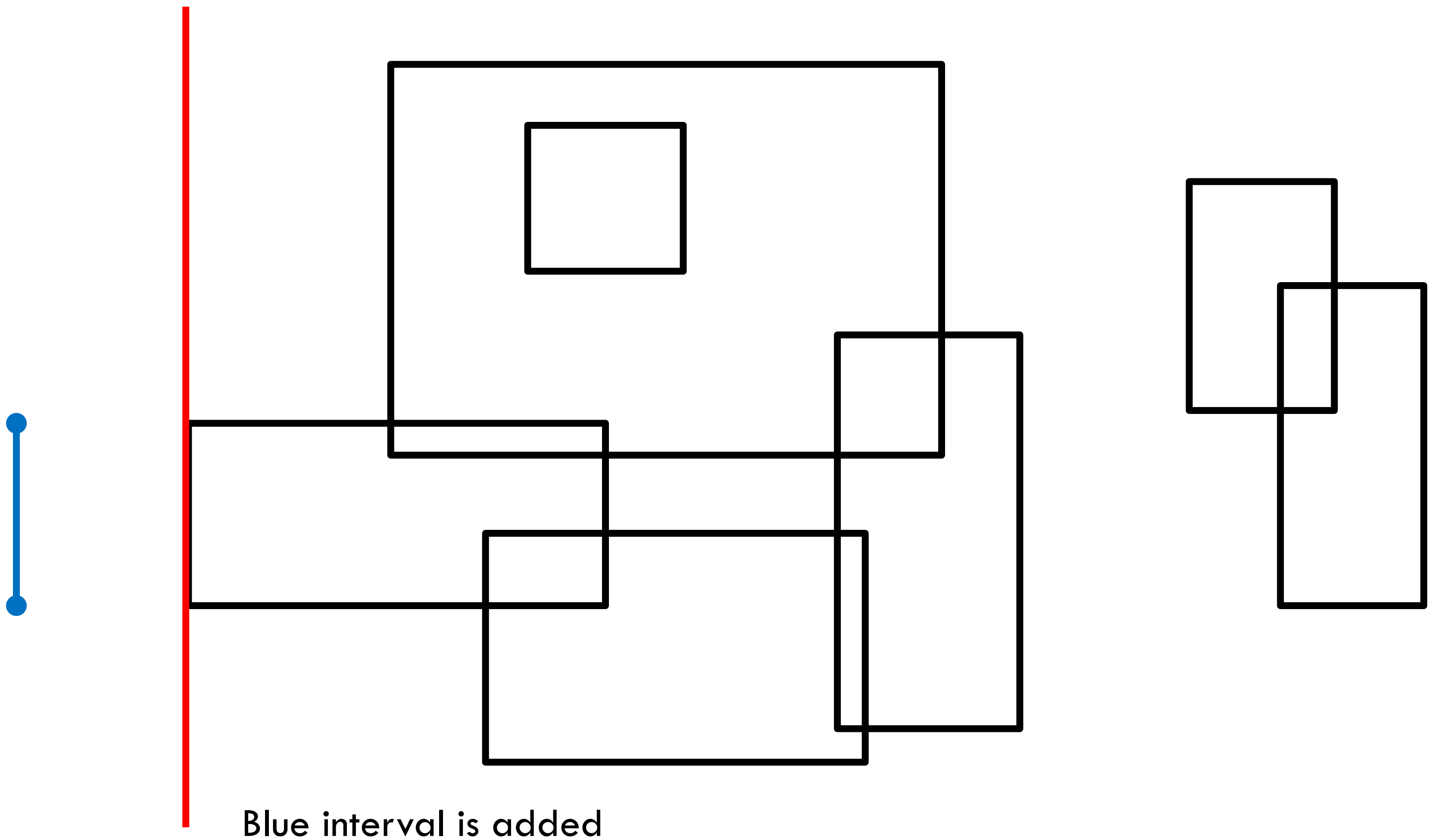
Sweep Line Algorithm

- Problem: Given n axis-aligned rectangles, find the area of the union of them
- We will sweep the plane from left to right
- Events: left and right edges of the rectangles
- The main idea is to maintain the set of "active" rectangles in order
  - It suffices to store the y-coordinates of the rectangles

