CSC373: Algorithm Design, Analysis and Complexity Fall 2017

DENIS PANKRATOV

NOVEMBER 22, 2017

Coping with NP-hardness

You proved your problem of interest is NP-hard

Now what?

In practice, you still need to solve problems

The high level goal is to get good enough solutions by any means necessary

“Good enough” = approximation algorithms

You could simply attack the problem of designing algorithms head on – invent something ad-hoc

Or you could try out some algorithmic paradigms – general methods that seem to work well for other problems, so they may be adapted to work for your problem too

Coping with NP-hardness

Do you know who else needs to routinely solve NP-hard problems?

NATURE!

Borrow some inspiration for algorithmic paradigms from it

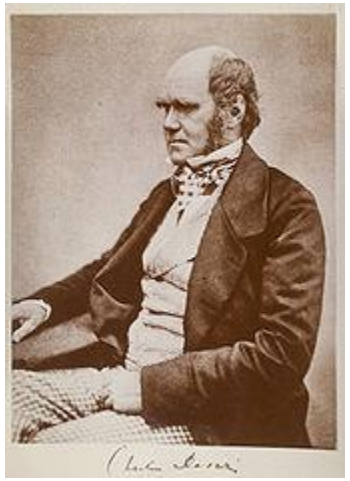
One of its most successful algorithmic paradigms is evolution:

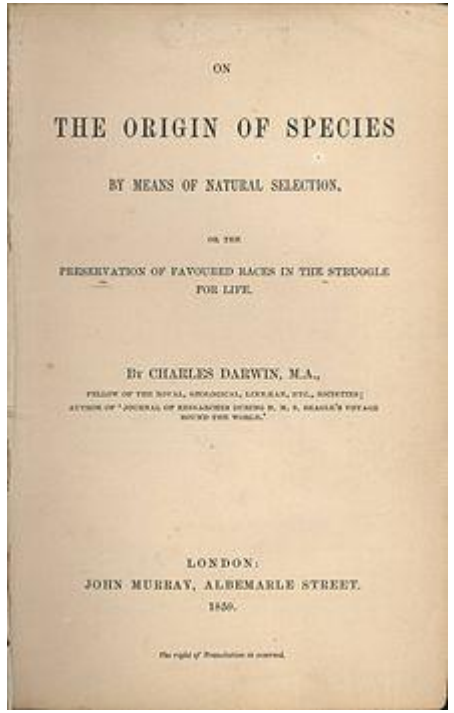
Incremental process of

(1) introducing small mutations,

(2) trying them out, and

(3) keeping them if they work well.





Local Search Algorithmic Paradigm

Start with some **solution** S
While there is a "better" solution S' in the **local neighborhood** of S
    update S to S'

possible small mutations ⇔ local neighborhood

trying mutations out ⇔ enumerating the entire local neighborhood

keeping mutations if they work well ⇔ updating S to S' if S' is better than S

Set of all possible solutions is called **solution space** or **state space**

Not to be confused with so-called "genetic algorithms" that mutate several solutions simultaneously and mutations might depend on several solutions

Local Search Applications

(1) As an approximation algorithmic paradigm with provable guarantees

(2) As an algorithmic heuristic without provable guarantees but that tends to work extremely well in practice

Examples:

Maximum flow (exact algorithms)

Finding independent sets in restricted classes of graphs (exact algorithms)

Maximum satisfiability, graph partitioning (provable guarantees)

Travelling salesman problem, VLSI design (heuristic)

Programming computer to play chess (heuristic)

As a preprocessing step for exact algorithms, etc.

Plan

(1) Exact Max-2-SAT (was briefly covered last lecture)

(2) Programming Computer to Play Chess

(3) Travelling Salesman Problem Heuristics

(4) Coping with Local Optima

Example: Exact Max-2 SAT

INPUT: CNF formula F

n variables $x_1, x_2, \dots, x_n$

m clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_m$

each clause has exactly 2 variables

OUTPUT: assignment τ to the variables that satisfies as many clauses as possible

This problem is NP-hard to solve exactly

Solution space = {all possible assignments to variables}. It's size is 2^n

Exact Max-2-SAT

Example n = 3, m = 6

$(\neg x_1 \vee x_2)\ (\neg x_2 \vee x_3)\ (\neg x_3 \vee \neg x_1)\ (x_1 \vee x_3)\ (\neg x_3 \vee x_2)\ (\neg x_2 \vee x_1)$

Most natural local search algorithm:

Start with some assignment

Try flipping any bit in that assignment and see if it increases number of satisfied clauses

If so, flip the bit and repeat; otherwise terminate

Exact Max-2-SAT

Example n = 3, m = 6

$(\neg x_1 \vee x_2)\ (\neg x_2 \vee x_3)\ (\neg x_3 \vee \neg x_1)\ (x_1 \vee x_3)\ (\neg x_3 \vee x_2)\ (\neg x_2 \vee x_1)$

x_1	x_2	x_3	$\neg x_1 \vee x_2$	$\neg x_2 \vee x_3$	$\neg x_3 \vee \neg x_1$	$x_1 \vee x_3$	$\neg x_3 \vee x_2$	$\neg x_2 \vee x_1$	**Num sat clauses**
1	0	1	0	1	0	1	0	1	3

Exact Max-2-SAT

Example n = 3, m = 6

$(\neg x_1 \vee x_2)\ (\neg x_2 \vee x_3)\ (\neg x_3 \vee \neg x_1)\ (x_1 \vee x_3)\ (\neg x_3 \vee x_2)\ (\neg x_2 \vee x_1)$

x_1	x_2	x_3	$\neg x_1 \vee x_2$	$\neg x_2 \vee x_3$	$\neg x_3 \vee \neg x_1$	$x_1 \vee x_3$	$\neg x_3 \vee x_2$	$\neg x_2 \vee x_1$	Num sat clauses
1	0	1	0	1	0	1	0	1	3
1	1	1	1	1	0	1	1	1	5

Exact Max-2-SAT

Example n = 3, m = 6

$(\neg x_1 \vee x_2)\ (\neg x_2 \vee x_3)\ (\neg x_3 \vee \neg x_1)\ (x_1 \vee x_3)\ (\neg x_3 \vee x_2)\ (\neg x_2 \vee x_1)$

x_1	x_2	x_3	$\neg x_1 \vee x_2$	$\neg x_2 \vee x_3$	$\neg x_3 \vee \neg x_1$	$x_1 \vee x_3$	$\neg x_3 \vee x_2$	$\neg x_2 \vee x_1$	Num sat clauses
1	0	1	0	1	0	1	0	1	3
1	1	1	1	1	0	1	1	1	5
0	1	1	1	1	1	1	1	0	5

Exact Max-2-SAT

Example n = 3, m = 6

$(\neg x_1 \vee x_2)\ (\neg x_2 \vee x_3)\ (\neg x_3 \vee \neg x_1)\ (x_1 \vee x_3)\ (\neg x_3 \vee x_2)\ (\neg x_2 \vee x_1)$

x_1	x_2	x_3	$\neg x_1 \vee x_2$	$\neg x_2 \vee x_3$	$\neg x_3 \vee \neg x_1$	$x_1 \vee x_3$	$\neg x_3 \vee x_2$	$\neg x_2 \vee x_1$	Num sat clauses
1	0	1	0	1	0	1	0	1	3
1	1	1	1	1	0	1	1	1	5
0	1	1	1	1	1	1	1	0	5
1	1	0	1	0	1	1	1	1	5

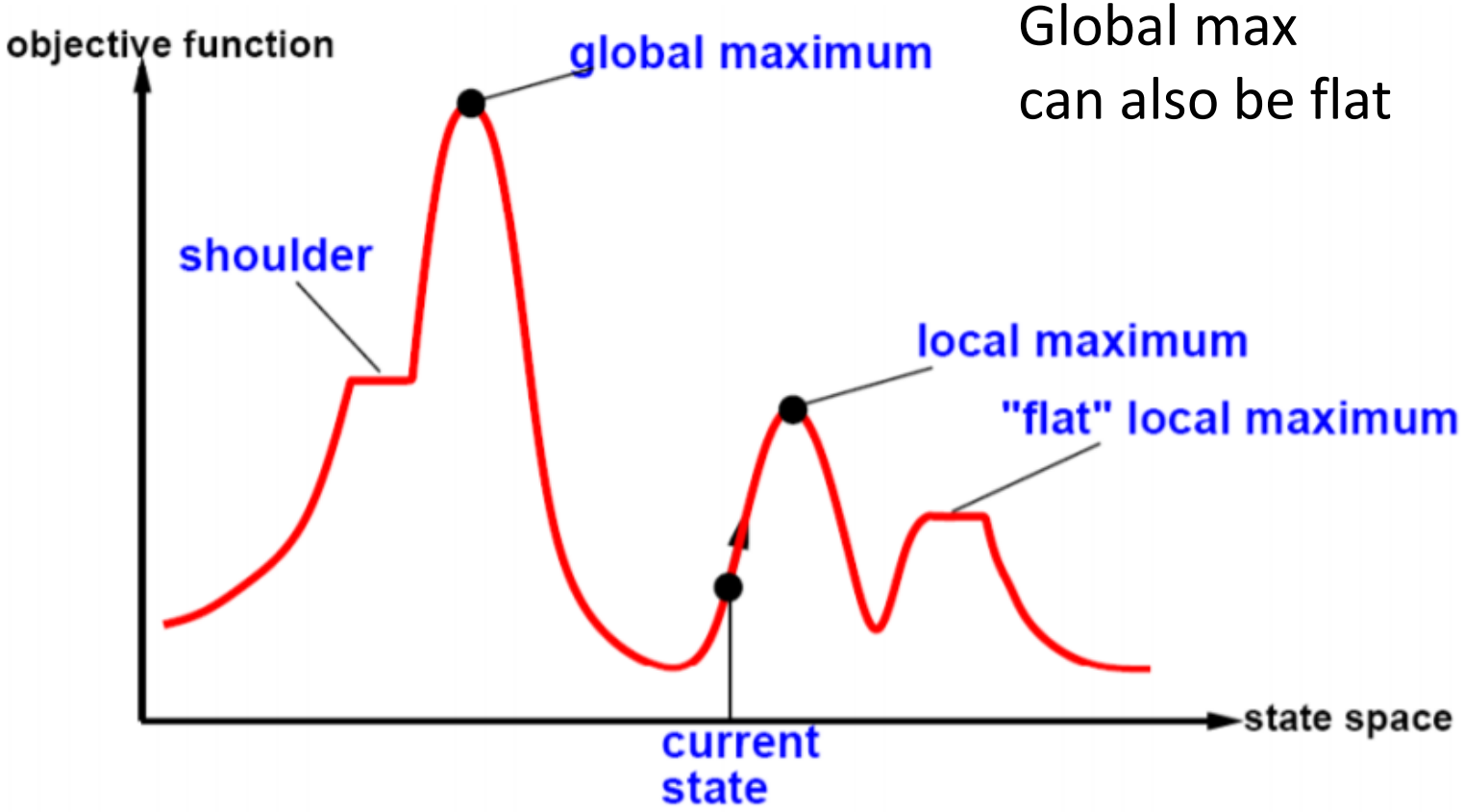
Exact Max-2-SAT

Example n = 3, m = 6

$(\neg x_1 \vee x_2)\ (\neg x_2 \vee x_3)\ (\neg x_3 \vee \neg x_1)\ (x_1 \vee x_3)\ (\neg x_3 \vee x_2)\ (\neg x_2 \vee x_1)$

x_1	x_2	x_3	$\neg x_1 \vee x_2$	$\neg x_2 \vee x_3$	$\neg x_3 \vee \neg x_1$	$x_1 \vee x_3$	$\neg x_3 \vee x_2$	$\neg x_2 \vee x_1$	Num sat clauses
1	0	1	0	1	0	1	0	1	3
1	1	1	1	1	0	1	1	1	5
0	1	1	1	1	1	1	1	0	5
1	1	0	1	0	1	1	1	1	5

Landscape of Solution Space



Global max
can also be flat

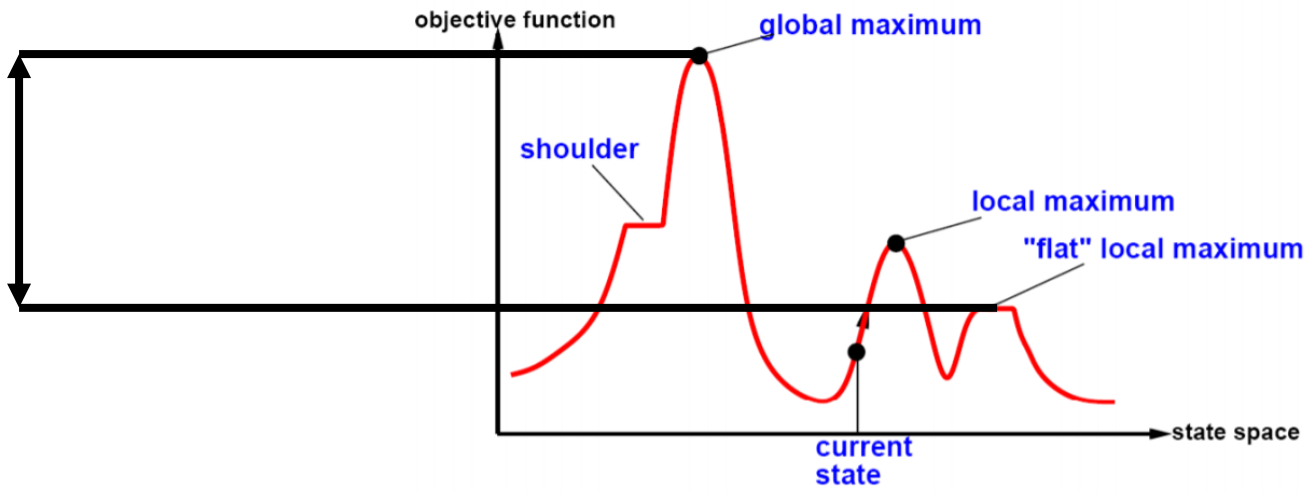
Approximation Guarantee

If local search algorithm terminates it ends up in a local optimum

How good is that optimum?

