CS1114: Study Guide 3

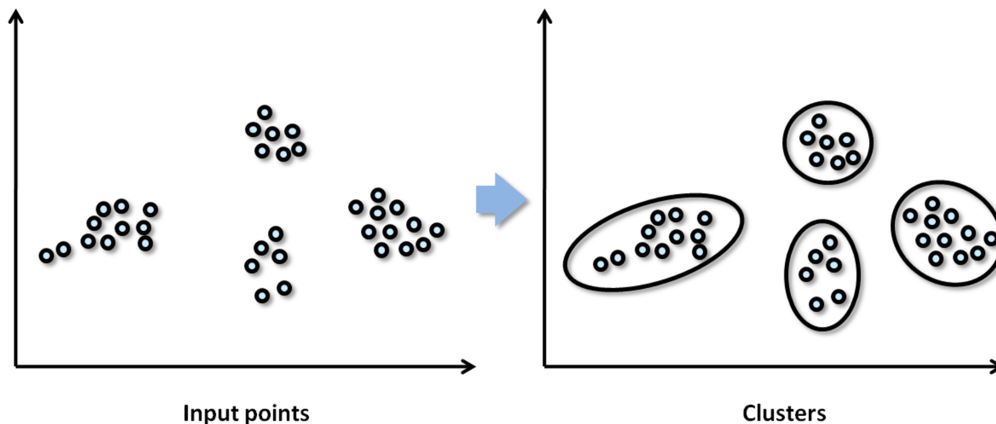
This document covers the topics we've covered in the final part of the course. Please refer to the class slides for more details.

1 Clustering redux and greedy algorithms

We covered part of clustering in the previous study guide, but we revisit it here. To motivate clustering using the text analysis we saw in the final part of the course, suppose we are given the following problem: someone gives us a bunch of texts whose authors are completely unknown (there was a mixup at the library, say), and asks us to categories the texts into groups all written by the same author. All we know is that texts written by the same author are likely to have similar statistics, and texts written by different authors are likely to have different statistics. We may or may not even know how many authors there were. This is an example of a *clustering* problem—we want to cluster the texts into groups such that the texts in the same group are similar, and texts in different groups are different.

Clustering problems are widespread in many different domains. Businesses often want to cluster our customers into different groups in order to better tailor their marketing to specific types of people (market segmentation). News aggregation websites like Google Maps cluster news stories from different sources into groups of stories that are all on the same basic topic. And clustering can be use to find epicenters of disease outbreak (such as the recent swine flu epidemic).

In our case, we will state the clustering problem as follows: we are given a set of n data points $x_1, x_2, \ldots, x_n$ (represented by vectors of some dimension), and a way to compute distances between these points (for instance, the typical Euclidean distance between vectors). An example clustering of 2D points (with four clusters) is shown below:



How do we define the clustering problem? We want to create a set of clusters and assign each point to a single cluster so as to maximize some underlying cost function that rates the

"goodness" of a clustering (or minimize "badness" or cost). This concept of optimization is an important one, appearing in many different contexts, and one critical part of it is to define a good objective function. What should this objective function look like for clustering? We have seen several possibilities, including one, k-means, in the previous part of the course. But to recap, untuitively, we want a cost function that prefers clusterings where similar points are close together, and where dissimilar points are far away. One of the simplest approaches is known as k-means.

1.1 k-means

The k-means clustering problem is as follows: we are given the number of clusters k we should compute (e.g., $k = 4$). We need to find (a) a set of cluster centers $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k$, and (b) an assignment of each point x_i to a cluster C_j, so as to minimize the sum of squared distances between points and their assigned cluster center. That is, we want to find the cluster centers and cluster assignments that minimize the following objective function:

$$\mathsf{c}ost(\bar{x}_1, ..., \bar{x}_k, C_1, \ldots, C_k) = \sum_{j=1}^{k} \sum_{x_i \in C_j} ||x_i - \bar{x}_j||^2$$

Where $||x - y||$ is the distance between two vectors or points x and y. At first, this problem looks like the kind of least squares problems we've seen in the past (such as line fitting), and which we know are easy to solve. Unfortunately, however, there's a critical difference: we need to find the cluster centers *and* the cluster assignments at the same time, and this makes the problem much more difficult. If we knew either of the two then the problem would be easy: given the cluster centers, the optimal assignment is to assign each data point to the closest cluster center; given the cluster assignment, the optimal centers are the means of each cluster. But we need to compute both.