Example

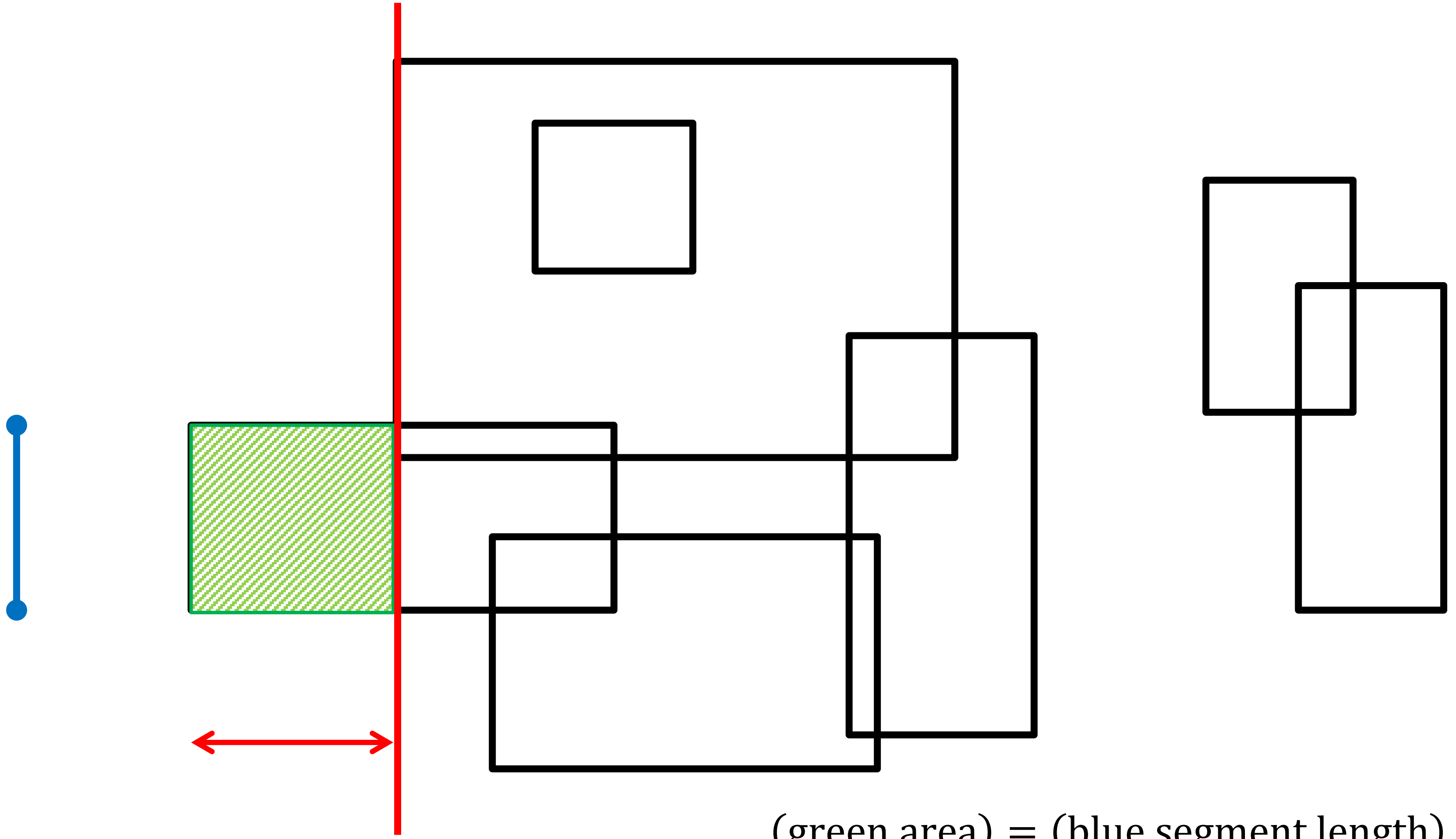


Example



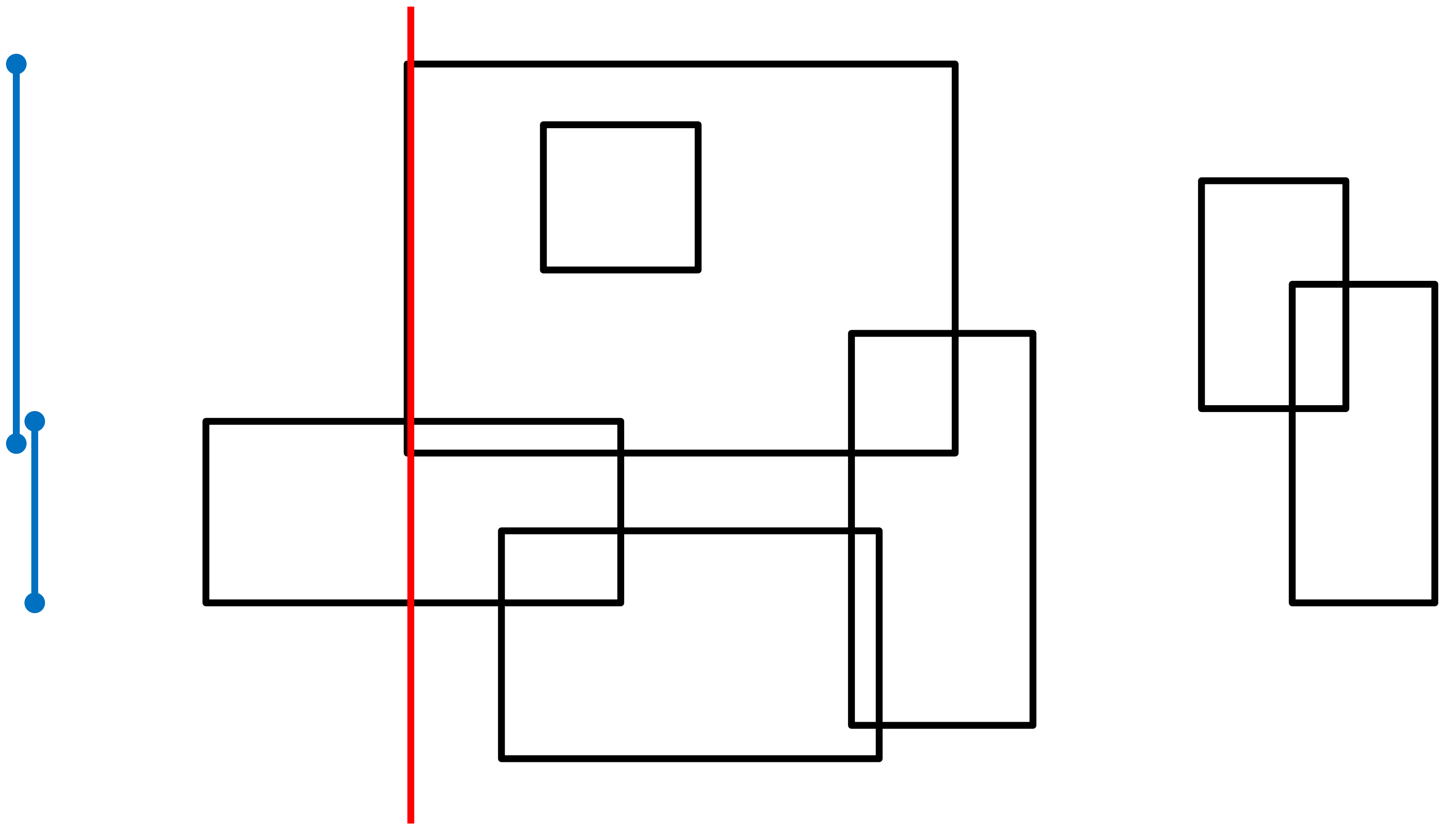
Blue interval is added
to the data structure

Example

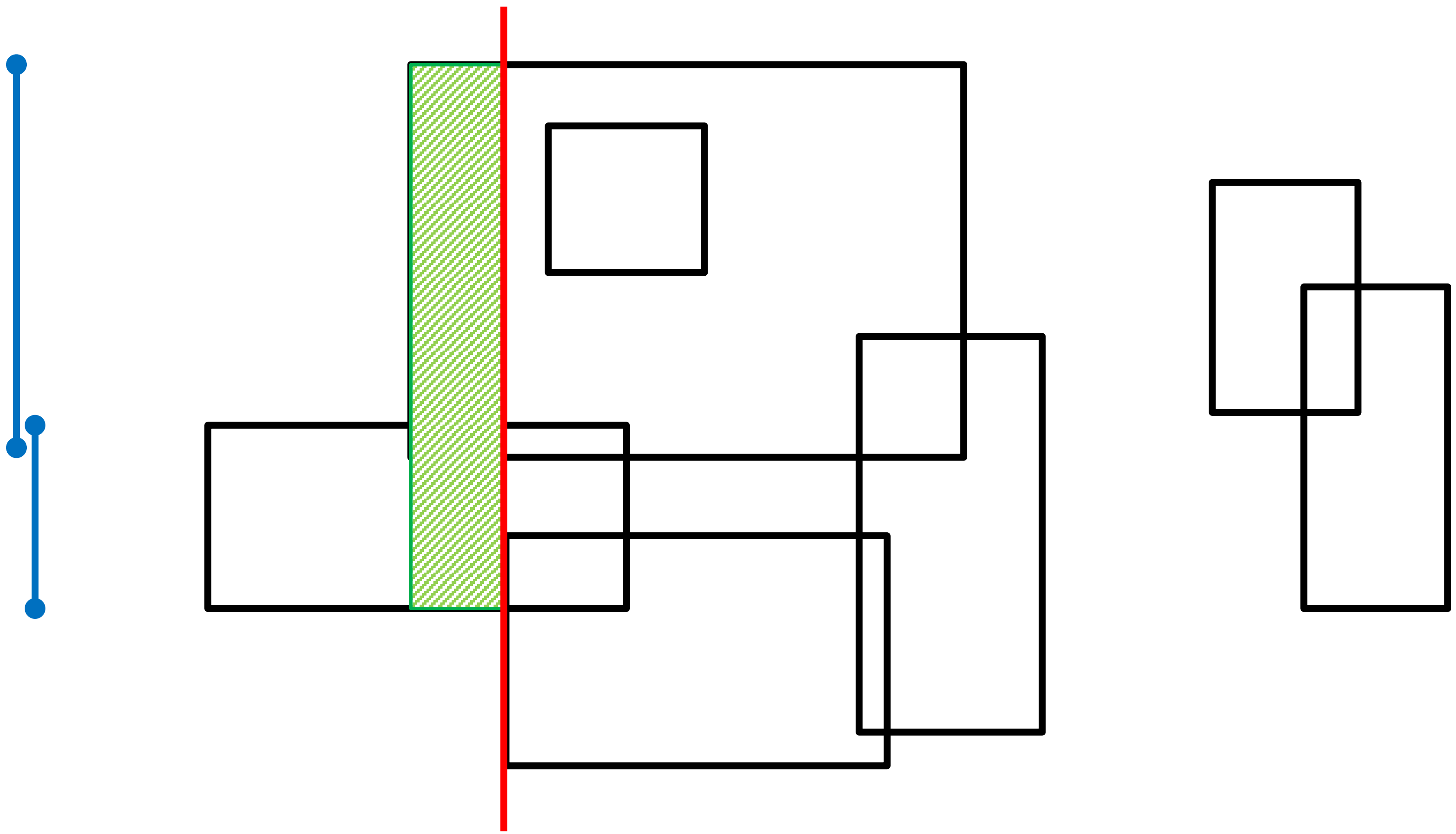


(green area) = (blue segment length)
× (distance that the sweep line traveled)