Worst-case approximation guarantee (locality gap) is the ratio between smallest local optimum and global optimum (for maximization problem)

Largest such ratio is the locality gap



Approximation Guarantee of Local Search for Exact MAX-2-SAT

We shall prove that our local search algorithms achieves 2/3 approximation ratio

Proof: Let τ be a local optimum

- S_0 - clauses not satisfied by τ
- S_1 - clauses satisfied by exactly one literal by τ
- S_2 - clauses satisfied by exactly two literals by τ

Eg. $(\neg x_1 \vee x_2)\ (\neg x_2 \vee x_3)\ (\neg x_3 \vee \neg x_1)\ (x_1 \vee x_3)\ (\neg x_3 \vee x_2)\ (\neg x_2 \vee x_1)$

Local opt τ: $x_1 = 1, x_2 = 1, x_3 = 1$

$S_0 = \{(\neg x_3 \vee \neg x_1)\}$

$S_1 = \{(\neg x_1 \vee x_2), (\neg x_2 \vee x_3), (\neg x_3 \vee x_2), (\neg x_2 \vee x_1)\}$

$S_2 = \{(x_1 \vee x_3)\}$

Approximation Guarantee of Local Search for Exact MAX-2-SAT

Let τ be a local optimum

- S_0 - clauses not satisfied by τ
- S_1 - clauses satisfied by exactly one literal by τ
- S_2 - clauses satisfied by exactly two literals by τ

Our algorithm finds an assignment satisfying $|S_1| + |S_2|$ clauses

Optimum is clearly less than the total number of clauses $|S_0| + |S_1| + |S_2|$

Thus, we need to show $\frac{|S_1|+|S_2|}{|S_0|+|S_1|+|S_2|} \geq \frac{2}{3}$. Alternatively, show $\frac{|S_0|}{|S_0|+|S_1|+|S_2|} \leq \frac{1}{3}$

Approximation Guarantee of Local Search for Exact MAX-2-SAT

Clause involves variable x_i if either x_i or $\neg x_i$ occurs in it

A_i - set of clauses in S_0 involving x_i

B_i - set of clauses in S_1 involving x_i and satisfied by x_i in τ

C_i - set of clauses in S_2 involving x_i

Local optimum means $|A_i| \leq |B_i|$ for all i

Also we have $\sum_i |A_i| = 2|S_0|$ and $\sum_i |B_i| = |S_1|$

Hence $2|S_0| \leq |S_1|$, thus

$$\frac{|S_0|}{|S_0|+|S_1|+|S_2|} \leq \frac{|S_0|}{|S_0|+2|S_0|+|S_2|} \leq \frac{|S_0|}{3|S_0|} \leq \frac{1}{3}$$

QED

Approximation Guarantee of Local Search for Exact MAX-2-SAT

Our simple local search finds a 2/3-approximation.

This analysis and algorithm can be generalized to the case where you associate arbitrary weights with clauses and the goal is to satisfy a set of clauses of maximum weight.

In our algorithm we said a modification of an assignment is improving if it increases the number of satisfied clauses

Better algorithm can be obtained with a different criterion for “improving” step (non-oblivious local search)

See Allan’s slides for more information!

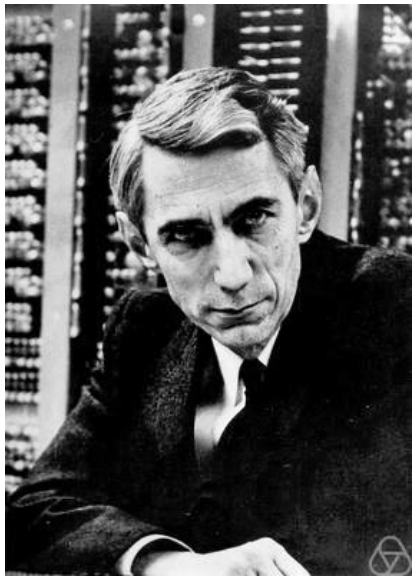
Other Interesting Applications

Claude Shannon (1916-2001)

One of the mathematical giants of the 20th century

Single-handedly introduced information theory

“A Mathematical Theory of Communication” (1948)



Less known paper

“Programming a computer for playing chess” (1950)

Programming a Computer to Play Chess

Shannon's proposal:

Encode a valid chess configuration by a 2-dimensional array

Define a "goodness" function

- maps each chess configuration to a number
- the higher the number the better configuration is for the player, e.g., if your king is checked in this configuration its not good. If you have a lot of pieces as opposed to your opponent, the configuration is good. Can make this precise in many ways

Computer program considers various valid moves and resulting configurations, and picks a better one according to the "goodness" function

Programming a Computer to Play Chess

Shannon's proposal is a local search algorithm!

State space: all possible valid chess configurations

Neighborhood: all chess configurations reachable from the current one in one step

Shannon also suggested extensions and improvements:

- Larger neighborhoods – chess configs resulting from several moves
- Certain moves are certainly bad and should not be explored further (pruning)
- Suggested that machine learning techniques should guide the choice of a "goodness" function and pruning strategies

Travelling Salesman Problem (TSP)

Complete graph

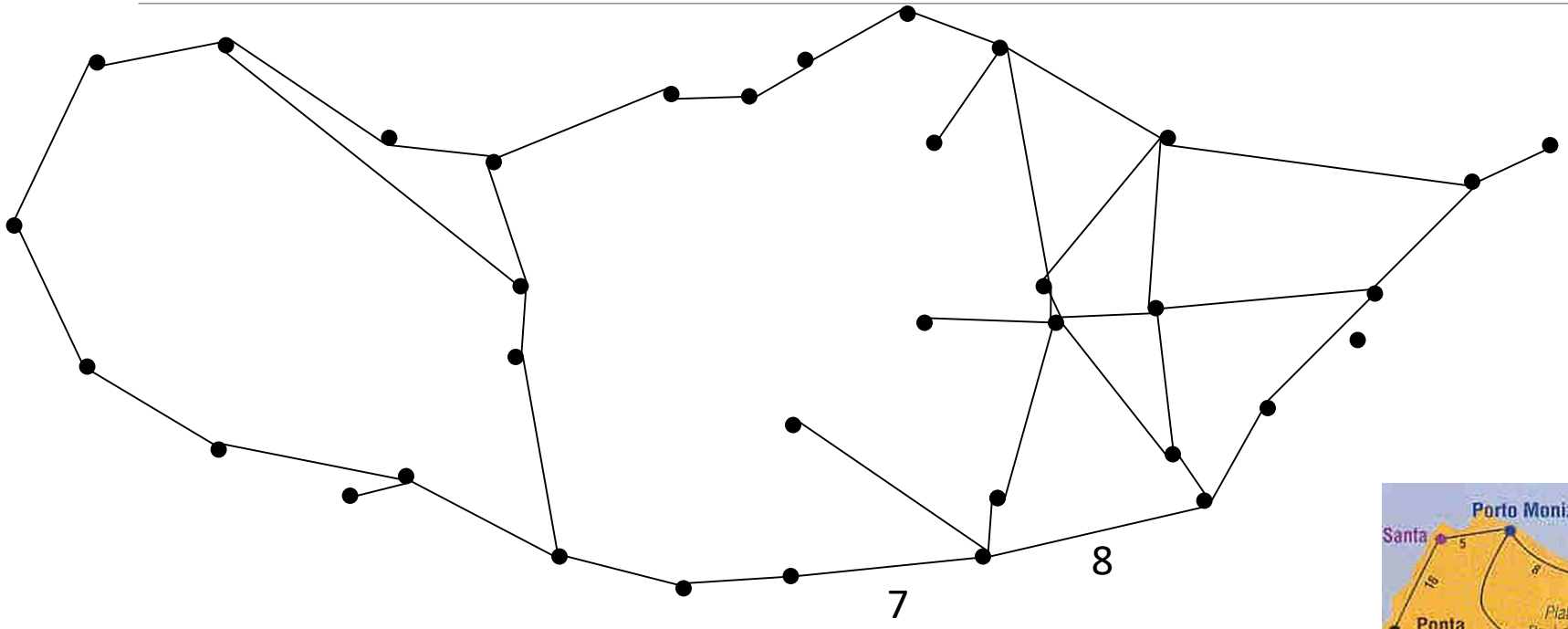
Positive weights on every edge

- Symmetric case: $w(i,j) = w(j,i)$
- Triangle inequality: $w(i,j) \leq w(i,k) + w(k,j)$
- Euclidean distance, i.e. nodes are located on a plane, weight of edge = dist

Find shortest tour (visit every node exactly once)

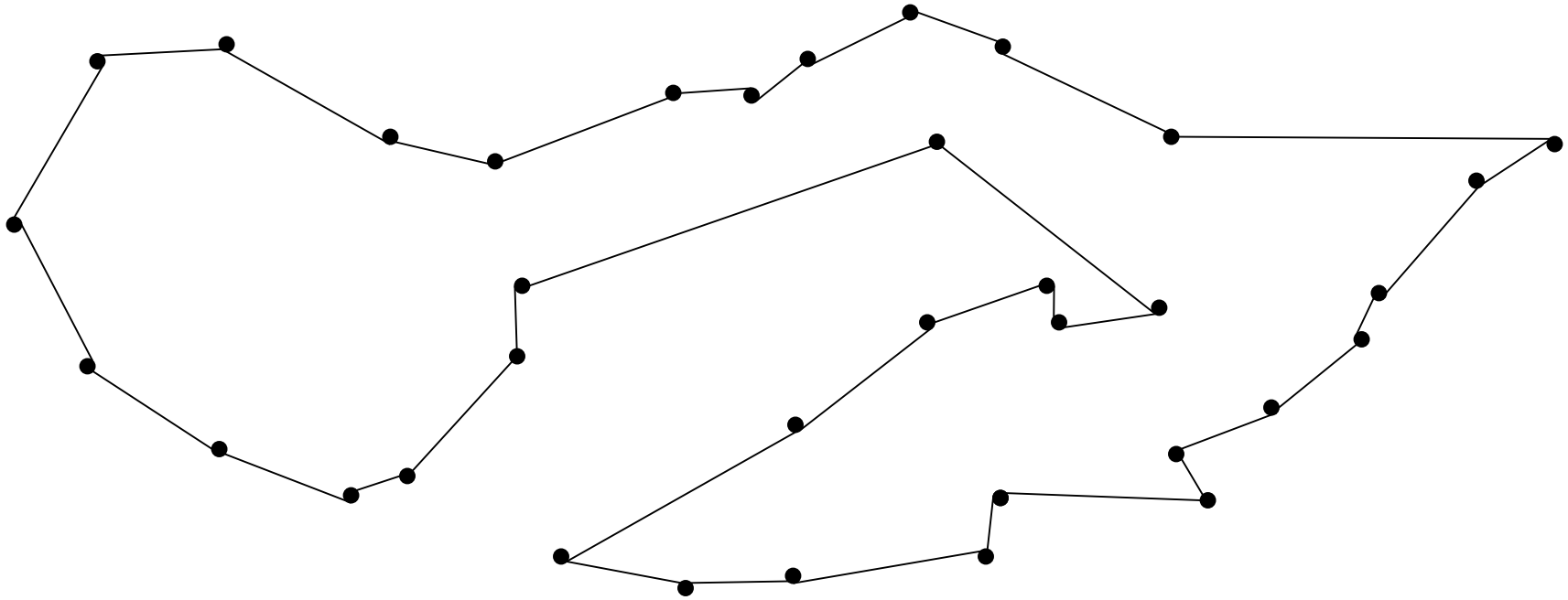
Madiera Island (west of Morocco)
Find a min distance tour visiting every city

Graph Representation



Not all edges are shown: there is an edge between every pair of vertices; weight of an edge = distance

Example of a Tour



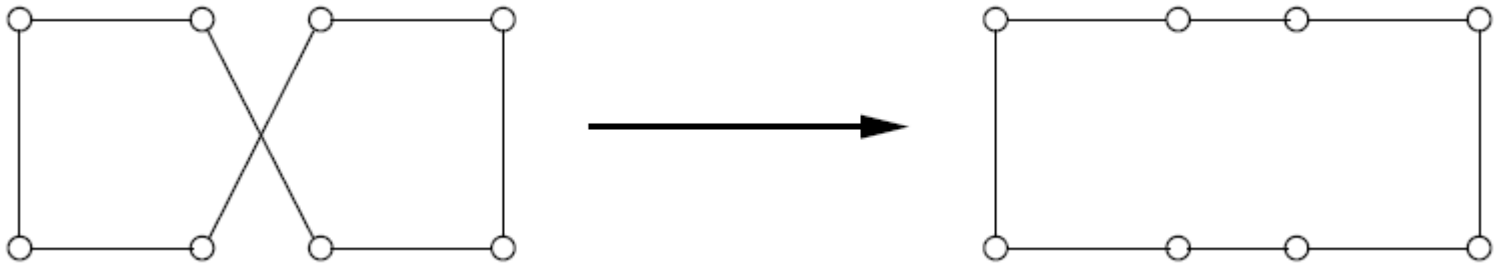
Local Search Heuristic

Solution space = {all possible tours, i.e., permutations, visiting n cities}
Solution space size is $(n-1)!$

Pick a solution, do local improving steps until get stuck in a local opt

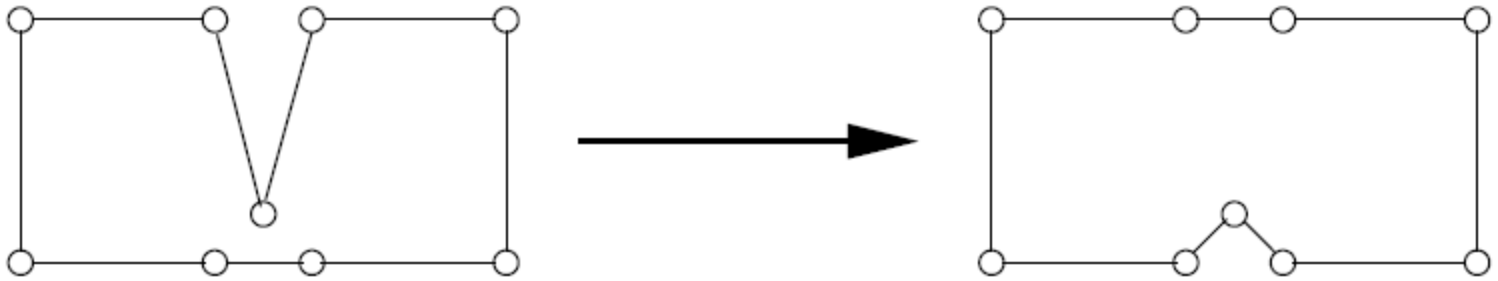
But what are local improving steps?