This small difference means that, far from being easy to solve, the k-means problem is in a class of problems that are *extremely* difficult. No known algorithm exists for computing the optimal k-means in less than exponential time in n (essentially, we have to try every possible clustering to find the one, and there are $O(k^n)$ possible clusterings). Compared to polynomial time algorithms for other problems we have seen—sorting ($O(n^2)$), median finding ($O(n)$), convex hull ($O(n^2)$), exponential time algorithms are *much* worse (it is similar to comparing snail sort to quicksort). The class of hard problems of which k-means is a member is known as NP-hard problems (NP stands for non-deterministic polynomial time, though you won't learn why until much later). We'll mention other hard problems later in this document.

Should we just give up? No, we can still hope to create a fast algorithm that's pretty good (for instance, one that always gets an answer that is pretty close to the optimal one). We saw one approach, *Lloyd's algorithm*, last time—remember that Lloyd's algorithm is an *iterative algorithm* which iterates between computing cluster centers and cluster assignments, making the clustering better and better over time. Here, we will discuss another approach: the *greedy* algorithm.

1.2 Greedy algorithms

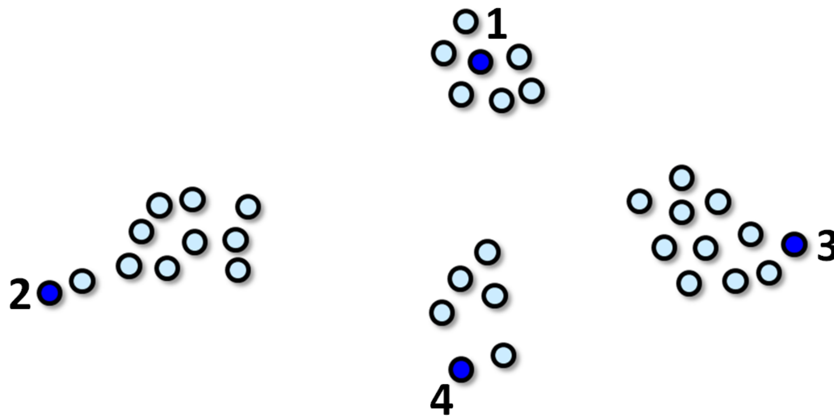
A greedy algorithm is one that always does whatever seems best at any given moment, without a long-term plan, and without revisiting past decisions. For some problems greedy algorithms are optimal—i.e., they give the best possible answer—while for other problems, they can give answers that are quite bad. An example problem where a greedy algorithm work well is making change with US currency with the smallest possible number of coins. This algorithm gives back the largest denomination of coin until the remaining amount is less than the value of that coin, then continues with the next largest denomination, etc., until the exact amount is returned. For instance, if all I can give back are quarters, dimes, nickels, and pennies, then a greedy algorithm would give back the quarters until the remaining change is less than $0.25, then give back dimes until the remaining change is less than $0.10, then repeats for nickels and pennies. For US currency, this simple algorithm is optimal under the objective function of "returning the minimum number of coins"—that is, this greedy algorithm always gives back the exact amount of change with the smallest possible number of coins. (Unfortunately, as we saw in class, this algorithm doesn't work so well for more unusually currency; imagine how you would give back 98 cents if all you had were 50 cent pieces, 49 cent pieces, and 1 cent pieces.)

We can apply this idea to the k-means problem as follows:

ALGORITHM KMEANSGREEDY($x_1, x_2, \ldots, x_n, k$)

1. Select one of the data points at random, and call it the first center.
2. Select the data point furthest from the first one, and call that the second cluster center.
3. Repeat by finding the data point furthest from all of the centers selected so far (i.e., the point for which the closest center is as far away as possible), and adding the point to the set.
4. Stop when we have selected k points.
5. Associate each data point with its closest cluster center.