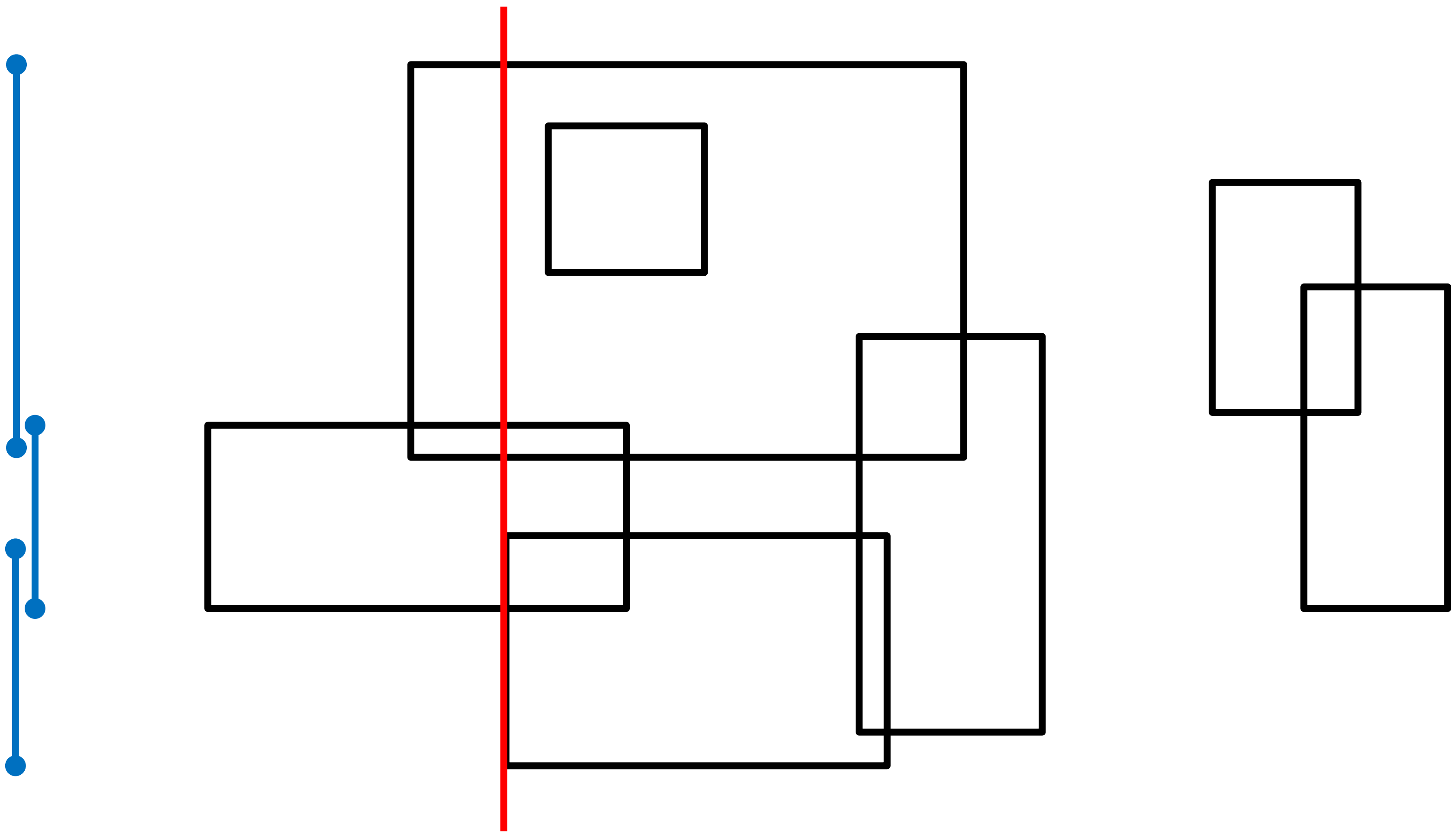
Example



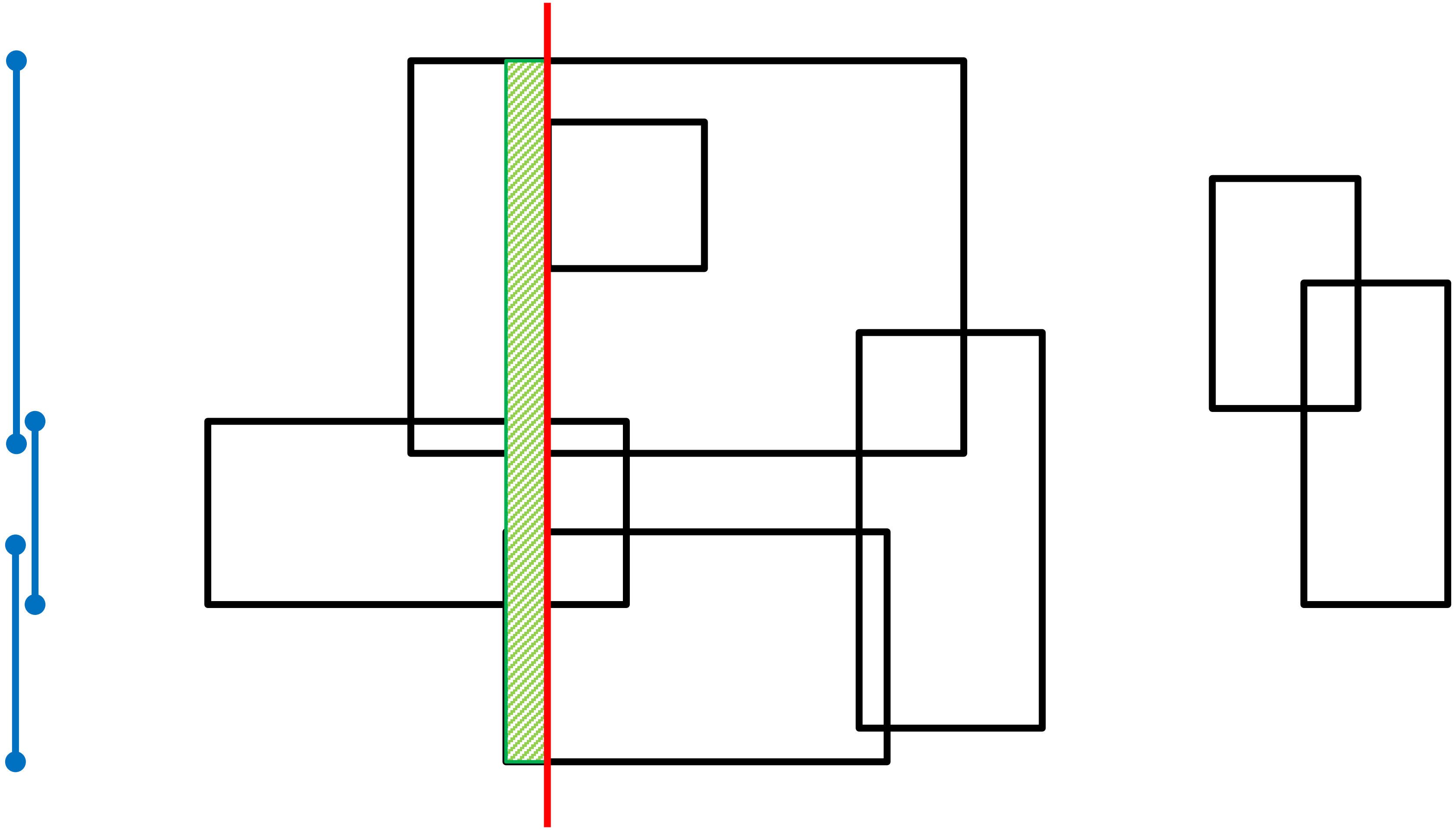
Example



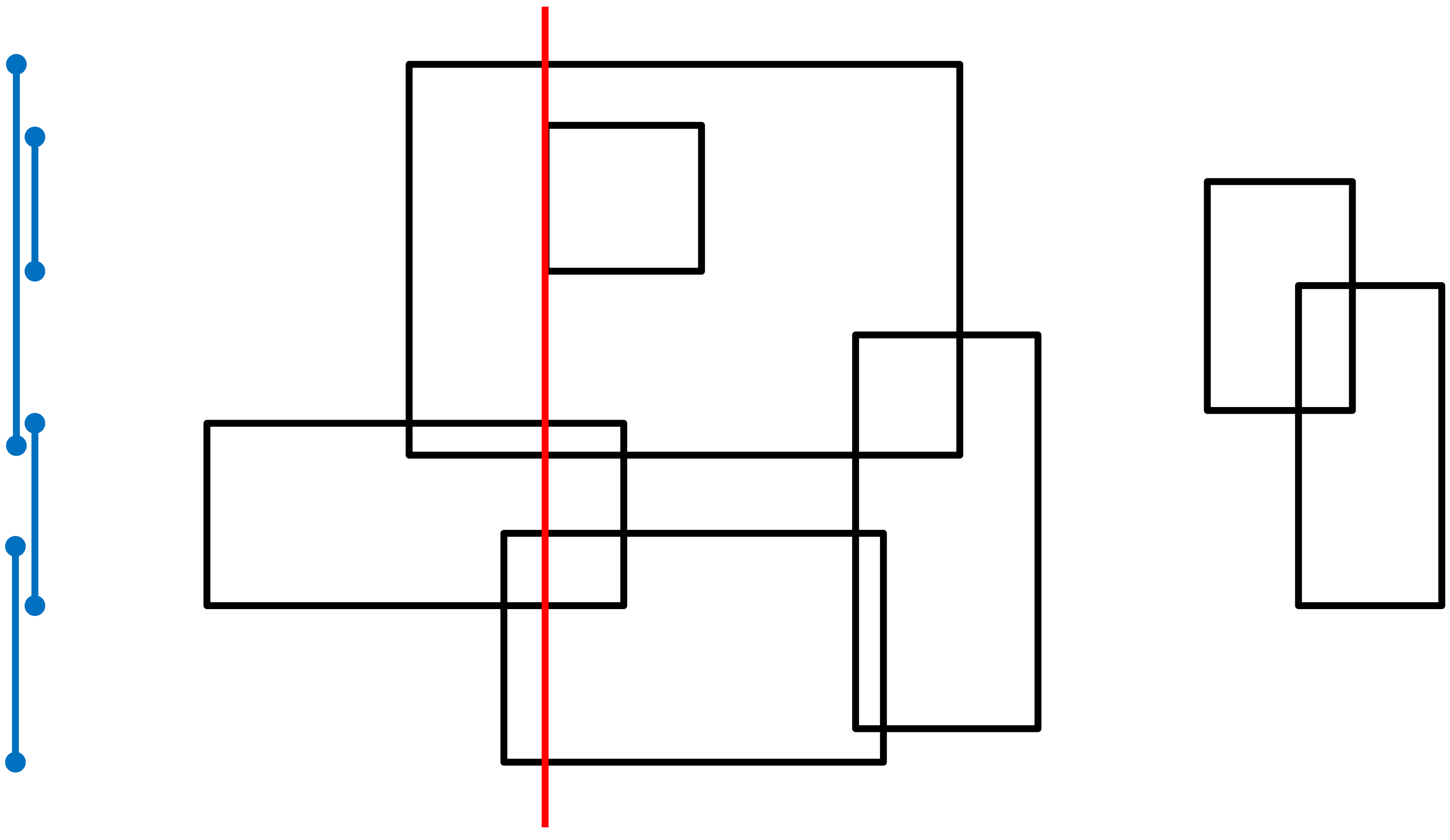
Example



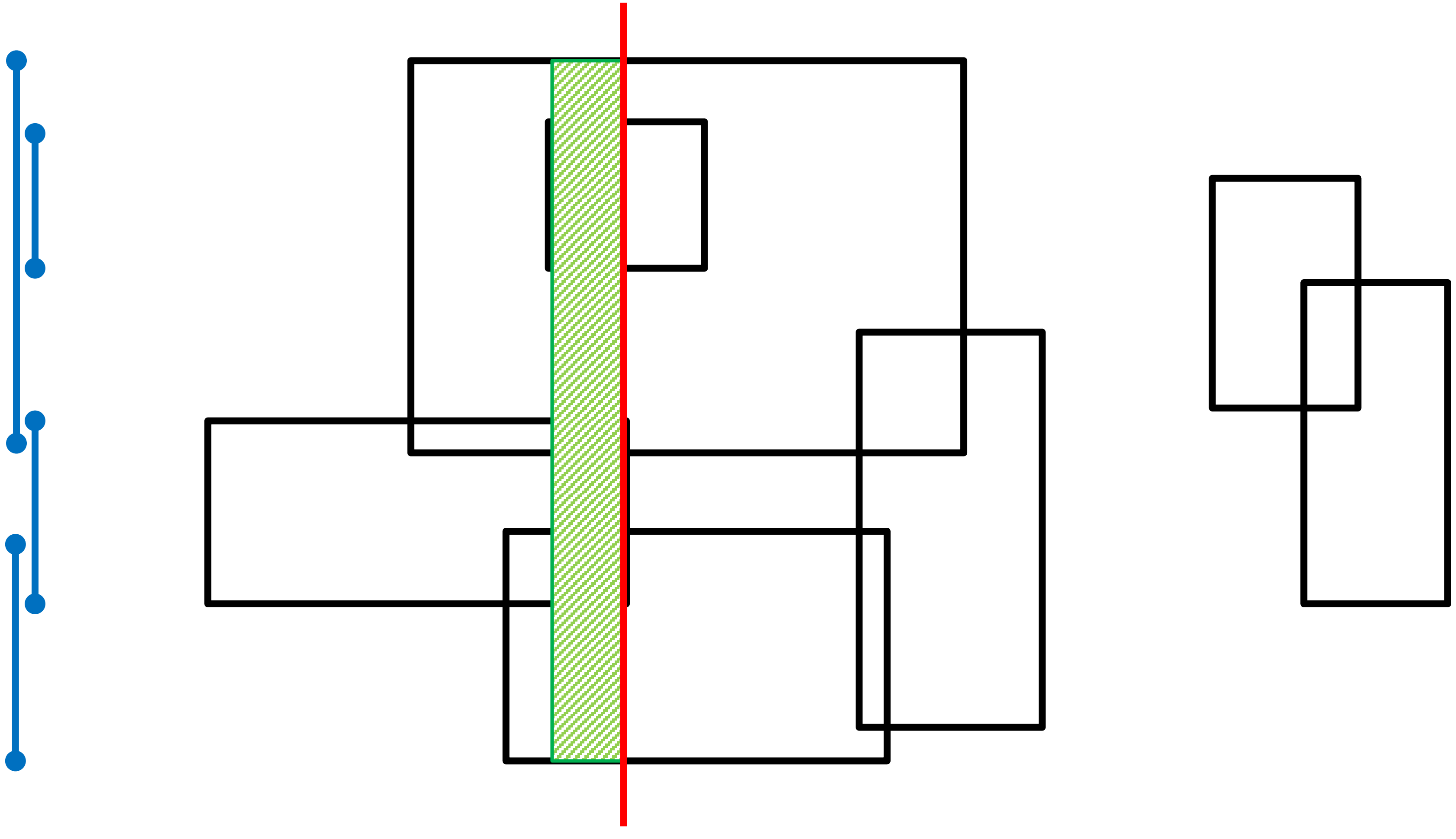
Example



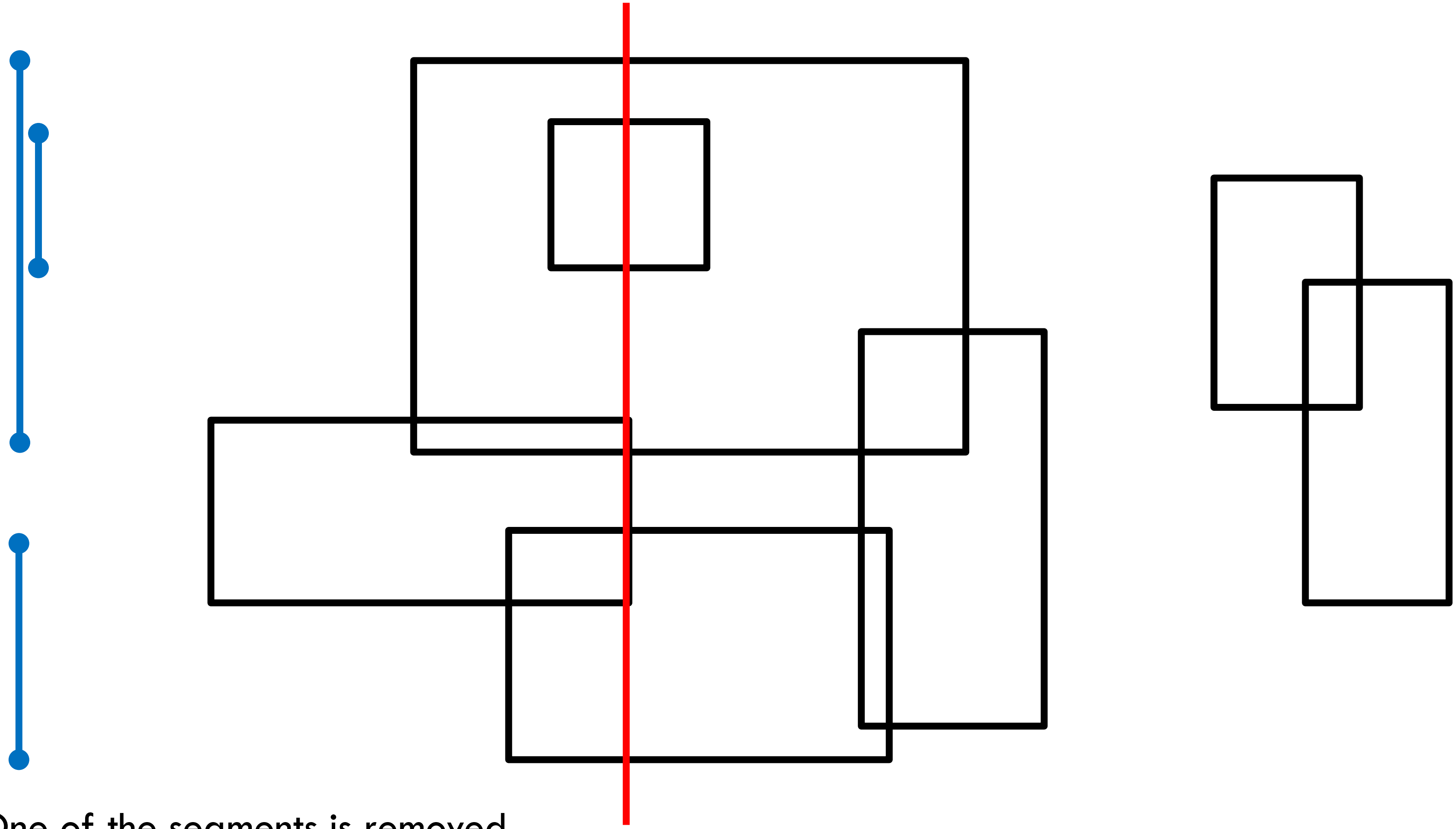
Example



Example

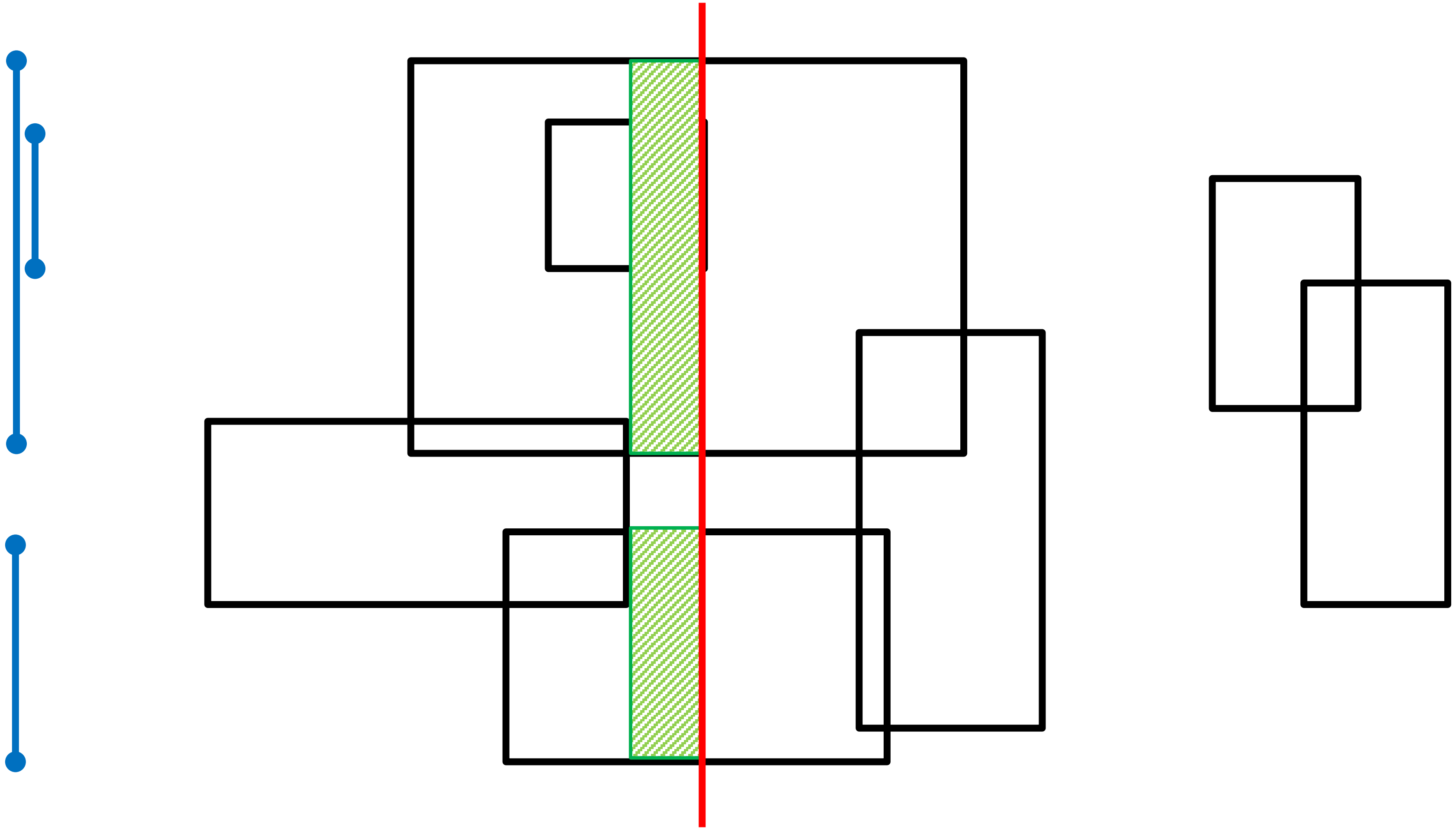


Example



One of the segments is removed

Example



Pseudopseudocode

- If the sweep line hits the left edge of a rectangle
  - Insert it to the data structure
- Right edge?
  - Remove it
- Move to the next event, and add the area(s) of the green rectangle(s)
  - Finding the length of the union of the blue segments is the hardest step
  - There is an easy $O(n)$ method for this step

Notes on Sweep Line Algorithms

- Sweep line algorithm is a generic concept
  - Come up with the right set of events and data structures for each problem
- Exercise problems
  - Finding the perimeter of the union of rectangles
  - Finding all k intersections of n line segments in $O\big((n+k)\log n\big)$ time

Intersecting Half-planes

- Representing a half-plane: $ax + by + c \leq 0$
- The intersection of half-planes is a convex area
  - If the intersection is bounded, it gives a convex polygon
- Given n half-planes, how do we compute the intersection of them?
  - i.e. Find vertices of the convex area
- There is an easy $O(n^3)$ algorithm and a hard $O(n \log n)$ one
  - We will cover the easy one