2-opt idea: take two edges out, put in two other edges in



Local Search for TSP

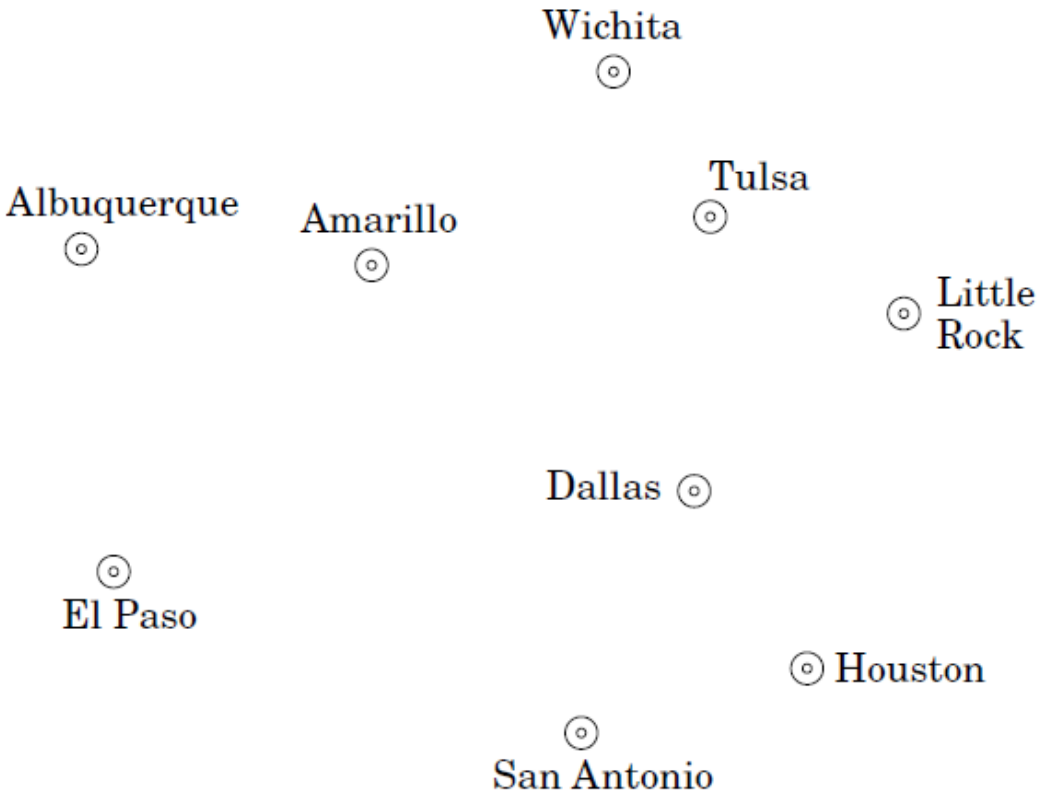
3-opt: take out 3 edges, put in 3 new edges



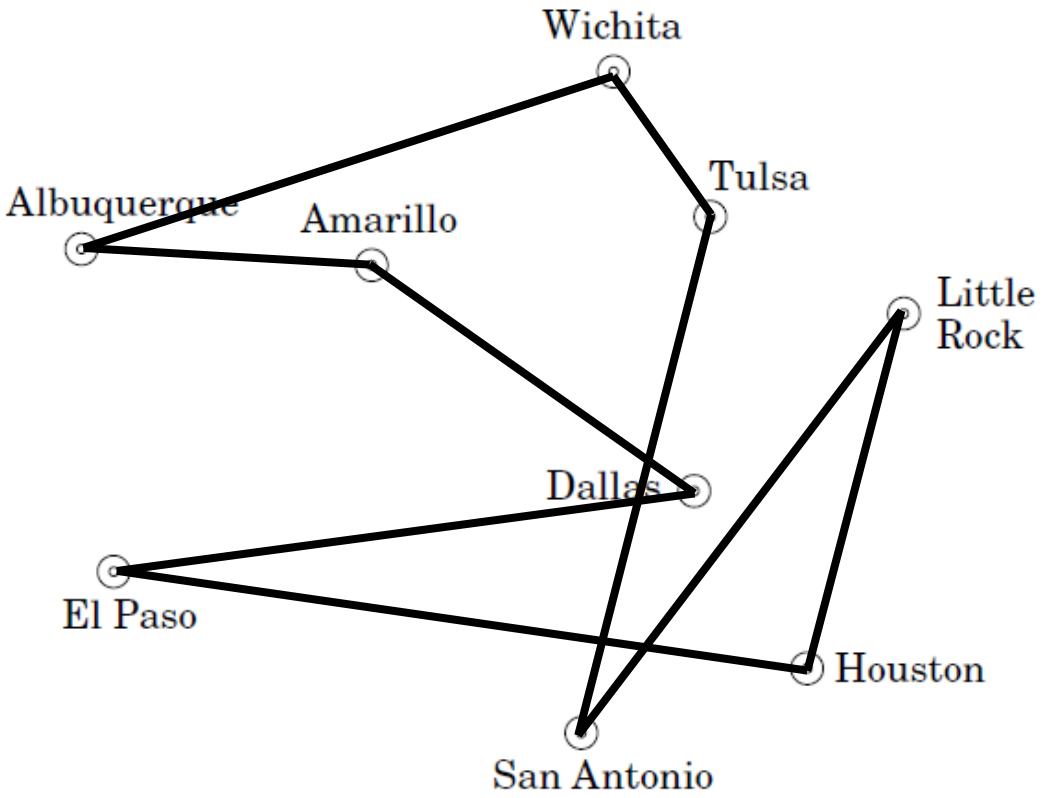
More generally, k-opt for any k is known as Lin-Kernighan heuristic

Bigger k bigger neighborhood!

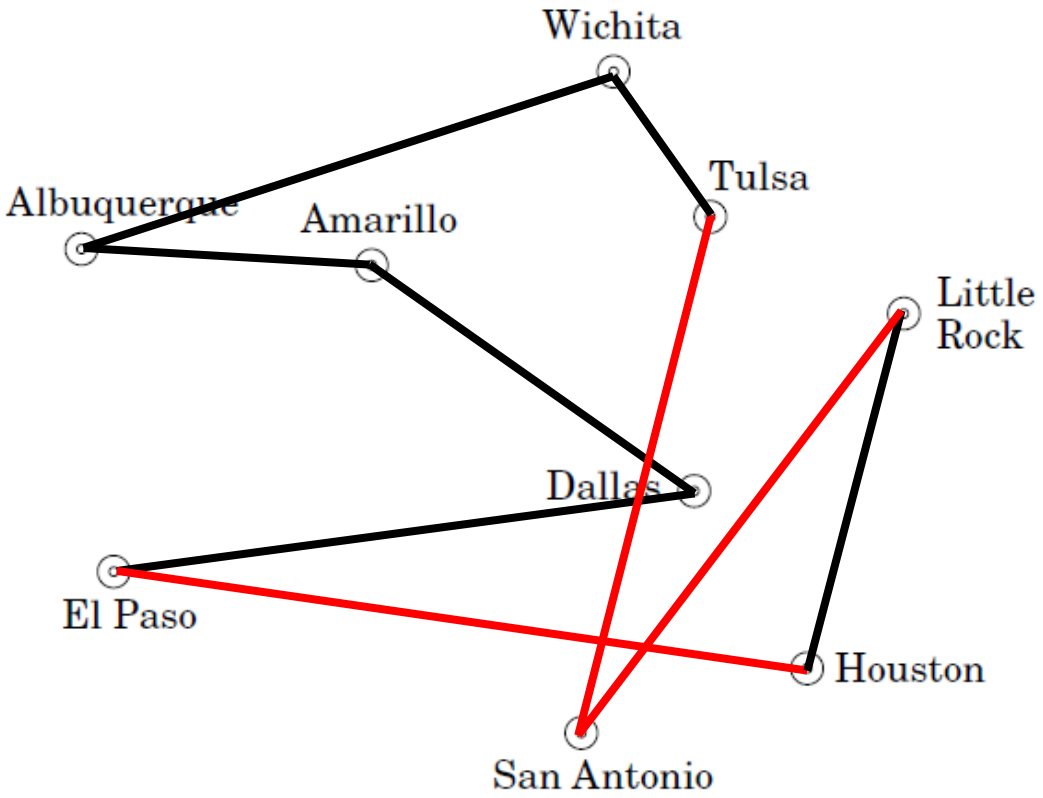
Example


Wichita
Albuquerque
Amarillo
Tulsa
Little Rock
Dallas
El Paso
Houston
San Antonio

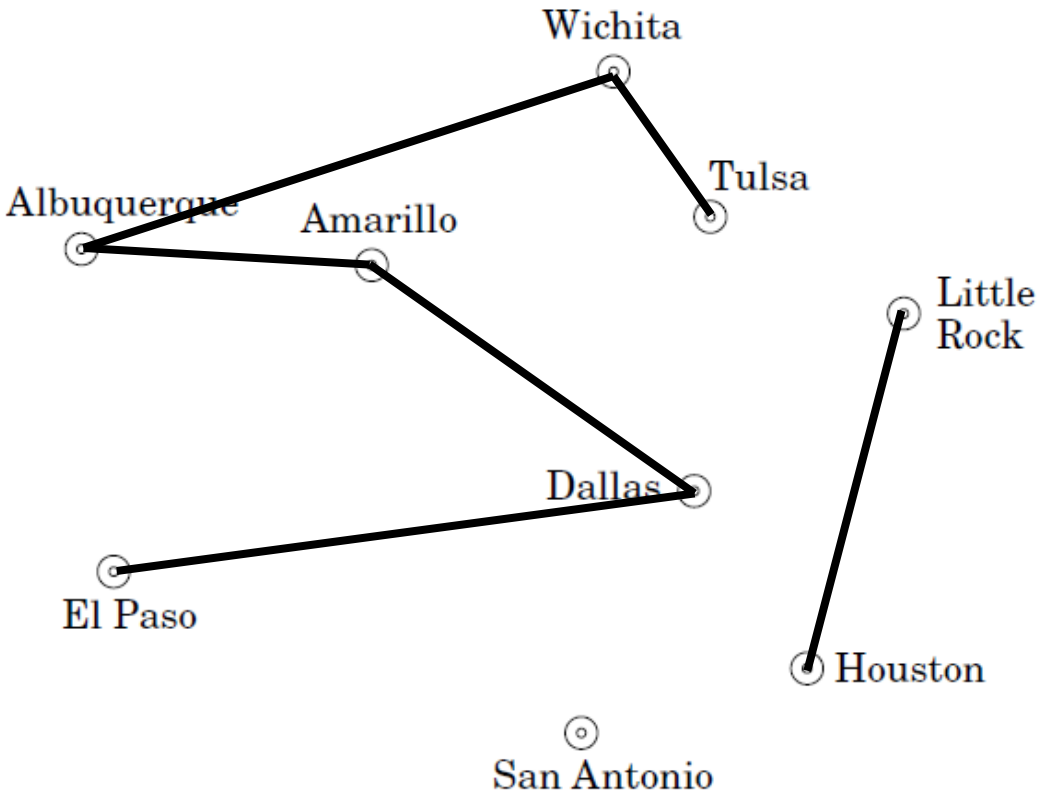
Example


Wichita
Albuquerque
Amarillo
Tulsa
Little
Rock
Dallas
El Paso
Houston
San Antonio

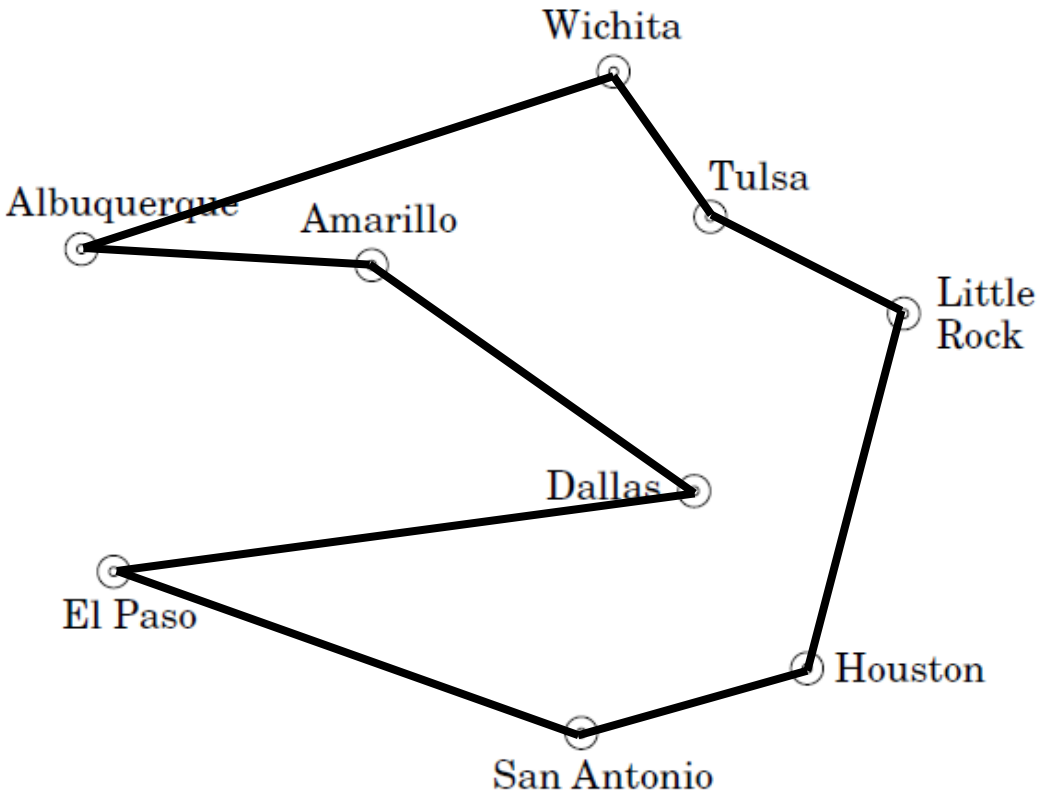
Example


Wichita
Tulsa
Albuquerque
Amarillo
Little
Rock
Dallas
El Paso
Houston
San Antonio