For instance, if we ran this algorithm on the 2D point set below, we might end up selecting the blue points as our cluster centers (in the order indicated by the numbers):



How well does KMeansGreedy work? Unfortunately, not well at all. The cost of the resulting cluster centers and cluster assignments can be much worse than the cost of the optimal clustering. However, if we tweak the problem slightly, the same greedy algorithm works much better.

1.3 k-centers

The k-centers problem is a variation on k-means, sometimes used in *facility location*. We're looking for exactly the same output (a set of cluster centers and a cluster assignment), but instead of minimizing the sum of squared errors, we want to minimize the *maximum* distance from a point to its cluster center. One way to think about this problem is that we're planning to build k hospitals in a city with n houses, and we want to place the hospitals so that the furthest distance from a house to the closest hospital is as small as possible (we want our ambulances to get to any house in as short a time as possible). We can write this objective function as follows:

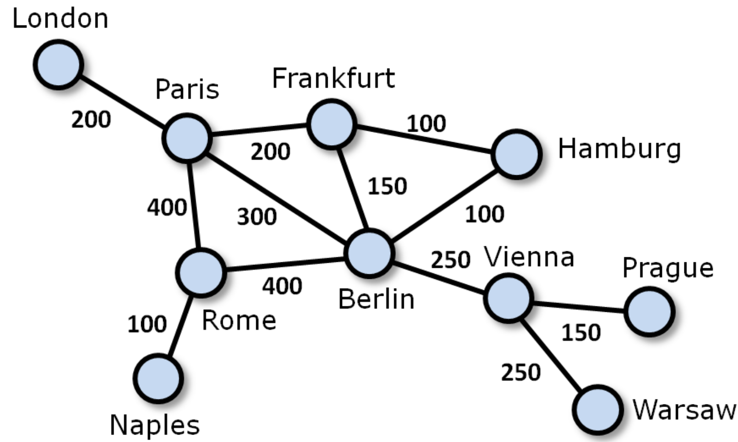
$$\mathsf{c}ost(\bar{x}_1, ..., \bar{x}_k, C_1, \ldots, C_k) = \max_{j=1}^{k} \max_{x_i \in C_j} ||x_i - \bar{x}_j||^2$$

(the square at the end is optional in this case, as the optimal solution is the same in both cases).

Unfortunately, the k-center problem is also NP-hard. But what happens when we apply KMeansGreedy to the k-center problem? Amazingly, we will always get a clustering whose cost is no more than two times worse than the optimal. The proof of this fact is beyond the scope of this course, but it suggests something very interesting: we can't easily compute the optimal answer, but we can compare the cost of the "greedy" answer to the cost of the optimal answer.

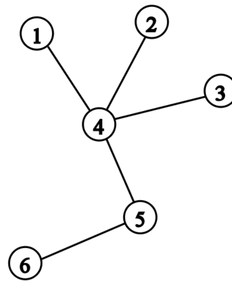
1.4 Minimum spanning trees

In a moment, we will look at yet another objective function for clustering that is much easier to optimize, but first, we will take a slight detour into graphs. First, we define a new type of graph called a *weighted* graph, in which each edge has a number associated with it (similar to the Markov chain graph above). Intuitively, this edge expresses some kind of distance between the two endpoints (for clustering, we will make this weight the actual distance between two input points, represented by nodes). For instance, below we have a simplified graph of the European rail system, in which each edge is weighted with the distance (in miles) between the two cities it connects:



We define the *cost* of a graph as the total sum of edge weights in the graph. The cost of the graph above is 2600 miles.