Intersecting Half-planes

- For each half-plane $a_i x + b_i y + c_i \leq 0$, define a straight line $e_i: a_i x + b_i y + c_i = 0$
- For each pair of e_i and e_j:
  - Compute their intersection $p = (p_x, p_y)$
  - Check if $a_k p_x + b_k p_y + c_k \leq 0$ for all half-planes
    - If so, store p in some array P
    - Otherwise, discard p
- Find the convex hull of the points in P

Intersecting Half-planes

- The intersection of half-planes can be unbounded
  - But usually, we are given limits on the min/max values of the coordinates
  - Add four half-planes $x \geq -M$, $x \leq M$, $y \geq -M$, $y \leq M$ (for large M) to ensure that the intersection is bounded
- Time complexity: $O(n^3)$
  - Pretty slow, but easy to code

Note on Binary Search

- Usually, binary search is used to find an item of interest in a sorted array
- There is a nice application of binary search, often used in geometry problems
  - Example: finding the largest circle that fits into a given polygon
    - Don't try to find a closed form solution or anything like that!
    - Instead, binary search on the answer

Ternary Search

- Another useful method in many geometry problems
- Finds the minimum point of a "convex" function f
  - Not exactly convex, but let's use this word anyway
- Initialize the search interval $[s, e]$
- Until $e - s$ becomes small:
  - $m_1 = s + (e - s)/3$, $m_2 = e - (e - s)/3$
  - If $f(m_1) \leq f(m_2)$, then set e to m_2
  - Otherwise, set s to m_1

CS 97SI: INTRODUCTION TO PROGRAMMING CONTESTS

Jaehyun Park

Last Lecture: String Algorithms

- String Matching Problem
- Hash Table
- Knuth-Morris-Pratt (KMP) Algorithm
- Suffix Trie
- Suffix Array
- Note on String Problems

String Matching Problem

- Given a text T and a pattern P, find all the occurrences of P within T
- Notations:
  - n and m: lengths of P and T
  - Σ: set of alphabets
    - Constant size
  - P_i: ith letter of P (1-indexed)
  - a, b, c: single letters in Σ
  - x, y, z: strings

String Matching Example

- $T =$ **AGCATGCTGCAGTCATGCTTAGGCTA**
- $P =$ **GCT**

- A naïve method takes $O(nm)$ time
  - We initiate string comparison at every starting point
  - Each comparison takes $O(m)$ time

- We can certainly do better!

Hash Function

- A function that takes a string and outputs a number
- A good hash function has few collisions
  - i.e. If $x \neq y, H(x) \neq H(y)$ with high probability
- An easy and powerful hash function is a polynomial mod some prime p
  - Consider each letter as a number (ASCII value is fine)
  - $H(x_1 \dots x_k) = x_1 a^{k-1} + x_2 a^{k-2} + \cdots + x_{k-1} a + x_k$
  - How do we find $H(x_2 \dots x_{k+1})$ from $H(x_1 \dots x_k)$?

Hash Table

- Main idea: preprocess T to speedup queries
  - Hash every substring of length k
  - k is a small constant

- For each query P, hash the first k letters of P to retrieve all the occurrences of it within T

- Don't forget to check collisions!

Hash Table

- Pros:
  - Easy to implement
  - Significant speedup in practice

- Cons:
  - Doesn't help the asymptotic efficiency
    - Can take $\Theta(nm)$ time if hashing is terrible
  - A lot of memory consumption

Knuth-Morris-Pratt (KMP) Matcher

- A linear time (!) algorithm that solves the string matching problem by preprocessing P in $\Theta(m)$ time
  - Main idea is to skip some comparisons by using the previous comparison result
- Uses an auxiliary array π that is defined as the following:
  - $\pi[i]$ is the largest integer smaller than i such that $P_1 \dots P_{\pi[i]}$ is a suffix of $P_1 \dots P_i$
- … It's better to see an example than the definition

π Table Example (from CLRS)

i	1	2	3	4	5	6	7	8	9	10
P_i	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

- $\pi[i]$: the largest integer smaller than i such that $P_1 \ldots P_{\pi[i]}$ is a suffix of $P_1 \ldots P_i$
  - e.g. $\pi[6] = 4$ since abab is a suffix of ababab
  - e.g. $\pi[9] = 0$ since no prefix of length ≤ 8 ends with c
- Let's see why this is useful

Using the π Table

- $T =$ **ABC ABCDAB ABCDABCDABDE**
- $P =$ **ABCDABD**
- $\pi = (0, 0, 0, 0, 1, 2, 0)$
- Start matching at the first position of T:

```
12345678901234567890123
ABC ABCDAB ABCDABCDABDE
ABCDABD
1234567
```

- Mismatch at the 4th letter of P!

Using the π Table

- There is no point in starting the comparison at T_2, T_3
  - We matched $k = 3$ letters so far
  - Shift P by $k - \pi[k] = 3$ letters

```
12345678901234567890123
ABC ABCDAB ABCDABCDABDE
   ABCDABD
   1234567
```

- Mismatch at T_4 again!

Using the π Table

- We define $\pi[0] = -1$
  - We matched $k = 0$ letters so far
  - Shift P by $k - \pi[k] = 1$ letter

```
12345678901234567890123
ABC ABCDAB ABCDABCDABDE
    ABCDABD
    1234567
```

- Mismatch at T_{11}!

Using the π Table

- $\pi[6] = 2$ says $P_1 P_2$ is a suffix of $P_1 \dots P_6$
- Shift P by $6 - \pi[6] = 4$ letters

```
12345678901234567890123
ABC ABCDAB ABCDABCDABDE
    ABCDABD
        ||
        ABCDABD
        1234567
```

- Again, no point in shifting P by 1, 2, or 3 letters

Using the π Table

- Mismatch at T_{11} again!

```
12345678901234567890123
ABC ABCDAB ABCDABCDABDE
          ABCDABD
          1234567
```

- Currently 2 letters are matched
- We shift P by $2 = 2 - \pi[2]$ letters

Using the π Table

- Mismatch at T_{11} yet again!

```
12345678901234567890123
ABC ABCDAB ABCDABCDABDE
          ABCDABD
          1234567
```

- Currently no letters are matched
- We shift P by $1 = 0 - \pi[0]$ letters

Using the π Table

- Mismatch at T_{18}

```
12345678901234567890123
ABC ABCDAB ABCDABCDABDE
           ABCDABD
           1234567
```

- Currently 6 letters are matched
- We shift P by $4 = 6 - \pi[6]$ letters

Using the π Table

- Finally, there is it!

```
12345678901234567890123
ABC ABCDAB ABCDABCDABDE
               ABCDABD
               1234567
```

- Currently all 7 letters are matched
- After recording this match (match at $T_{16} \dots T_{22}$), we shift P again in order to find other matches
  - Shift by $7 = 7 - \pi[7]$ letters

Computing π

- Observation 1: if $P_1 \dots P_{\pi[i]}$ is a suffix of $P_1 \dots P_i$, then $P_1 \dots P_{\pi[i]-1}$ is a suffix of $P_1 \dots P_{i-1}$
  - Well, obviously…
- Observation 2: all the prefixes of P that are a suffix of $P_1 \dots P_i$ can be obtained by recursively applying π to i
  - e.g. $P_1 \dots P_{\pi[i]}$, $P_1 \dots P_{\pi[\pi[i]]}$, $P_1 \dots P_{\pi\left[\pi[\pi[i]]\right]}$ are all suffixes of $P_1 \dots P_i$

Computing π

- A non-obvious conclusion:
  - First, let's write $\pi^{(k)}[i]$ as $\pi[\cdot]$ applied k times to i
  - e.g. $\pi^{(2)}[i] = \pi\big[\pi[i]\big]$
  - $\pi[i]$ is equal to $\pi^{(k)}[i-1]+1$, where k is the smallest integer that satisfies $P_{\pi^{(k)}[i-1]+1} = P_i$
    - If there is no such k, $\pi[i] = 0$
- Intuition: we look at all the prefixes of P that are suffixes of $P_1 \dots P_{i-1}$ and find the longest one whose next letter matches P_i too