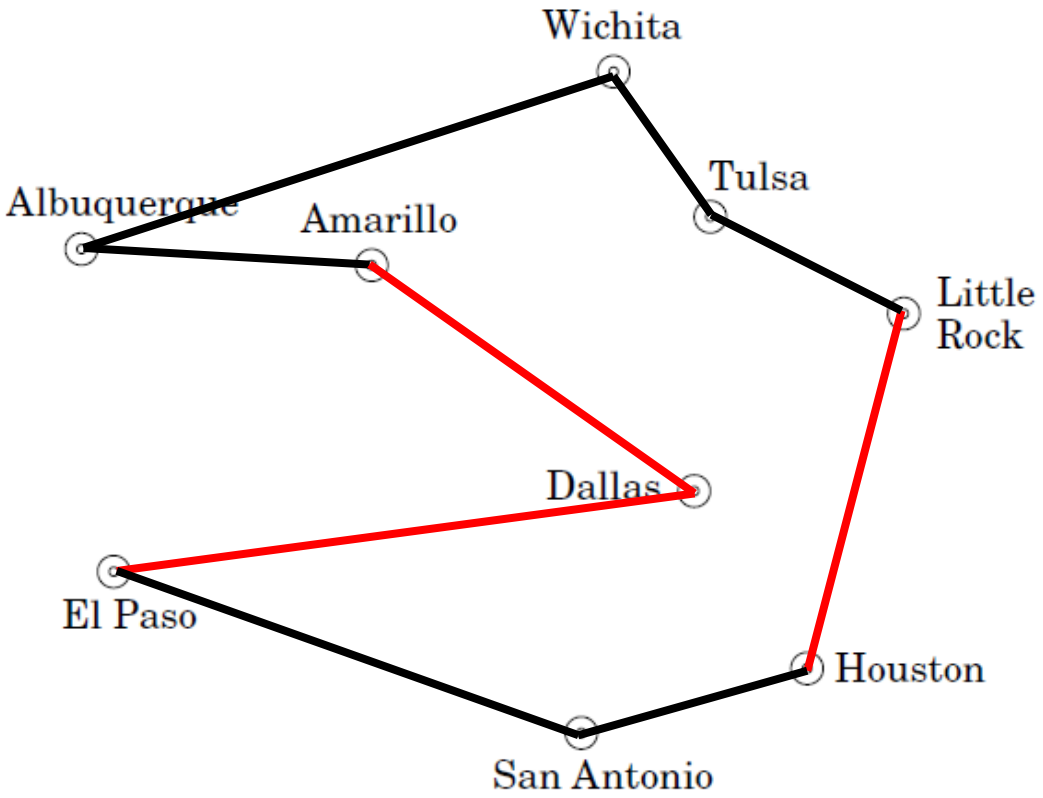
Example


Wichita
Tulsa
Albuquerque
Amarillo
Little
Rock
Dallas
El Paso
Houston
San Antonio

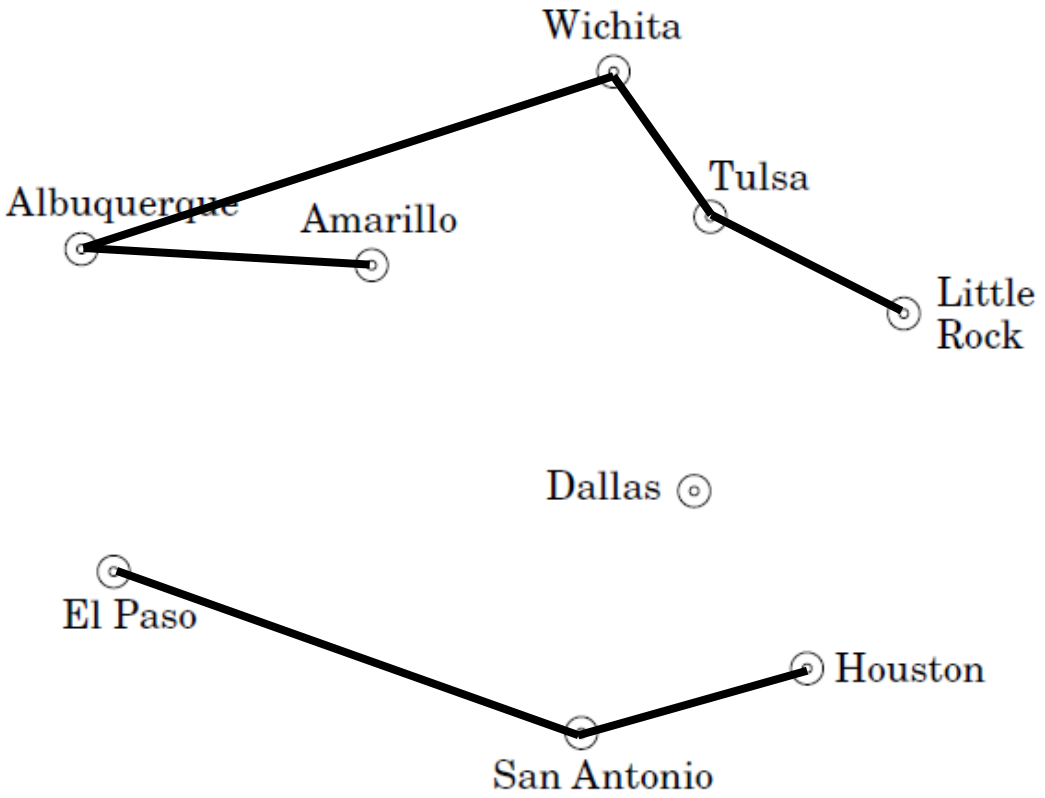
Example


Wichita
Albuquerque
Amarillo
Tulsa
Little
Rock
Dallas
El Paso
Houston
San Antonio

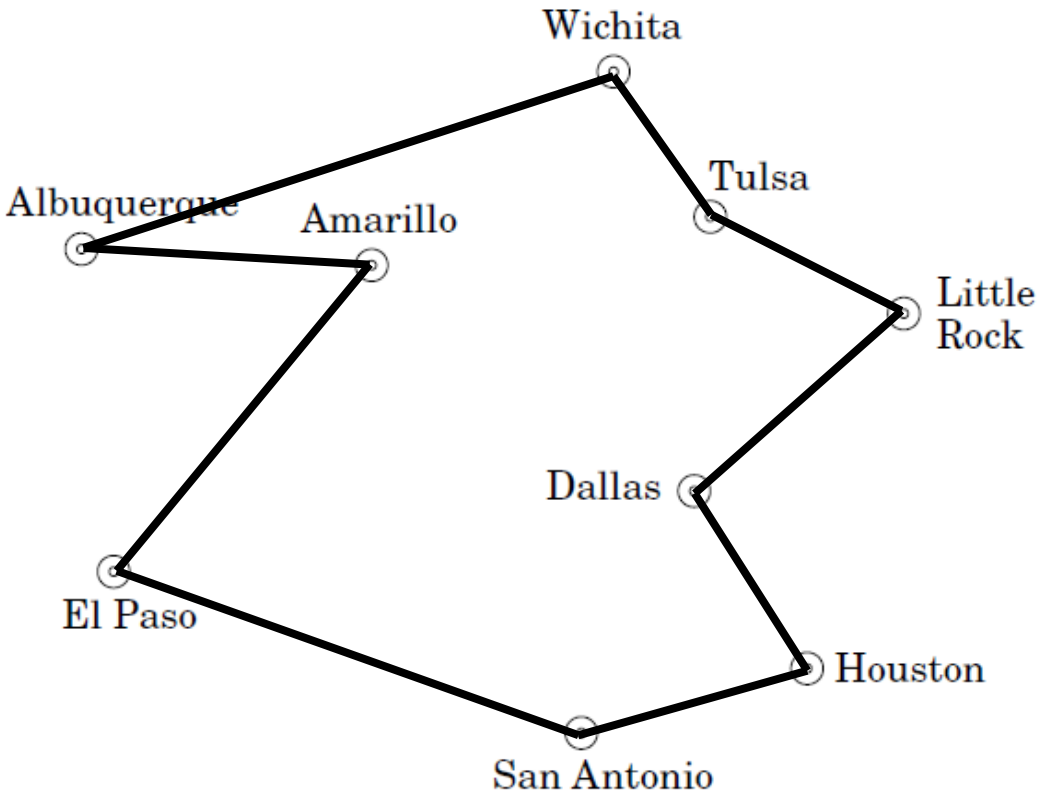
Example


Wichita
Albuquerque
Amarillo
Tulsa
Little
Rock
Dallas
El Paso
Houston
San Antonio

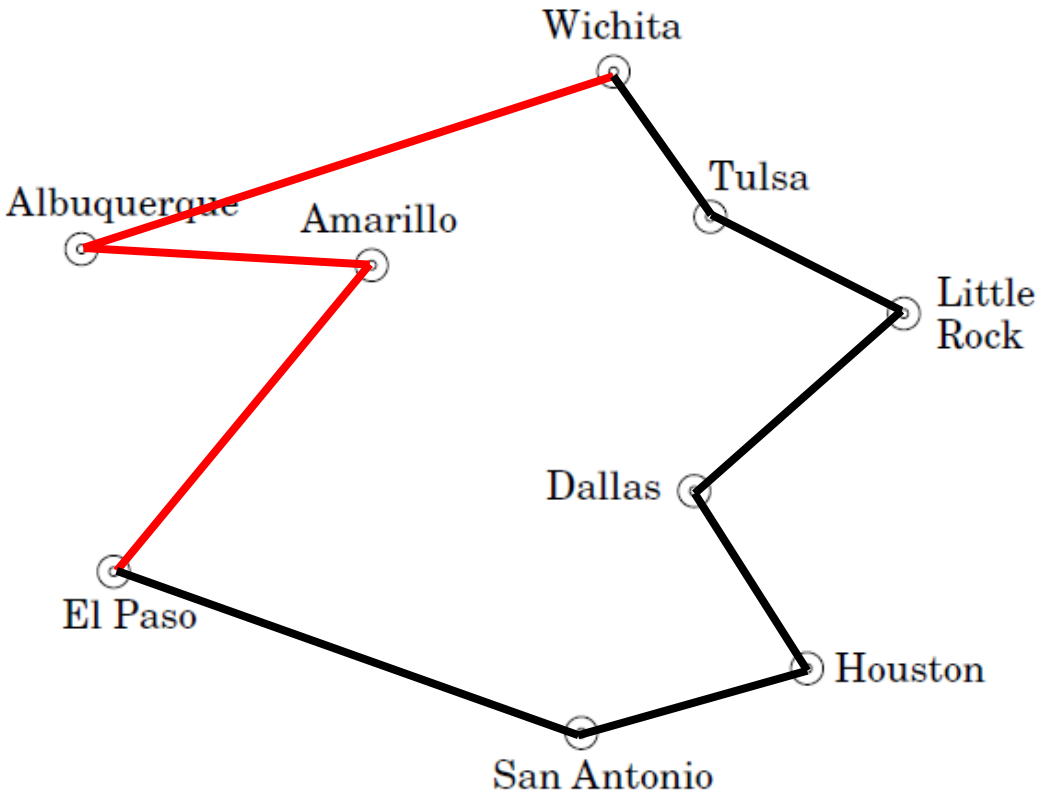
Example


Wichita
Albuquerque
Amarillo
Tulsa
Little
Rock
Dallas
El Paso
Houston
San Antonio