Next, recall that a tree is a graph that is connected and has no cycles, such as the graph below:

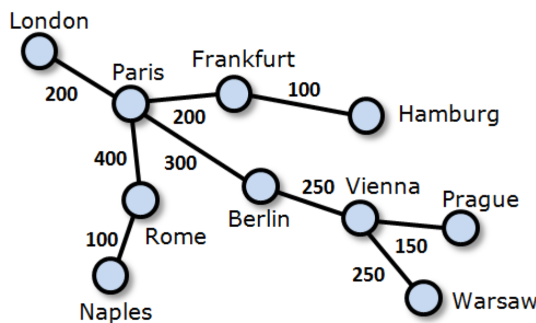


A *spanning tree* is a subgraph of a (connected) graph that has two properties:

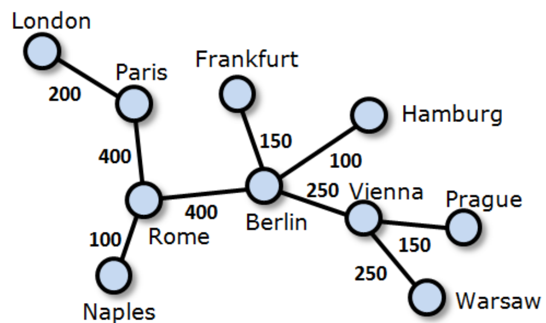
1. It connects all the vertices in the original graph.
2. It is a tree.

(Note that a subgraph is exactly what it sounds like—a graph that has a subset of the vertices and edges of another graph.)

The graph above has several spanning trees, including the two below:



Cost: **1950**



Cost: **2000**

Note that a spanning tree for a (connected) graph with n vertices always has n vertices and $n-1$ edges.

A fundamental problem in computer science is the *minimum spanning tree* (MST) problem: the problem of finding the spanning tree with minimum cost. As illustrated above, different spanning trees have different costs. One way to think about this problem (in the context of the graph above) is that we want to connect all of the cities represented in the graph (i.e., it should be possible to reach any city from any other city), but we want to lay as few miles of track as possible. The MST tells us how to do this.

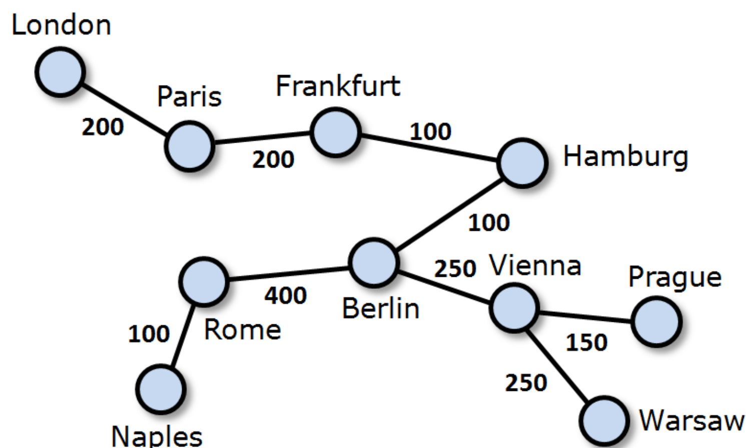
We can try to compute the MST with the following greedy algorithm:

Algorithm MSTGreedy($x_1, x_2, \ldots, x_n, k$)

1. Start with a graph with all the vertices but no edges.
2. Sort all of the edges by increasing weight.
3. For each edge in order, add it to the graph, unless it connects two nodes that are already in the same connected component.

This algorithm is known as Kruskal's algorithm (and is one of *many* algorithms for computing an MST). One nice fact is that it always results in a spanning tree. One even nicer fact is that it always gives us the *minimum* spanning tree. Again, the proof of this is beyond the scope of this class.

The graphs show the result of running Kruskal's algorithm on the rail network above. To see Kruskal's algorithm in action, please see the slides for Lecture 19.