Implementation

```
pi[0] = -1;
int k = -1;
for(int i = 1; i <= m; i++) {
  while(k >= 0 && P[k+1] != P[i])
    k = pi[k];
  pi[i] = ++k;
}
```

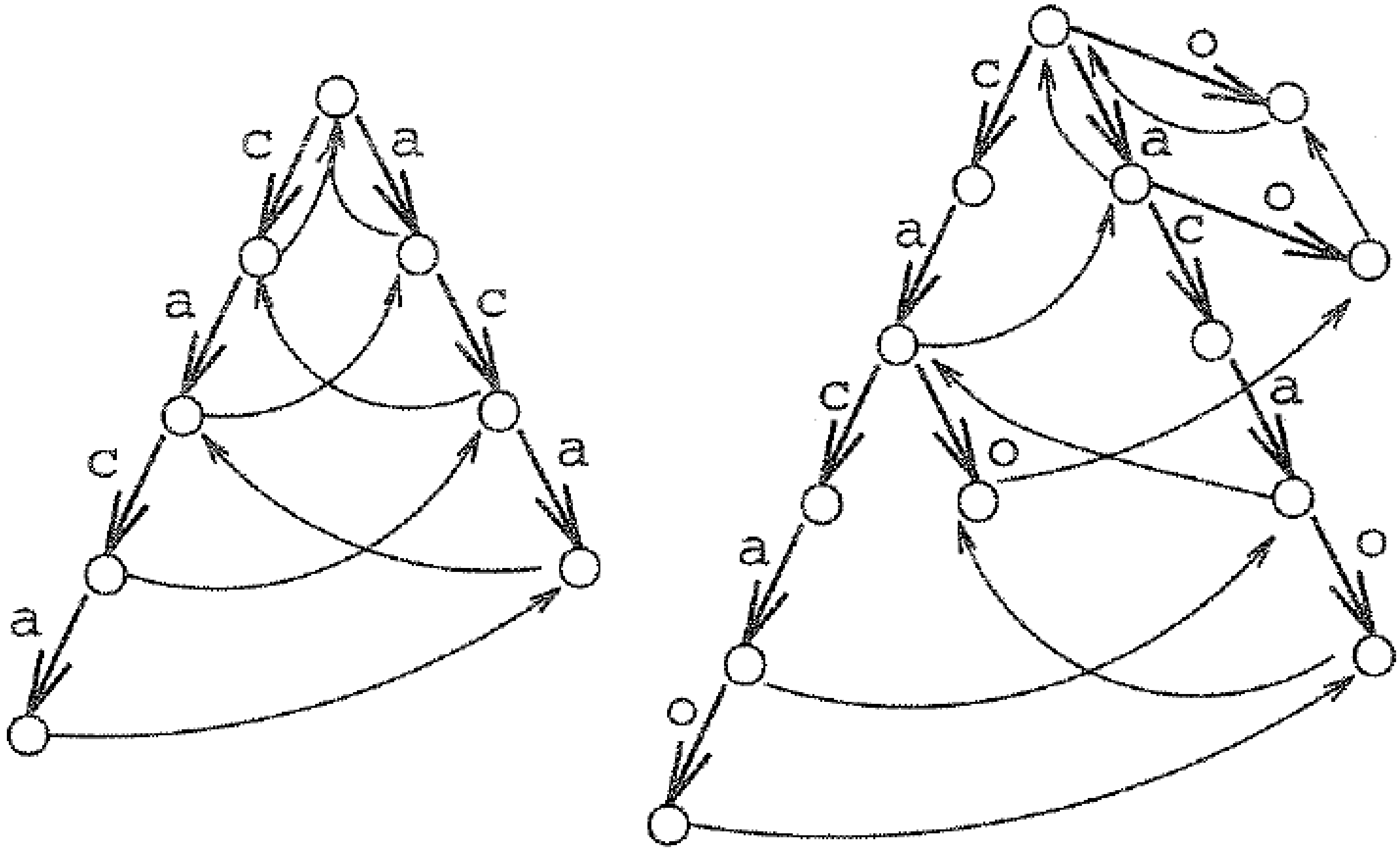
Pattern Matching Implementation

```
int k = 0;
for(int i = 1; i <= n; i++) {
  while(k >= 0 && P[k+1] != T[i])
    k = pi[k];
  k++;
  if(k == m) {
    // P matches T[i-m+1..i]
    k = pi[k];
  }
}
```

Suffix Trie

- Suffix trie of a string T is a rooted tree that stores all the suffixes (thus all the substrings)
- Each node corresponds to some substring of T
- Each edge is associated with an alphabet
- For each node that corresponds to ax, there is a special pointer called *suffix link* that leads to the node corresponding to x
- Surprisingly easy to implement!

Suffix Trie Example



(Figure modified from Ukkonen's original paper)

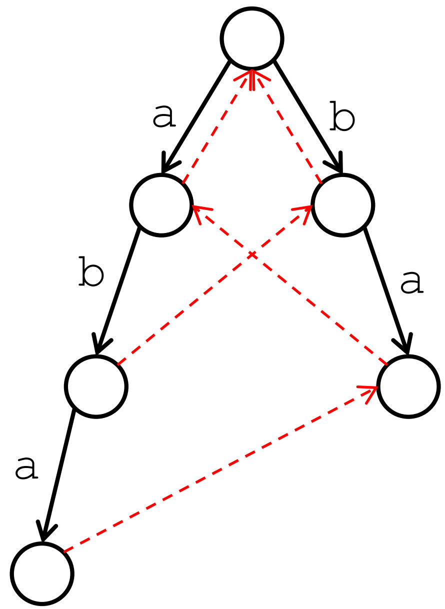
Incremental Construction

- Given the suffix tree for $T_1 \dots T_n$
  - Then we append $T_{n+1} = a$ to T, creating necessary nodes
- Start at node u corresponding to $T_1 \dots T_n$
  - Create an a-transition to a new node v
- Take the suffix link at u to go to u', corresponding to $T_2 \dots T_n$
  - Create an a-transition to a new node v'
  - Create a suffix link from v to v'

Incremental Construction

- We repeat the previous process:
  - Take the suffix link at the current node
  - Make a new a-transition there
  - Create the suffix link from the previous node
- We stop if the node already has an a-transition
  - Because from this point, all nodes that are reachable via suffix links already have an a-transition