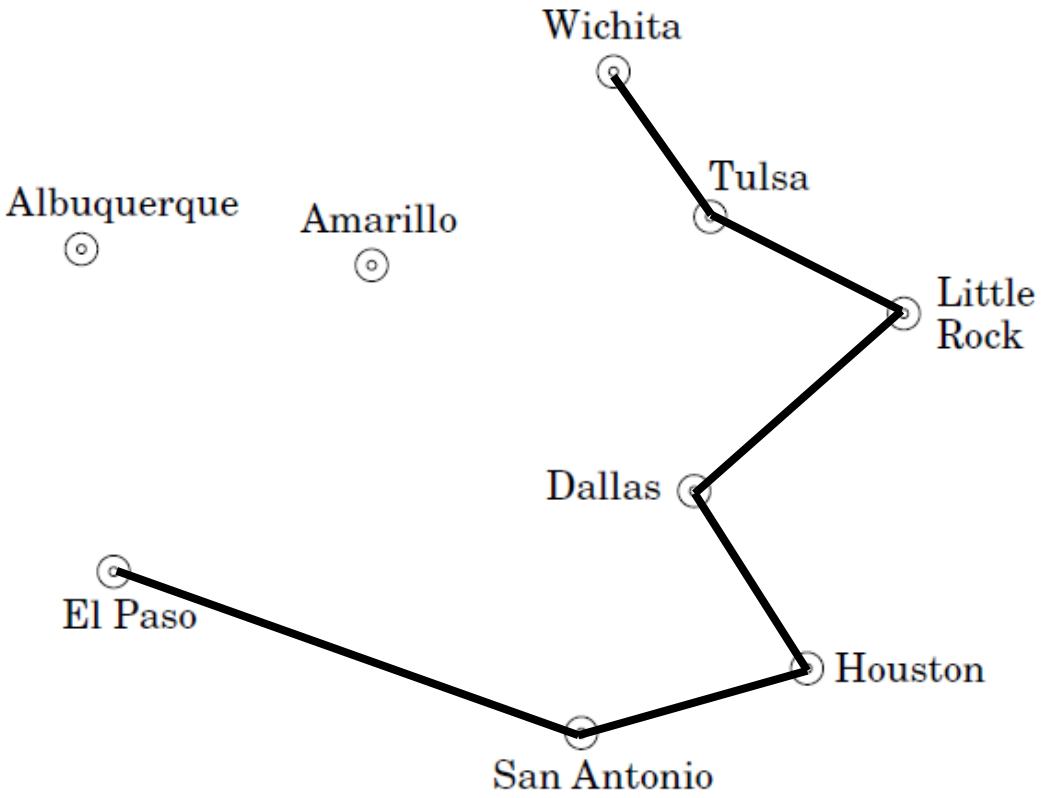
Example


Wichita
Albuquerque
Amarillo
Tulsa
Little
Rock
Dallas
El Paso
Houston
San Antonio

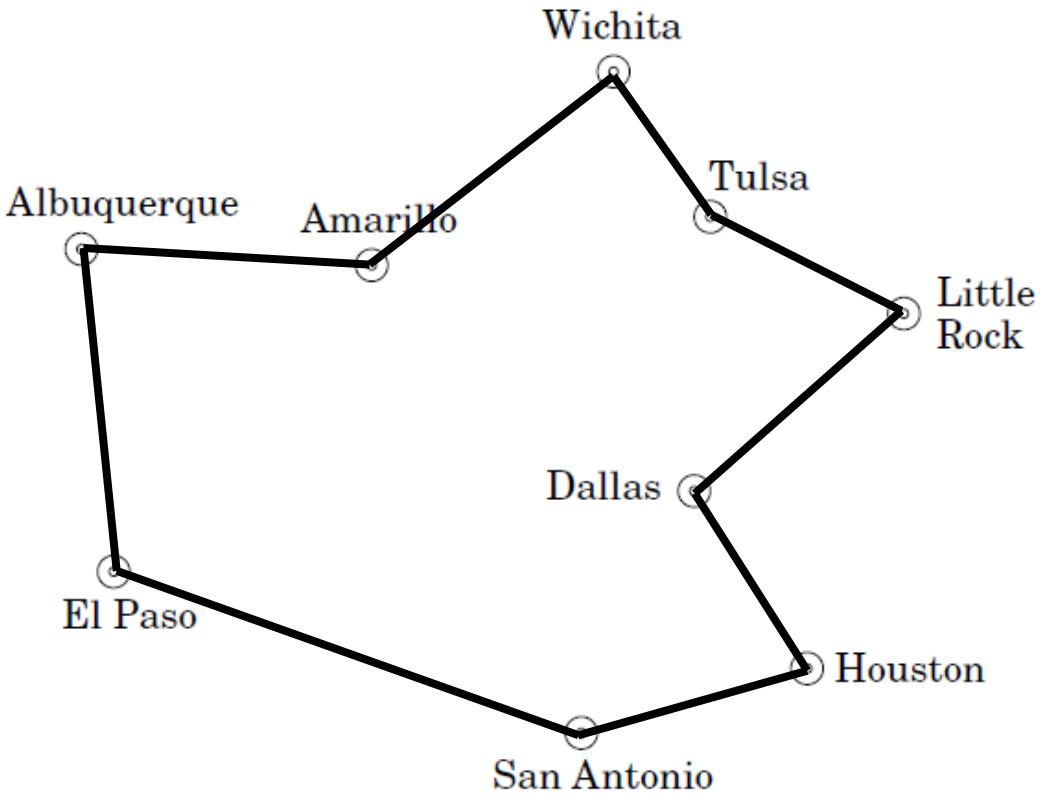
Example


Wichita
Albuquerque
Amarillo
Tulsa
Little
Rock
Dallas
El Paso
Houston
San Antonio

Example


Wichita
Tulsa
Albuquerque
Amarillo
Little
Rock
Dallas
El Paso
Houston
San Antonio

Example


Wichita
Albuquerque
Amarillo
Tulsa
Little
Rock
Dallas
El Paso
Houston
San Antonio

Local Search for TSP, Exact Methods, Some Benchmarks

1954 : 49 cities

1971 : 64 cities

1975 : 100 cities

1977 : 120 cities

1980 : 318 cities

1987 : 2,392 cities

1994 : 7,397 cities

1998 : 13,509 cities

2001 : 15,112 cities

Many of these results rely on Lin-Kernighan heuristic

Thus, nowadays people can solve TSP exactly with tens of thousands of vertices, but the problem is NP-hard!

This is a testament to how good these heuristics are in practice

Why Heuristic?

Well defined algorithms, so

- What is the overall running time?
- What is its approximation ratio?

In spite of large interest in local search methods for TSP, no one knows an answer to either of those questions.

Dealing with Local Optima

Algorithm gets stuck in local optima

To improve performance want methods to push algorithm out of local optima

Simplest methods:

Random restarts – when algorithm gets stuck in a local optimum, restart it at a random initial solution

Add **more randomization** – when algorithms picks which solution to move to (locally), pick any improving solution with some probability

Logic for more randomization is that maybe your algorithm gets stuck in a bad local optimum due to **systematic** choices it makes during the runtime

Dealing with Local Optima

Taking idea of randomization further:

With some small probability, allow your algorithm to make a local step that is **worse** than the current solution

Vary this probability with time

- Initially this probability is rather large to allow your algorithm to explore solution space widely
- Later this probability of taking a non-improving step decreases more and more allowing the algorithm to narrow down on a promising region in the solution space

Dealing with Local Optima

Previous slide describes **simulated annealing** technique

```
let s be any starting solution
repeat
    randomly choose a solution s' in the neighborhood of s
    if Δ = cost(s') − cost(s) is negative:
        replace s by s'
    else:
        replace s by s' with probability e^(−Δ/T).
```

It is usually very successful in practice outperforming simple local search, but it is incredibly difficult to analyze

NEW TOPIC:
DUAL FITTING

What is Dual Fitting?

Technique for analyzing approximation ratio of an algorithm

In many cases, the algorithm is greedy

The algorithm is interpreted as building a primal solution

In the analysis, you build a dual solution of similar value

Set Cover Problem

We are given a universe of n elements $\{1,2, \dots, n\}$ and m subsets of the universe $S_1, S_2, \dots, S_m$

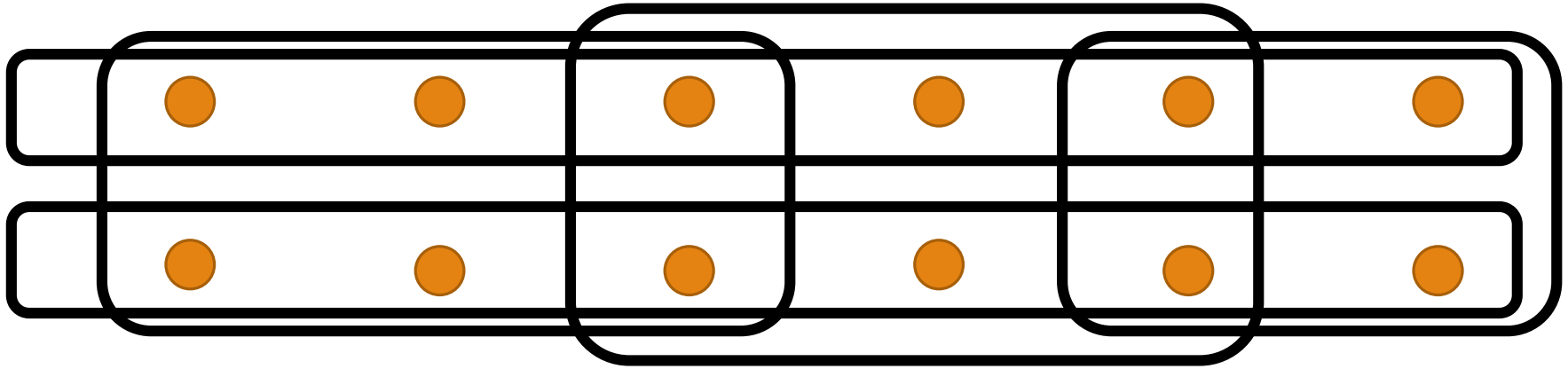
A **set cover** is a collection of these sets that covers each element at least ones

The goal is to find a cover of minimum size. This is NP-hard.

Greedy Algorithm

Pick a subset that covers the most number of uncovered elements.

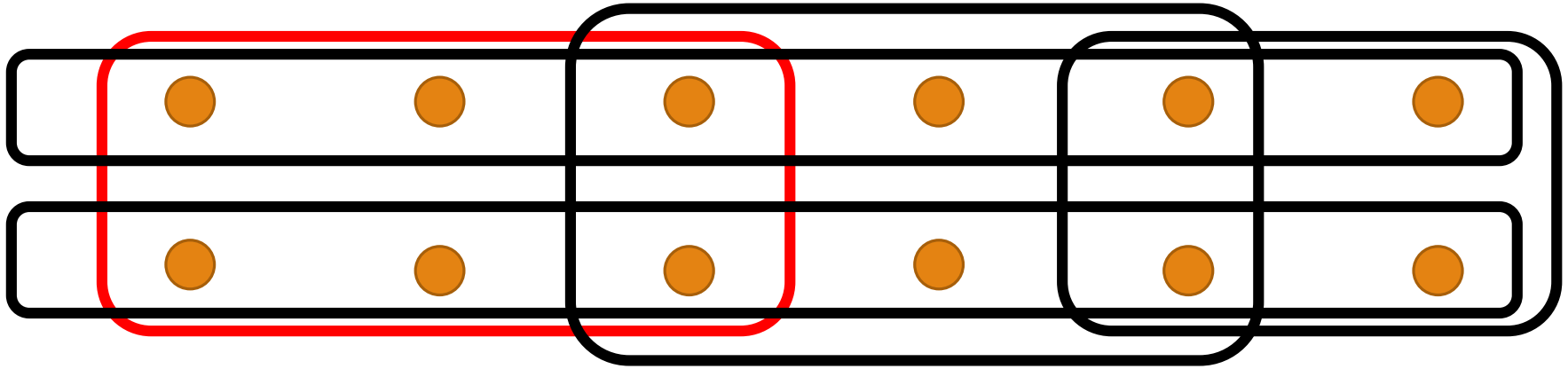
Repeat until all elements are covered.



Greedy Algorithm

Pick a subset that covers the most number of uncovered elements.

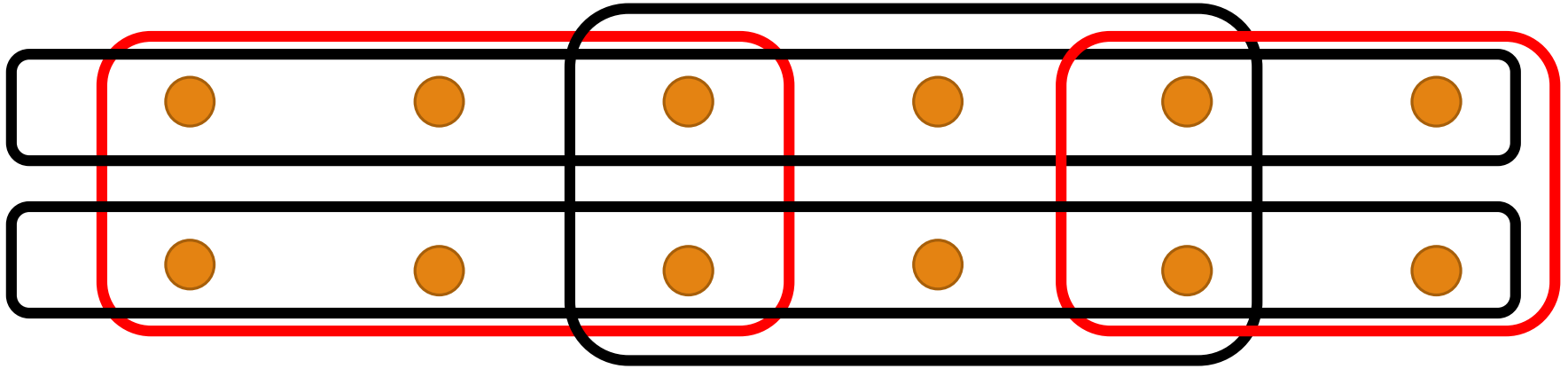
Repeat until all elements are covered.



Greedy Algorithm

Pick a subset that covers the most number of uncovered elements.