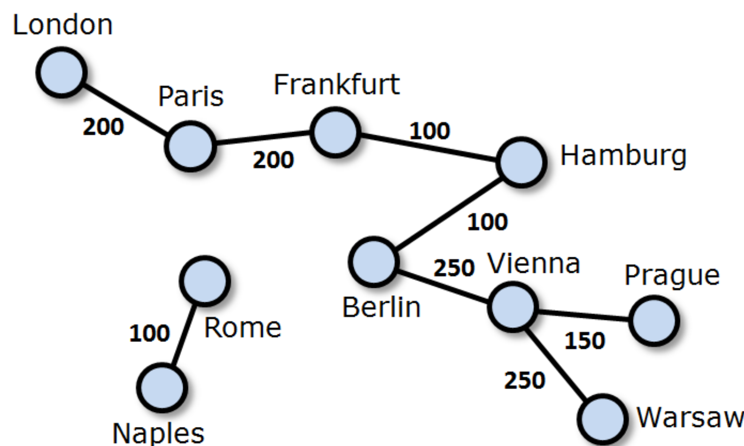
The cost of the MST shown above is 1750. Thus, at minimum, we have to lay 1750 miles of track to connect all the cities together (though in practice, we would probably want to lay more based on which lines would be heavily trafficked, to make it a convenient way to travel).

1.5 Maximum spacing clustering

How can we use the minimum spanning tree for clustering? Let's consider another objective function for clustering: maximizing *spacing*. We define the spacing of a set of clusters as the distance between the closest pair of points in different clusters. For a good clustering, we

would expect the spacing to be large; hence, maximizing the spacing seems like a reasonable objective function for clustering. As it turns out, we just talked above an algorithm that computes clustering with maximum spacing.

First, note that the MST algorithm, after adding m edges, results in a graph with $n - m$ connected components (we start with each vertex in its own connected component, and each time we add an edge, we connect two previously disconnected components, reducing the count by one). We can consider these connected components as our clusters; after adding m edges, we have $n-m$ clusters. Thus, to get k clusters, we just need to stop the MST construction with Kruskal's algorithm $k - 1$ edges early. As it turns out, these clusters will have the maximum spacing (the proof of this fact is in the slides, but won't be necessary to know for the exam). Another way to say this is that to find the clustering with the maximum spacing, we can find the MST using Kruskal's algorithm, then remove the last $k - 1$ edges we added. The spacing for this clustering is the weight of the next edge we would add to the MST, by definition—Kruskal's algorithm always adds the shortest edge that connects two different components (or clusters). For instance, the two clusters with the maximum spacing for the rail network above is shown below, and the spacing is 400 (the weight of the next edge we would add to connect these components, i.e. Rome to Berlin (or Rome to Paris)). The two clusters correspond to the two connected components of the graph (Italy vs. the rest of Europe).



The MST is a fairly simple problem, therefore finding the clustering with the maximum spacing is also easy (much easier than the k-means or k-center problems).

1.6 Conclusion

To recap, here are some of the important facts covered in this section:

1. The k-means problem is very hard, and the greedy algorithm can perform very poorly. An iterative algorithm (Lloyd's algorithm) is also non-optimal, but is used frequently in practice.

2. The k-center problem is also very hard, but the greedy algorithm the guaranteed to give us answer with cost *no worse* than twice the cost of the optimal solution.

3. Finding the k clusters with the maximum spacing is much easier, and can be solved by running Kruskal's algorithm for MST, stopping $k-1$ edges early.

2 Markov chains

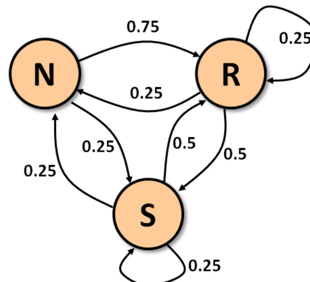
Sequences of things come up all the time when dealing with the real world. The closing stock price of Google each day, the average temperature each day in Ithaca, the words in James Joyce's *Ulysses*, and the nucleotides that make up our DNA are all examples of sequences. One simple way to model and predict sequences is with *Markov chains*. A Markov chain is a model of sequences that makes the assumption that the next item in a sequence is only dependent on a small number of items that come before it (often just one—known as a first-order Markov chain). This is known as the *Markov assumption*.