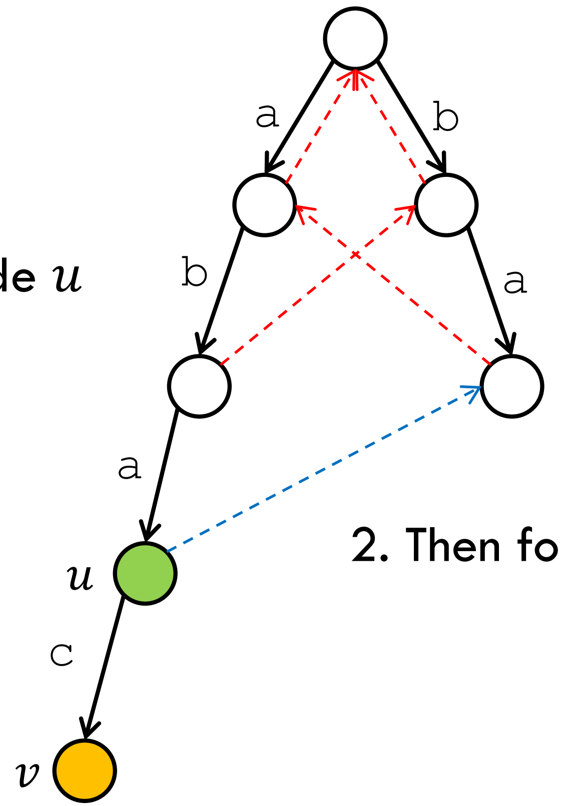
Construction Example



Given the suffix trie for `aba`
We want to add a new letter `c`

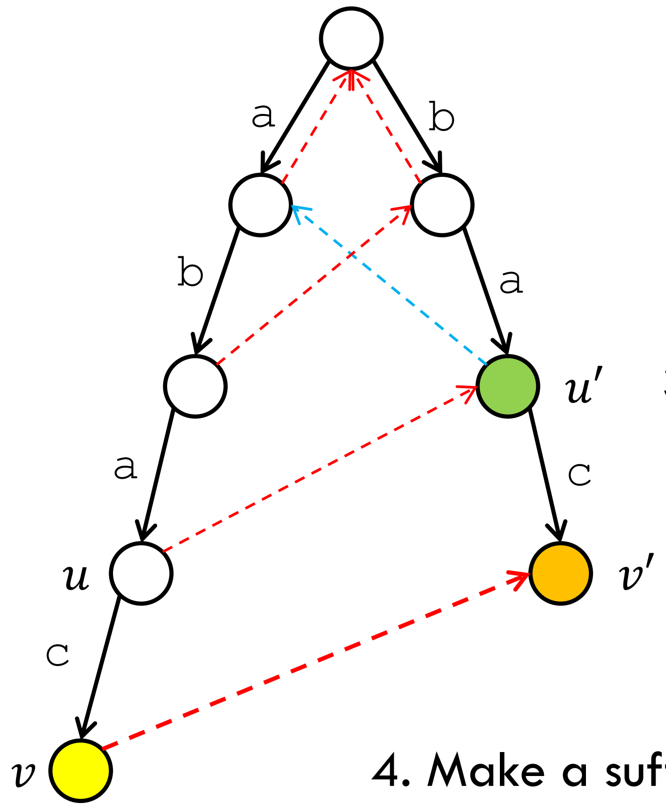
Construction Example

1. Start at the green node u and make a $\mathtt{c}$-transition



2. Then follow the suffix link

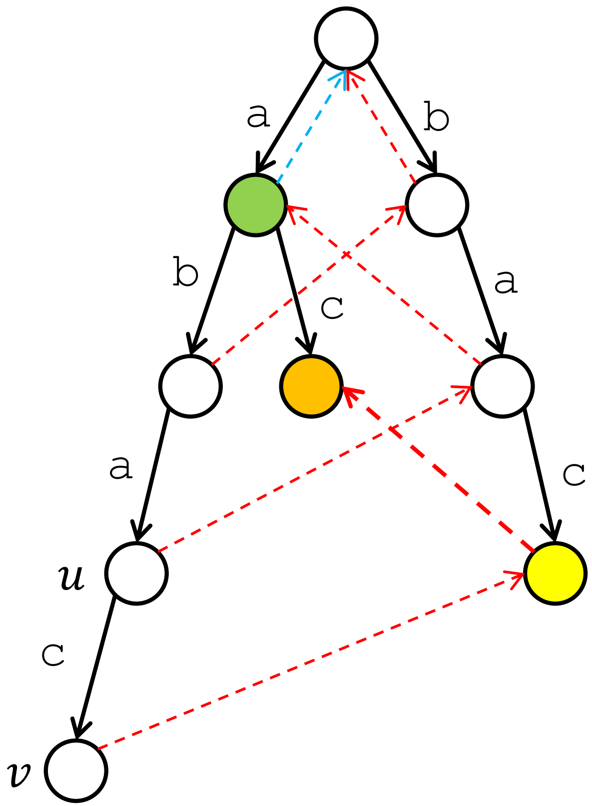
Construction Example



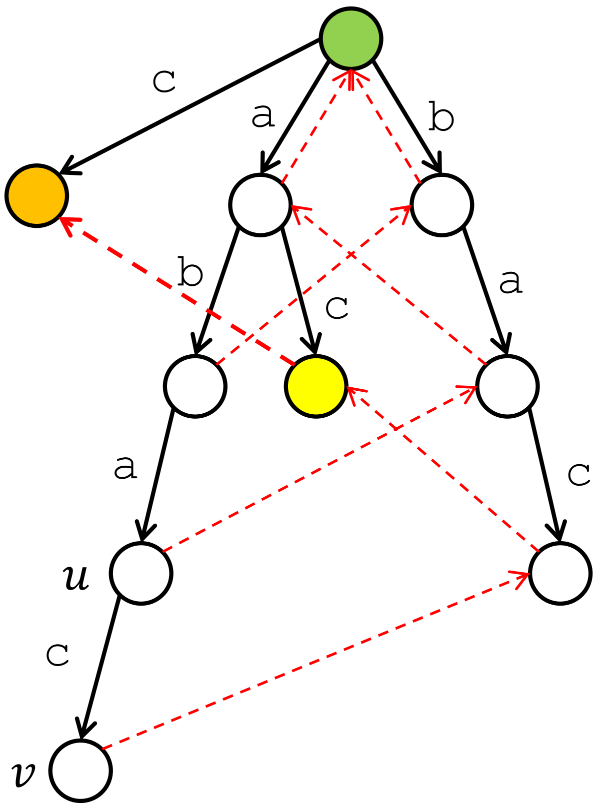
3. Make a `c`-transition at u'

4. Make a suffix link from v

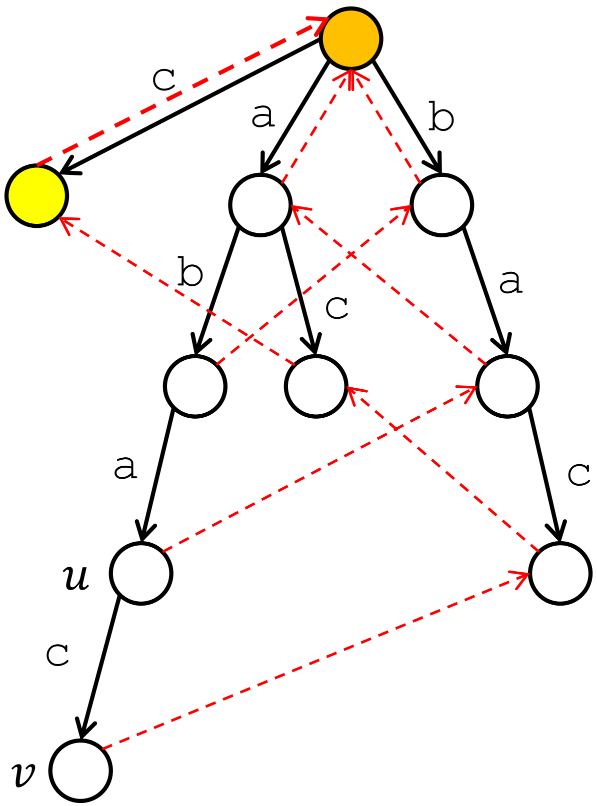
Construction Example



Construction Example



Construction Example



Suffix Trie Analysis

- Construction time is linear in the tree size
  - But the tree size can be quadratic in n
    - e.g. $T = \mathtt{aa...abb...b}$

Pattern Matching

- To find P, start at the root and keep following edges labeled with P_1, P_2, etc.
- Got stuck? Then P doesn't exist in T

Suffix Array

Input string	Get all suffixes	Sort the suffixes	Take the indices
	1 BANANA	6 A	
	2 ANANA	4 ANA	
	3 NANA	2 ANANA	
BANANA	4 ANA	1 BANANA	6,4,2,1,5,3
	5 NA	5 NA	
	6 A	3 NANA	

Suffix Array

- Memory usage is $O(n)$
- Has the same computational power as suffix trie
- Can be constructed in $O(n)$ time (!)
  - But it's hard to implement
- There is an approachable $O(n \log^2 n)$ algorithm
  - If you want to see how it works, read the paper on the course website
  - http://cs97si.stanford.edu/suffix-array.pdf

Note on String Problems

- Always be aware of the null-terminators
- Simple hash works so well in many problems
  - Even for problems that aren't supposed to be solved by hashing
- If a problem involves rotations of a string, consider concatenating it with itself and see if it helps
- Stanford team notebook has implementations of suffix arrays and the KMP matcher