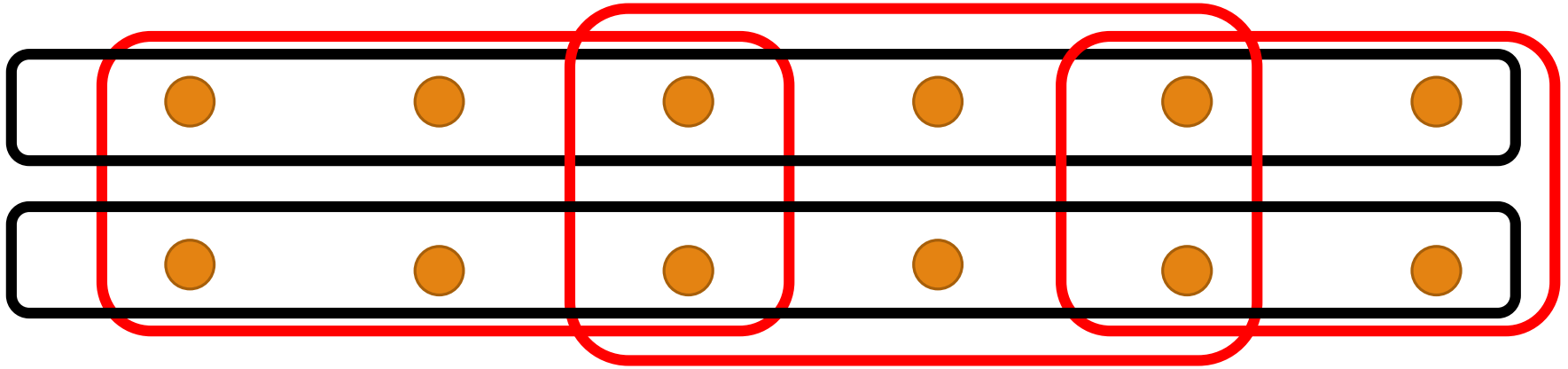
Repeat until all elements are covered.



Greedy Algorithm

Pick a subset that covers the most number of uncovered elements.

Repeat until all elements are covered.



Greedy Algorithm

Pick a subset that covers the most number of uncovered elements.

Repeat until all elements are covered.

Our goal: prove that this algorithm is within $\boldsymbol{ln\, n + O(1)}$ of the optimal value.

An Integer Program for Set Cover

x_S is an indicator variable: 1 if S is picked in the set cover and 0 otherwise

$$\min \sum_S x_S$$

$$\forall e : \sum_{S:e \in S} x_S \geq 1$$

$$\forall S : x_S \in \{0,1\}$$

Every element e should be included in at least one set.

LP Relaxation of Set Cover

Replace the integrality constraint $x_S \in \{0,1\}$ by a linear constraint $0 \leq x_S \leq 1$

$$\min \sum_S x_S$$

$$\forall e : \sum_{S:e \in S} x_S \geq 1$$

$$\forall S : x_S \in \{0,1\}$$

$$\min \sum_S x_S$$

$$\forall e : \sum_{S:e \in S} x_S \geq 1$$

$$\forall S : x_S \geq 0$$

Dual LP

PRIMAL

$$\min \sum_{S} x_S$$

$$\forall e : \sum_{S:e \in S} x_S \geq 1 \qquad y_e$$

$$\forall S : x_S \geq 0$$

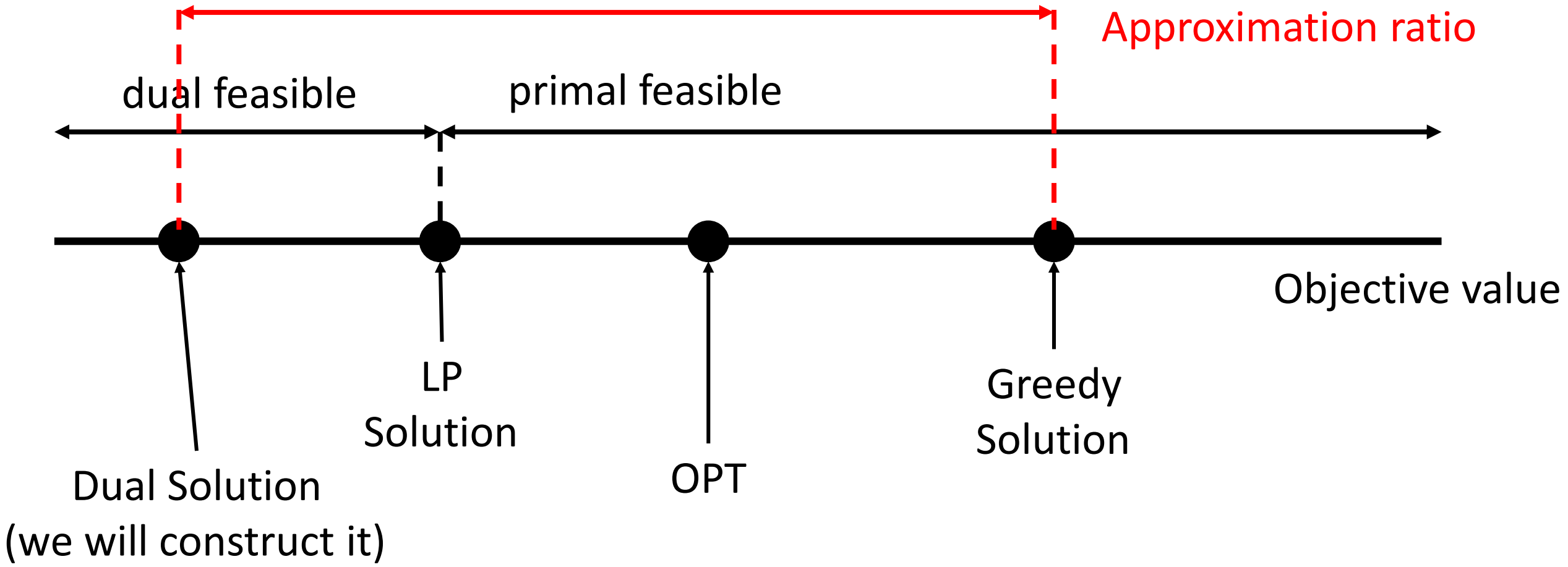
DUAL

$$\max \sum_{e \in U} y_e$$

$$\forall S : \sum_{e \in S} y_e \leq 1$$

$$\forall e : y_e \geq 0$$

Analysis Goal: Sandwich OPT between Algorithm's Soln and a Feasible Dual Soln



The Algorithm vs the Prover

PRIMAL

$$\min \sum_{S} x_S$$

$$\forall e : \sum_{S:e \in S} x_S \geq 1$$

$$\forall S : x_S \geq 0$$

DUAL

$$\max \sum_{e \in U} y_e$$

$$\forall S : \sum_{e \in S} y_e \leq 1$$

$$\forall e : y_e \geq 0$$

Consider two entities: the algorithm and the prover

The algorithm can be viewed as constructing a feasible solution to PRIMAL

The prover will be constructing a feasible solution to the DUAL

If the prover succeeds at constructing a feasible solution within α fraction of the primal solution, it means that the approximation ratio is α

Prover is completely imaginary – we play the role of the prover

The Algorithm vs the Prover

PRIMAL	DUAL
$\min \sum_{S} x_S$	$\max \sum_{e \in U} y_e$
$\forall e : \sum_{S: e \in S} x_S \geq 1$	$\forall S : \sum_{e \in S} y_e \leq 1$
$\forall S : x_S \geq 0$	$\forall e : y_e \geq 0$

Consider a particular step in the algorithm
- Algorithm selects a new set S
- This set covers k new elements

The algorithm can be viewed as constructing a feasible solution to PRIMAL: $\boldsymbol{x_S = 1}$
Note: this adds value 1 to the primal objective

As a prover, we need to create some dual soln, i.e., assign y_e values to all newly covered elements e.
First try: assign $\boldsymbol{y_e = 1/k}$
The good: we added the total of $k(1/k) = 1$ value to the dual objective
The bad: our dual vars violate dual constraints (set S can have prev. covered elts)

Making Prover's Dual Solution Feasible

We set $\boldsymbol{y_e = 1/k}$ where k is the number of new elements covered at the time element e was covered.

Consider an arbitrary set S. We would like to enforce $\sum_{e \in S} y_e \leq 1$.

Assume without loss of generality that if greedy covers e_j before e_k then $j < k$.

When greedy picked a set to cover e_j, it could also have picked S and covered $|S| - j + 1$ new elements.

So $y_e \leq 1/(|S| - j + 1)$

$\sum_{e \in S} y_e \leq (1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{|S|}) \leq \ln|S| + O(1)$

Making Prover's Dual Solution Feasible

We set $\boldsymbol{y_e = 1/k}$ where k is the number of new elements covered at the time element e was covered.

When variables are set as above, we have

$$\sum_{e \in S} y_e \leq \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{|S|}\right) \leq \ln|S| + O(1) \leq \ln n + O(1)$$

But we want $\sum_{e \in S} y_e \leq 1$

So…? Rescale the variables! Set $\boldsymbol{\hat{y}_e = \frac{y_e}{\ln n + O(1)}}$

Then the variables $\boldsymbol{\hat{y}_e}$ form a feasible dual solution.

Making Prover's Dual Solution Feasible

We set $\boldsymbol{y_e = 1/k}$ where k is the number of new elements covered at the time element e was covered.

Setting $\boldsymbol{\hat{y}_e = \frac{y_e}{\ln n + O(1)}}$ made the prover's solution feasible.

What happened to the objective value?

Recall that adding $\boldsymbol{y_e}$ for all e gave exactly the same value as algo's primal soln

Hence, adding $\boldsymbol{\hat{y}_e}$ for all e gives a solution of value within $\boldsymbol{\ln n + O(1)}$ of primal's.

This finishes the proof, since opt is sandwiched between feasible dual and greedy solution.

NEW TOPIC:
RANDOMIZED ALGORITHMS

Deterministic Algorithms

Goals:

Correctness – algo always solves the problem correctly

Efficiency – algo always solves the problem quickly, e.g., polytime

Randomized Algorithms

Behavior of the algorithm can vary even for a given fixed input!

Goals:

Correctness – algo often solves the problem correctly (for every input!)

Efficiency – algo often solves the problem quickly, e.g., expected polytime

Probabilistic Analysis of Algorithms

Goals:

Correctness – algo solves the problem correctly on **most** inputs

Efficiency – algo solves the problem quickly, e.g., polytime, on **most** inputs

Reasons for Studying Randomized Algorithms

Simplicity

Performance

For many problems, a randomized algorithm is either the simplest known, the fastest known, or both

Turning randomized algorithms into deterministic is called “derandomization”

A randomized algorithm is often discovered first, then it is derandomized, but in the process either runtime or approximation ratio might suffer

In certain cases, people don’t know if an algorithm can be derandomized

Lastly, in some problem settings (e.g., sublinear time algorithms) randomization is provably necessary!

Applications of Randomized Algorithms

Number theory (primality testing)

Data structures (hashing)

Algebraic identities (polynomial and matrix identity verification)

Mathematical programming (rounding linear program relaxations)

Counting and enumeration (approximating counting of perfect matchings)

Networking (deadlock avoidance)

Probabilistic method for proving theorems

etc…

This Lecture: Randomized Algos

Plan:

(1) Probability overview

(2) Types of randomized algorithms

(3) Polynomial identity testing

Probability Theory Overview

Mathematical theory of chance

We will only consider finite discrete probability spaces

Probability space is a pair (Ω, p), where

- Ω – **sample space**, $|\Omega| < \infty$
- $p: \Omega \to [0,1]$ – **probability distribution**, i.e., a function such that

$$\sum_{\omega \in \Omega} p(\omega) = 1$$

Probability Theory Overview

Event is a subset of the sample space, $A \subseteq \Omega$.

How many distinct events are there? Answer: $2^{|\Omega|}$ (prove it!)

Probability distribution extends to events:

$$\text{Let } A \subseteq \Omega \text{ then } p(A) = \sum_{\omega \in \Omega} p(\omega)$$

$\omega \in \Omega$ is called an **elementary event**

Probability Theory Overview

Example: throw a fair six-sided dice once

Sample space $\Omega = \{1,2,3,4,5,6\}$

Probability distribution $p(1) = p(2) = p(3) = p(4) = p(5) = p(6) = \frac{1}{6}$

The above distribution is called **uniform**

More generally, uniform distribution is such that $p(\omega) = 1/|\Omega|$

Event A: "even number shows up on the top of the dice"

Formally, $A = \{2,4,6\}$, so $p(A) = p(2) + p(4) + p(6) = \frac{3}{6} = \frac{1}{2}$

Probability Theory Overview

Back to general definitions

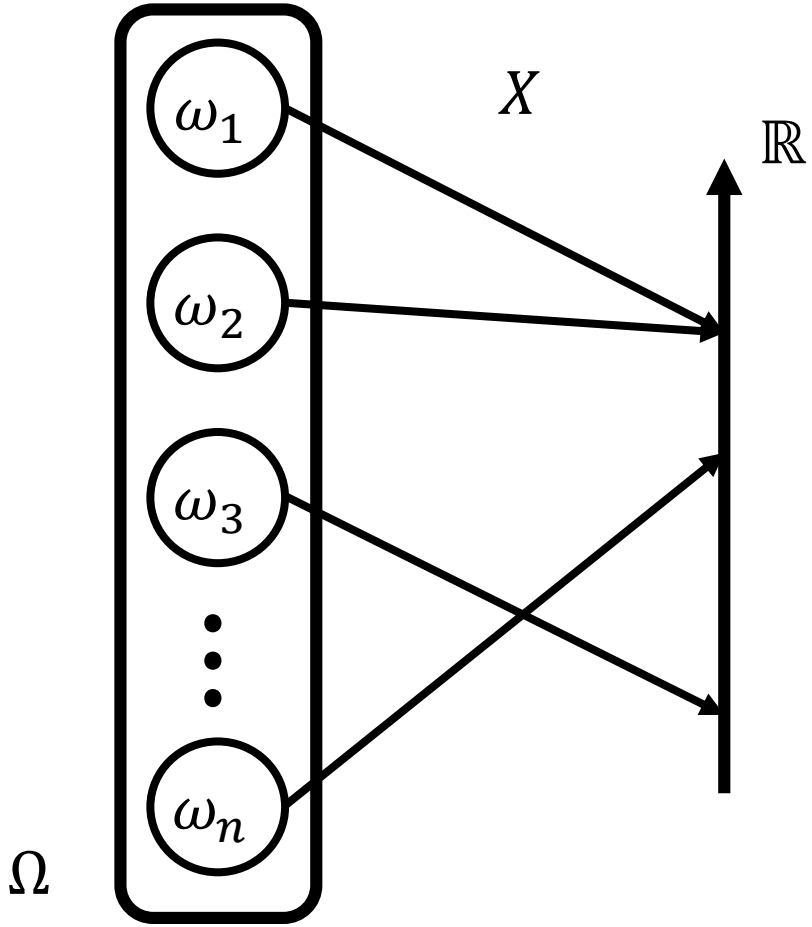
Fix probability space (Ω, p)