A Markov chain can be represented in several ways. In class we only considered *discrete* Markov chains—chains that describe sequences made up of a finite number of symbols or *states* (e.g., English words or DNA; the average temperature in Ithaca, on the other hand, is a *continuous* quantity, as it can take on any real number). We can represent a discrete first-order Markov chain as simple a table or matrix giving the probabilities of transitioning from each state to every other state. One simple example we saw in class was a Markov chain describing the weather in Ithaca during springtime, assuming that there were three possible weather conditions: nice (N), snowy (S), and rainy (R). Here is a transition matrix (consisting of (mostly) made-up numbers) describing such a chain:

	N	S	R
N	0.0	0.75	0.25
S	0.25	0.25	0.5
R	0.25	0.5	0.25

In a transition matrix, the rows correspond to the *previous* state, which we will call $\mathbf{x}_{t-1}$, and the columns correspond to the *current state*, which we will call $\mathbf{x}_t$. Thus, according to this Markov chain T, if it was nice yesterday, the probability that it will be snowy today is $T_{N,S} = 0.75$, or 75%. Note that each row of a transition must add up to one (so that each row is a valid probability distribution), though each column may or may not add up to one.

Another way to visualize a Markov chain is as a graph on the set of states, where each edge (u, v) is weighted with the probability of transitioning from state $\mathbf{x}_{t-1} = i$ to state $\mathbf{x}_t = v$. The graph corresponding to the Markov chain above is:



Note that this graph has *self-loops*, or vertices that point to themselves. We've left off the edges with 0 probability (in this case, only the edge from state **N** to itself.

If we know what happened yesterday, a Markov chain gives us the probabilities of being each state today. We can also use a Markov chain to predict what will happen further into the future. For instance, if it was nice yesterday, we can compute the probability that it will be nice tomorrow (i.e., after two transitions) by simply computing the probability of all possible ways it could be nice again two days after being nice, and adding the probabilities together. The possible ways are $N \to N \to N$, $N \to S \to N$, and $N \to R \to N$. The probability of $N \to N \to N$ is $0.0 \cdot 0.0 = 0.0$, the probability of $N \to S \to N$ is $0.75 \cdot 0.25 = 0.1875$, and the probability of $N \to R \to N$ is $0.25 \cdot 0.25 = 0.065$, so the probability of having a nice day two days after a nice day is $0.0 + 0.1875 + 0.065 = 0.25$ or 25%.

Note that this sum of products is simply the dot product of the row $\mathbf{x_{t-1}} = \mathbf{N}$ with the column $\mathbf{x_t} = \mathbf{N}$. In fact, we can compute a matrix giving the probabilities of states at time $\mathbf{x_{t+1}}$ given the state at $\mathbf{x}_{t-1}$ by simply multiplying T times itself:

$$T^2 = \begin{bmatrix} 0.0 & 0.75 & 0.25 \\ 0.25 & 0.25 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{bmatrix} \begin{bmatrix} 0.0 & 0.75 & 0.25 \\ 0.25 & 0.25 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{bmatrix} = \begin{bmatrix} 0.2500 & 0.3125 & 0.4375 \\ 0.1875 & 0.5000 & 0.3125 \\ 0.1875 & 0.4375 & 0.3750 \end{bmatrix}$$

So, for instance, if it was snowy yesterday, the probability that it will be snowy tomorrow is $T^2_{S,S} = 0.5$, or 50%. A matrix predicting the state at time $t + k$ can be computed by taking T to the $k + 1^{th}$ power: T^{k+1}.