Real-valued random variable is a function $X: \Omega \to \mathbb{R}$

It is a misnomer: random variable is

- Neither random
- Nor a variable

Probability Theory Overview



Real-valued random variable "collapses" elementary events into groups and associates real numbers with those groups

These groups are subsets of sample space, i.e., events, which we denote by "$X = value$"

Probability Theory Overview

For example, consider a throw of a fair six-sided dice

Let X be the random variable defined as follows:

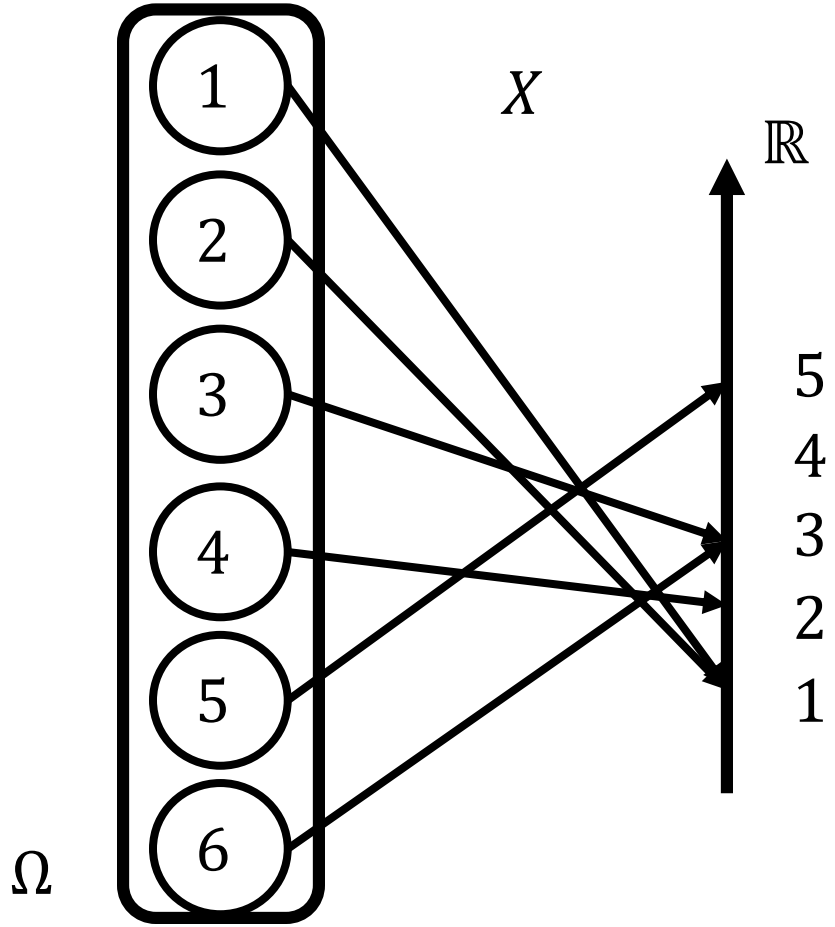
If the number that comes up is odd, X is equal to that number

If the number that comes up is even, X is equal to half of that number

In other words,

$$X(1) = 1$$
$$X(2) = 2/2 = 1$$
$$X(3) = 3$$
$$X(4) = 4/2 = 2$$
$$X(5) = 5$$
$$X(6) = 6/2 = 3$$

Probability Theory Overview



Events corresponding to outcomes of random variable

$$P(X = 1) = P(\{1,2\}) = p(1) + p(2) = \frac{2}{6} = \frac{1}{3}$$

$$P(X = 2) = P(\{4\}) = p(4) = \frac{1}{6}$$

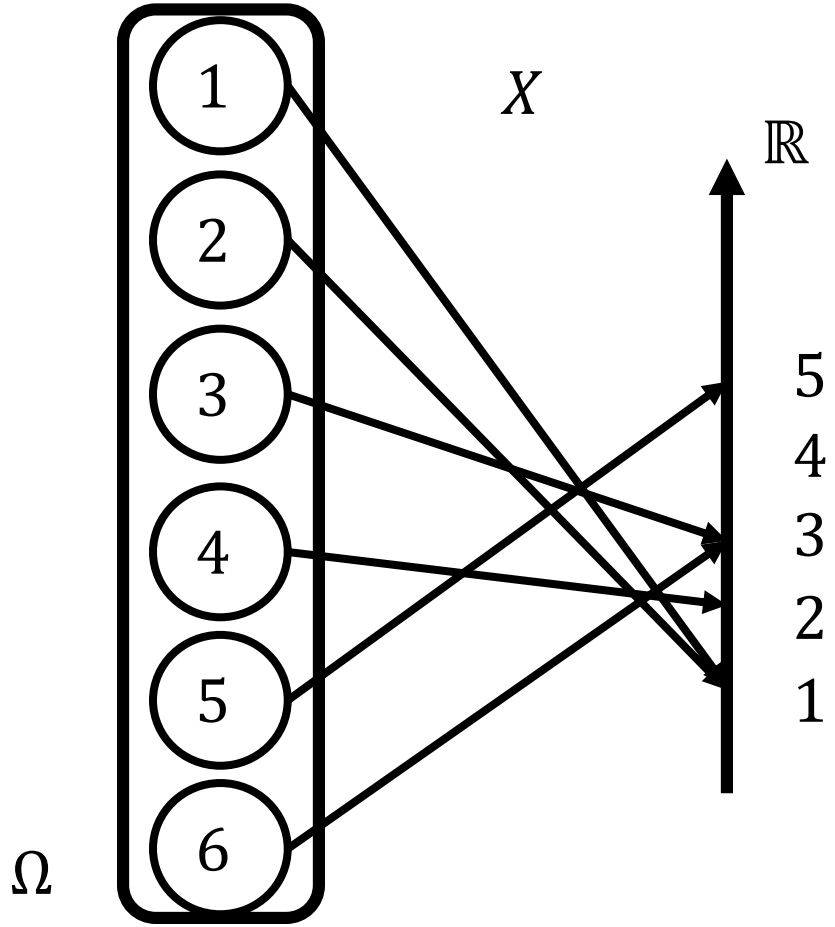
$$P(X = 3) = P(\{3,6\}) = p(3) + p(6) = \frac{2}{6} = \frac{1}{3}$$

$$P(X = 5) = P(\{5\}) = p(5) = \frac{1}{6}$$

NOTE:

$$P(X = 1) + P(X = 2) + P(X = 3) + P(X = 5) = 1$$

Probability Theory Overview



Thus, we get an auxiliary probability space!

Sample space

outcomes of random variable X, e.g., $\{1,2,3,5\}$

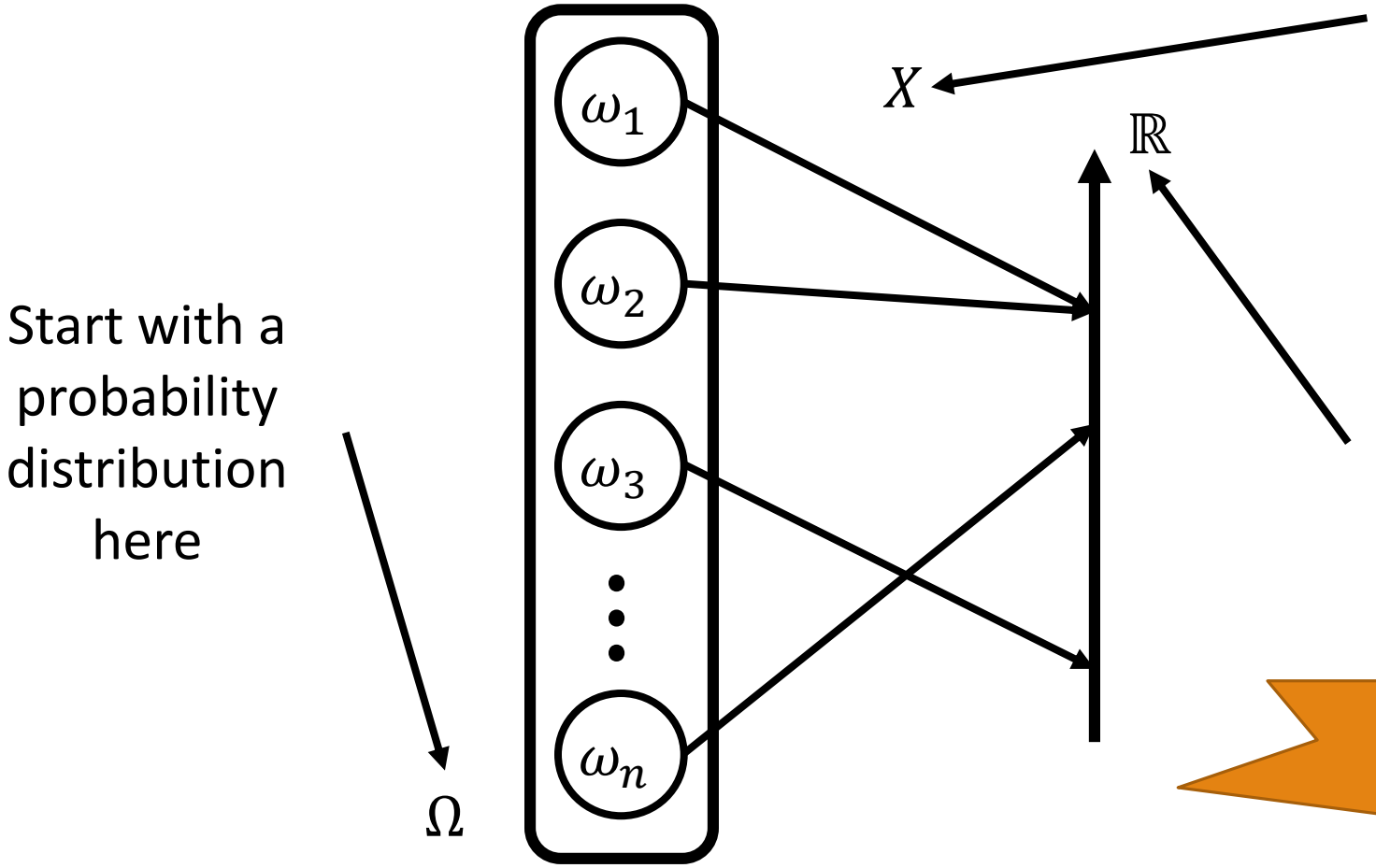
Probability distribution

probability of outcomes, e.g.,

$$p(1) = p(3) = \frac{1}{3}, p(2) = p(5) = \frac{1}{6}$$

Probability Theory Overview

Pass it "through" a random variable



Start with a probability distribution here

Get a probability distribution here

distribution of X

Probability Theory Overview

Expected value of a random variable X is defined as

$$E(X) = \sum_{\omega \in \Omega} X(\omega) p(\omega)$$

Equivalently,

$$E(X) = \sum_i i \, P(X = i)$$

Exercise: prove the equivalence

Observe that second definition does not require you to know the original distribution, only distribution of X

Probability Theory Pitfalls

Probability theory models something very natural – chance!

It is easy to think that you sort-of-understand it, and dive into problems right away, start calculating answers, and getting results (sometimes even correct).

Without understanding basic definitions you won't get very far – even beginner problems would seem incomprehensible.

You should be very clear what are probability spaces, events, random variables, expectations, etc.

To keep yourself in check, get into habit of asking yourself "What is the sample space? Is it a set of numbers/names/dates/apples? What is the size of the sample space? What is the distribution? Is it uniform?"

Examples of Pitfalls

Events have probabilities!

Random variables have expectations!

“Expectation of event A” does not make sense! Never say it!

“Probability of random variable X” does not make sense! Never say it!

Probabilities can never be negative!

Probabilities can never be greater than 1!

Things You Should Know to Understand and Analyze Randomized Algorithms

Conditional probabilities

Independent random variables

Conditional expectations

Law of complete probability

Moments of random variables

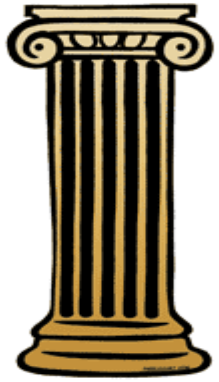
Standard discrete distr: uniform, Bernoulli, Binomial, Geometric, Poisson, …

Standard continuous distr: uniform, Gaussian, exponential, …

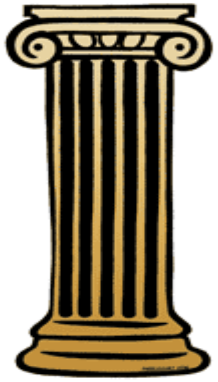
…

Three Pillars of Probability Theory

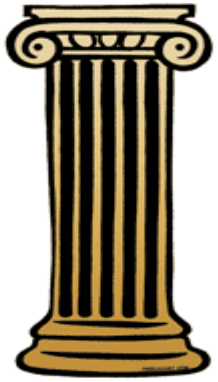
Linearity of Expectation



Union Bound



Chernoff Bound



Deceptively simple!
Incredibly powerful!
If you understand and practice these techniques, you would be able to solve MANY MANY MANY MANY MANY probability theory problems!

Back to Randomized Algorithms

Can think of randomness as an infinite supply of random bits, e.g., 1101000010100011 1111101010000010111110011001110101101010…

Algorithm accesses them sequentially, so if it runs in finite time, it only uses finitely many of the bits.

Polytime algorithm uses polynomially many random bits

Randomness in Randomized Algorithms

Suppose that on input x your algorithm uses exactly N random bits (WLOG)

What is the sample space?

$\Omega = \{0,1\}^N$

What is the probability distribution?

p is uniform, i.e., for any random string $r \in \Omega$ we have $p(r) = 1/2^N$

Randomness in Randomized Algorithms

But why do we only allow independent coin flips?

What if our algorithm requires a different distribution?

Turns out you can create samples from most other distributions (either exactly or approximately) we care about

For example, if you wanted to get a uniform random number between [0,1] to 5 bits of accuracy, you can take 5 coin flips, e.g., 01101, and put a zero followed by dot after it:

0.01101 – interpret it as a binary fractional number

Randomness in Randomized Algorithms

Another example:

Start with a probability space of n flips of a fair coin

Define a random variable X to be the number of times heads come up

Then X has binomial distribution

X can take on values 0, 1, 2, …., n

For $k \in \{0,1, \dots, n\}$ we have $P(X = k) = \binom{n}{k} 2^{-n}$

Suppose in designing an algorithm you need a sample from binomial distribution with parameter $n = 5$

Take 5 random bits from the random string and add them together

Randomness in Randomized Algorithms

More generally, any random variable whose cumulative distribution function is efficiently invertible can be efficiently sampled

Bottom line:

with just uniform random bits we can approximately simulate almost all other distributions we care about

Random Variables of Interest

x denotes input, r denotes a random string

Main random variables of interest:

$T(x, r)$ = runtime of your algorithm on input x with randomness r

$C(x, r)$ = 1, if your algorithm gives the right output on input x when it uses randomness r; 0, otherwise.

Desired Properties

$T(x, r)$ = runtime of your algorithm on input x with randomness r

$C(x, r)$ = 1, if your algorithm gives the right output on input x when it uses randomness r; 0, otherwise.

(1) Runtime

$$T(x, r) \leq poly(|x|) \text{ for every input } x \text{ and random string } r$$

Weaker requirement: $E_r\big(T(x, r)\big) \leq poly(|x|)$ for every input x

(2) Probability of success

$$P_r(C(x, r) = 1) \geq 9/10 \text{ for every input } x$$

Monte Carlo vs Las Vegas Algorithms

Monte Carlo algorithms *always* run for a small number of steps (e.g., $T(x,r) \leq poly(|x|)$) and produce a correct answer *with high enough probability* (e.g., $P_r(C(x,r)=1) \geq 9/10$).

Las Vegas algorithms run for *expected* small number of steps (e.g., $E_r\big(T(x,r)\big) \leq poly(|x|)$), but *always* produce a correct answer (i.e., $P_r(C(x,r)=1)=1$).

Thus, Las Vegas algorithms for some values of randomness might run for much longer time.

Monte Carlo vs Las Vegas

Monte Carlo algos

- Also called an algo with 2-sided error
- Probability of success in the definition is not important, as long as it is $\frac{1}{2} + \varepsilon$ for some constant $\varepsilon > 0$

Las Vegas algos

- Also called an algo with 0-sided error

Example of a Las Vegas Algorithm

Quicksort

(1) to sort an array of elements, pick a random number in the array - pivot

(2) put all numbers less than the pivot in one array, all numbers bigger than the pivot in another array

(3) recursively sort the two arrays

Note that

Quicksort always produces the right answer

Its expected runtime is $O(n \log n)$

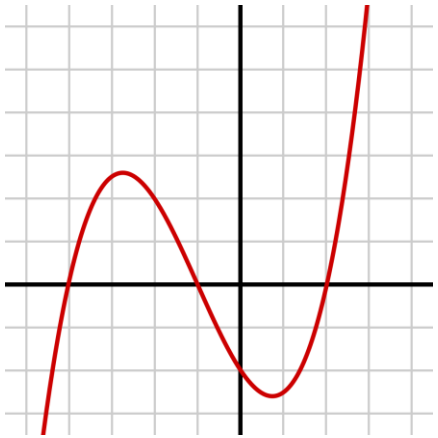
Sometimes it might run for $\Omega(n^2)$ steps if it was unlucky in pivot selections

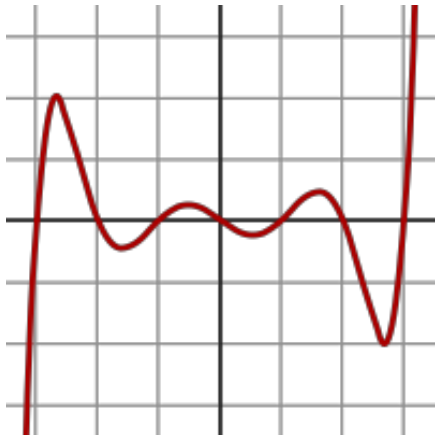
First Serious Example: Polynomial Identity Testing

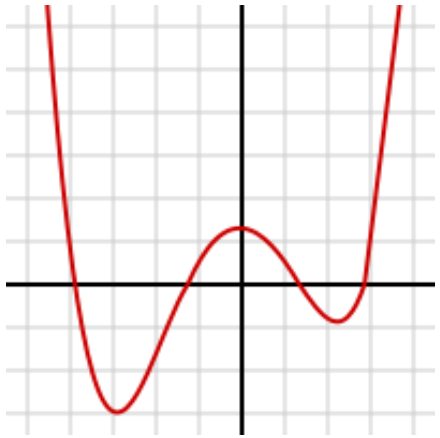
Polynomial of degree d in one variable x is a function of the form

$$p(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$$

where $a_i \in \mathbb{Q}$ (for example) and $a_d \neq 0$







Polynomial Identity Testing

Two polynomials $p(x)$ and $q(x)$ are equal if all the coefficients are the same

Sometimes, a polynomial is not represented in the above form right away

A polynomial could be represented by an implicit formula, e.g.:

$q(x) = (x-3)(10x+5)(3x^5-3x+8)\ldots(x+48)+(x^{27}-3)(2x-8)$

We know it is a polynomial, but without opening up the brackets, we don't know its coefficients! There could be nontrivial cancellations.

Note that when $q(x)$ is given implicitly as above, its degree is at most the size of such an implicit representation. In what follows, we assume d is both a bound on degree and size of representation.

Polynomial Identity Testing

Given two polynomials $p(x)$ and $q(x)$ in such an implicit form, we would like to test if they are the same polynomial

Suppose that the degree of each polynomial is at most d

We could open up the brackets and simplify the polynomials, but it is too much work – fast and complicated algorithms for doing this require time $\Omega(d \log d)$

Can we do better?

Can we check if two polynomials given implicitly are the same in time $O(d)$ and without opening up the brackets?

Polynomial Identity Testing to Polynomial Zero Testing

Instead of checking $p(x) = q(x)$ we can check if $p(x) - q(x) = 0$.

If $p(x)$ and $q(x)$ are degree at most d polynomials then $p(x) - q(x)$ is also degree at most d polynomial.

So, now the problem becomes

Given a polynomial $p(x)$ of degree at most d represented implicitly, check if it is the identically 0 polynomial without opening up the brackets and simplifying all the coefficients

Polynomial Zero Testing

Problem: given a polynomial $p(x)$ of degree at most d represented implicitly, check if it is the identically 0 polynomial without opening up the brackets and simplifying all the coefficients

Key observations:

(1) polynomial $p(x)$ of degree d has at most d roots – those numbers where the polynomial evaluates to zero

(2) evaluating a polynomial of degree at most d can be done with $O(d)$ basic arithmetic operations (addition, subtraction, multiplication) – Exercise! (recall that d is both the bound on the implicit representation and on the degree)

Randomized Polynomial Zero Testing

Problem: given a polynomial $p(x)$ of degree at most d represented implicitly, check if it is the identically 0 polynomial without opening up the brackets and simplifying all the coefficients

Randomized algorithm:

- Pick a number r uniformly at random from $\{1,2, \dots, 10d\}$
- Evaluate $p(r)$. If it returns non-zero, output "NON-ZERO POLY"; otherwise, output "ZERO POLY"

Analysis of Randomized Zero Testing Algo

Runtime: $O(d)$ – always

Correctness:

if $p(x)$ is the zero polynomial, it always evaluates to zero, so our algorithm is always correct!

If $p(x)$ is non-zero, it has at most d roots, so at most d out of $10d$ first numbers evaluate to zero. With probability $9/10$ we select a number on which $p(x)$ does not evaluate to zero, so we output the right answer with probability $9/10$

This is a Monte Carlo algorithm with one-sided error.

Error probability can be boosted by either repetition or selecting a bigger range

Boosting the Probability of Success

Repeat the algorithm k times with fresh randomness

Output “ZERO” only if all runs output “ZERO”

Probability of success?

If $p(x)$ is the zero polynomial, we always output “ZERO”

If $p(x)$ is not the zero polynomial, we make an error if and only if ALL runs make an error. Since runs are independent, probabilities multiply:

Probability that a single run makes an error is $1/10$, then the probability that all runs make errors is $1/10^k$.

Cosmic Rays

From Wiki:

“Cosmic rays are high-energy radiation, mainly originating outside the Solar System and even from distant galaxies. Upon impact with the Earth's atmosphere, cosmic rays can produce showers of secondary particles that sometimes reach the surface.”

When they hit a RAM module on your laptop they can corrupt main memory

Depending on where cosmic rays hit, your program can crash, or simply continue execution with erroneous data

Boosting the Probability of Success

If your program runs for 1 minute and occupies 20 MB of RAM, IBM estimates that there is roughly $1/10^7$ chance that cosmic rays will hit the RAM module and introduce errors to your program.

If you run the repeated zero polynomial testing algorithm for $k = 10$ iterations, the probability of error is $1/10^{10}$, but it still runs in $10\ O(d) = O(d)$ time.

For all practical purposes, the result of this randomized algorithm is as dependable as any other computation that you run on your computer

BUT the algorithm is super simple and super efficient!

Extending the Algorithm to Multiple Vars

Example: polynomial in 3 variables x_1, x_2, x_3

$$p(x_1, x_2, x_3) = 10x_1^3 x_2^4 x_3 + 8x_1 x_2 x_3^4 - x_1 x_3 + x_2 - 6$$

Total degree is 8 (maximum sum of degrees in a single term)

More generally, we will consider polynomials in many variables

These polynomials can also be represented implicitly

We want to see when two polynomials in several variables are the same

Alternatively, when a single polynomial in several variables is the zero polynomial

Nontrivial Example

Consider a square $n \times n$ matrix whose entries are variables x_{ij}:

$$M = \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nn} \end{pmatrix}$$

The determinant of M is defined as follows:

$$\det(M) = \sum_{\sigma}(-1)^{sgn(\sigma)} x_{1\sigma(1)} x_{2\sigma(2)} \cdots x_{n\sigma(n)}$$

Where σ ranges over all permutations of $\{1,2,\ldots,n\}$

Thus, the determinant is a polynomial in n^2 variables of total degree n and with $n!$ terms

Nontrivial Example

$$\det(M) = \sum_{\sigma}(-1)^{sgn(\sigma)} x_{1\sigma(1)} x_{2\sigma(2)} \cdots x_{n\sigma(n)}$$

n^2 variables, total degree n, $n!$ Terms

For $n = 100$, there is not enough material in the observable universe to create the paper necessary to write down this polynomial explicitly

Nonetheless, for a particular setting of values to variables x_{ij} we can evaluate this polynomial efficiently using Gaussian Elimination!

This is an implicit representation of a polynomial: evaluation is quick, but we cannot expand the polynomial coefficients

Multivariate Polynomial Zero Testing

Polynomial Zero Testing with one variable required a bound on the number of roots of a polynomial of degree n

Thus, we need a generalization of such a bound to several variables and total degree

Schwartz-Zippel Lemma. Let $q(x_1, x_2, \dots, x_n)$ be a non-zero multivariate polynomial of total degree d over $\mathbb{Q}$. Let S be a finite subset of $\mathbb{Q}$, and let $r_1, r_2, \dots, r_n$ be chosen independently and uniformly at random from S. Then

$$P(q(r_1, r_2, \dots, r_n) = 0) \leq \frac{d}{|S|}$$

Multivariate Polynomial Zero Testing

INPUT: polynomial q on n variables of total degree d given implicitly

OUTPUT: yes if q is identically zero polynomial, no otherwise

Algorithm:

Pick $r_1, r_2, \ldots, r_n$ uniformly at random from $\{1,2, \ldots, 10d\}$

Output yes if and only if $q(r_1, r_2, \ldots, r_n) = 0$

Analysis:

If q is zero then we always output yes

If q is nonzero, by Schwarz-Zippel, we output no with probability at least $9/10$

Multivariate Polynomial Zero Testing

Algorithm:

Pick $r_1, r_2, \dots, r_n$ uniformly at random from $\{1,2, \dots, 10d\}$

Output yes if and only if $q(r_1, r_2, \dots, r_n) = 0$

As before, we can boost the probability of success to be almost 1 by repetition

Example of a problem for which the best known algorithm is randomized

Can this algorithm be derandomized? Huge open question in theoretical CS

Interesting Application

The Vandermonde matrix $M(x_1, x_2, \dots, x_n)$

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}$$

Vandermonde's identity: $\det(M) = \prod_{j<i}(x_i - x_j)$ (EXERCISE – prove it!)

Vandermonde's Identity

$$\det\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} = \prod_{j<i}(x_i - x_j)$$

This is polynomial identity testing!

Cannot write down and simplify the poly on the left – it has $n!$ Terms

BUT can evaluate it efficiently using Gaussian elimination

Thus, can use our algo to verify equality of polynomials with high probability

The difference with a mathematical proof is that in pure math we are not satisfied with knowing that a theorem holds with 99% certainty :)