How do we create the probabilities for a Markov chain. We usually compute these based on a large sequence of existing data. For instance, if we had ten years worth of weather data for Ithaca, we could use this data to compute the transition probabilities for a Markov chain. In class, we looked at applying this technique to text: given a work (or set of works) by an author, we analyzed the text to compute a transition matrix describing the sequence of words used by that author. Here's a very simple example of a transition matrix created from the text

> "A dog is a man's best friend. It's a dog eat dog world out there."

	a	dog	is	man's	best	friend	it's	eat	world	out	there	.
a		2/3		1/3								
dog			1/3					1/3	1/3			
is	1											
man's					1							
best						1						
friend												1
it's	1											
eat		1										
world										1		
out											1	
there												1
.							1					

In this case, we've chosen to represent a period ('.') as its own word (though we could choose to ignore punctuation entirely). According to this transition matrix, the probability of generating the string

"dog eat dog eat dog eat dog"

(assuming we generate a 7 word string starting with "dog") is the product of the transition probabilities

$$\textbf{dog} \rightarrow \textbf{eat} \rightarrow \textbf{dog} \rightarrow \textbf{eat} \rightarrow \textbf{dog}$$

or

$$\frac{1}{3} \cdot 1 \cdot \frac{1}{3} \cdot 1 \cdot \frac{1}{3} = \frac{1}{27}.$$

The probability of generating the string

"dog is a dog eat world,"

on the other hand, is zero, because the probability of a transition **eat** $\rightarrow$ **world** is zero. In Assignment 6, you dealt with the problem of zero probability strings, and with the problem of determining authorship of disputed texts—given a large database of text with known authorship.

3 Hard problems in computer science

Let us end with a discussion of problems that are considered *hard* in computer science, by a certain specific definition of hard (i.e., NP-hard). For optimization problems, such as k-centers, you can think of these as problems for which we know of no polynomial-time algorithm

that finds the optimal answer to the problem in general. (I.e., we know of know polynomial-time *exact* algorithm.) Here, polynomial-time means $O(n^k)$ for some constant k—for instance, $O(1)$, $O(n)$, $O(n^2)$, or $O(n^100)$. For instance, for some problems the best know algorithm is exponential in n, i.e. $O(2^n)$, which is much worse than any power of n (even n^100). On the other hand, we may have algorithms that can compute an approximate answer quickly, such as the greedy k-centers algorithm described above.

3.1 Travelling salesman problem

One of the most famous hard problems in CS is the travelling salesman problem (TSP). Imagine a salesman who has to drive to a set of n cities as part of his sales job (or, say, a postal worker who needs to visit n houses), returning to the first city at the end (this is called a "tour"). The salesman can otherwise visit the cities in any order. Which order results in the smallest required distance travelled? This is a natural optimization problem (minimize total distance travelled), and it turns out that some ordering are much better than others. For instance, if the cities are Boston, NYC, San Francisco, and LA, the tour

```
Tour 1: Boston -> NYC -> San Francisco -> LA -> Boston
```

would probably be much better than the tour

```
Tour 2: Boston -> San Francisco -> NYC -> LA -> Boston
```

because `Tour 1` criss-crosses the United States from east coast to west and back once, instead of twice (as in tour 2).

More formally, we can state the TSP as follows: given a weighted, complete graph G, with n nodes, compute a cycle containing all of the nodes exactly once (except the start and end node)—i.e., a tour—that has the minimum total cost over all tours (where the cost of a tour is the sum of edge weights along the tour).

There is no known polynomial-time algorithm that is guaranteed to find the best solution to the TSP in general. However, we can still try and come up with algorithms that find "pretty good" answers. In class we talked about a greedy algorithm for the TSP. This algorithm starts with some node v (randomly chosen, say), then selects the smallest edge (v, u) going out of v, and visits u next. Then the smallest edge going to a node not already visited is selected to visit next, and so on until all nodes have been selected once. It's easy to find cases where this greedy algorithm fails to find the shortest tour, however.

One could also come up with an incremental algorithm, that starts with a random tour (a random permutation of the nodes), then swaps pairs of nodes if the swap will result a tour of lower cost (a bit like bubblesort). This continues until no swap yields a better tour—we have then reached a tour that is a "local minimum"—no simple swap will result in a better tour. Unfortunately, this algorithm also fails to find the shortest tour in general.

3.2 Other hard problems

Many other interesting optimization problems cannot be solved exactly in polynomial time (at least, if such algorithms exist, we haven't found them yet). These problems include graph coloring (color a graph with the minimum possible number of colors), finding the largest clique in a graph, playing Tetris, and the general Sudoku problem.