

Approximation Algorithms

<https://cs.pomona.edu/classes/cs140/>

Outline

Topics and Learning Objectives

- Discuss strategies for finding solutions to **difficult** problems
- Apply an approximation algorithm to an NP-Hard problem

Exercise

- None

NP-Complete

What does it mean if your problem is NP-Complete?

1. It belongs to NP, and
2. It belongs to NP-Hard.

What does it mean to belong to NP?

- We can **verify** a solution as correct or incorrect in polynomial time.

What does it mean to belong to NP-Hard?

- We do not know an algorithm to **solve** it in polynomial-time.

So, your problem is NP-Hard...

- This **does not** mean you cannot solve your problem.
- This **does not** mean that you cannot get an optimal solution.
- It does mean that you should set your expectations appropriately.
- You are probably not going to accidentally prove that $P = NP$.

Strategies

1. Focus on solving a special case that is tractable
 - The general Knapsack problem is NP-Complete, but we solved it by looking at problems where the total capacity W was $O(nW)$.
1. Solve the problem in exponential time (but faster than brute-force)
 - We looked an algorithm for TSP that runs in $O(n^2 2^n)$ instead of $O(n!)$
2. Solve the problem using some heuristics
 - These algorithms are **not guaranteed to give optimal solutions**,
 - but they are (generally) **fast**.

The Traveling Salesman Problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

- Input: a **complete**, undirected graph with non-negative edge costs
- Output: a minimum cost tour (a cycle that visits each vertex once)

Solving the TSP

- There are $n!$ total possible tours.

Input Size	Brute-Force $n!$	Exponential $O(n^2 2^n)$
14	87 billion 178 million ...	~ 3 million
15	1 trillion 307 billion ...	~ 7 million
16	20 trillion 922 billion ...	~ 16 million ...
30	265 nonillion 252 octillion 859 septillion 812 sextillion 191 quintillion 58 quadrillion 636 trillion ...	~ 966 billion 367 million ...

Why is TSP so difficult?

Doesn't it seem like it is just a special case of SSSP, with one extra edge back to the start vertex?

Remember our SSSP sub-problems (Bellman-Ford):

For every edge edge budget (**FOR** num_edges **IN** [$0 \dots n$])

Let L_{ij} = the length of the shortest path from 1 to j that uses at most i edges

Why is TSP so difficult?

For every edge budget (**FOR** num_edges **IN** [$0 \dots n$])

Let L_{ij} = the length of the shortest path from 1 to j that uses at most i edges

How are they different?

- Subproblems of SSSP do not solve the original TSP problem (SSSP does not **require** the use of i edges).
- SSSP doesn't enforce that we cannot visit a vertex more than once.
- If we change SSSP to enforce the use of i edges with no repeats, we lose the ability to solve larger problems from smaller problems.

Dynamic Programming for TSP

For every destination j in $\{1, 2, \dots, n\}$, and
for every subset S of $\{1, 2, \dots, n\}$

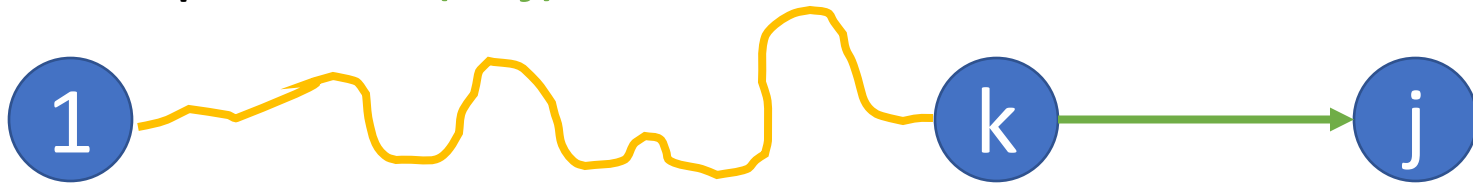
$L_{s,j}$ = the minimum length of a path from 1 to j that visits all
of the vertices in S

How does this improve on brute-force?

- It does not care about the order in which we visit the vertices in S .
- But, there are still an exponential number of choices for $S \rightarrow O(2^n)$

Optimal Substructure Lemma

- Let P be a shortest path from 1 to j that visits S .
- If the last hop of P is (k, j)



- Then P' is the shortest path from 1 to k

$$L_{i,j} = \min_{k \in S, k \neq j} L_{S-\{j\},k} + C_{kj}$$

What if we don't need the optimal path?
Just one that is "good enough"?

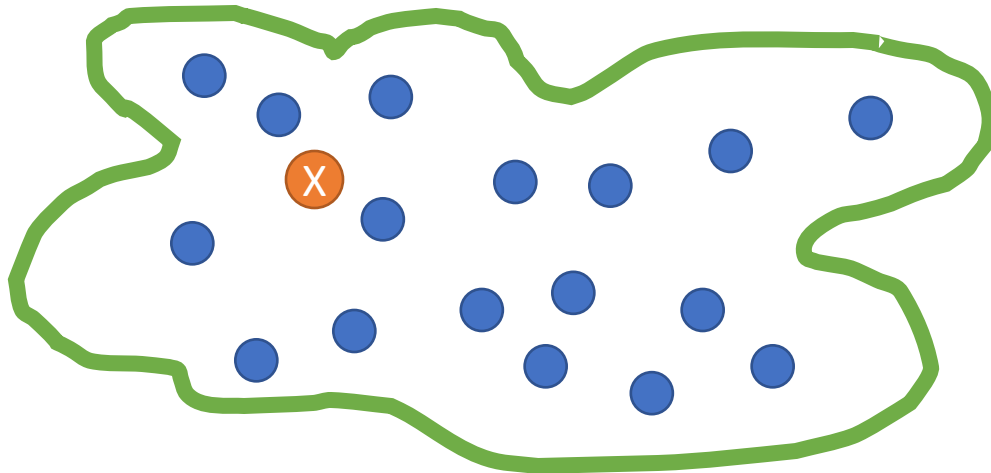
Local Search Heuristic for Hard Problems

Local Search

- Let X be a set of candidate solutions to a problem
- For example, let it be all possible tours of a graph

The key to local search is to define a neighborhood:

- For each x in X , specify which y in X are its “neighbors”

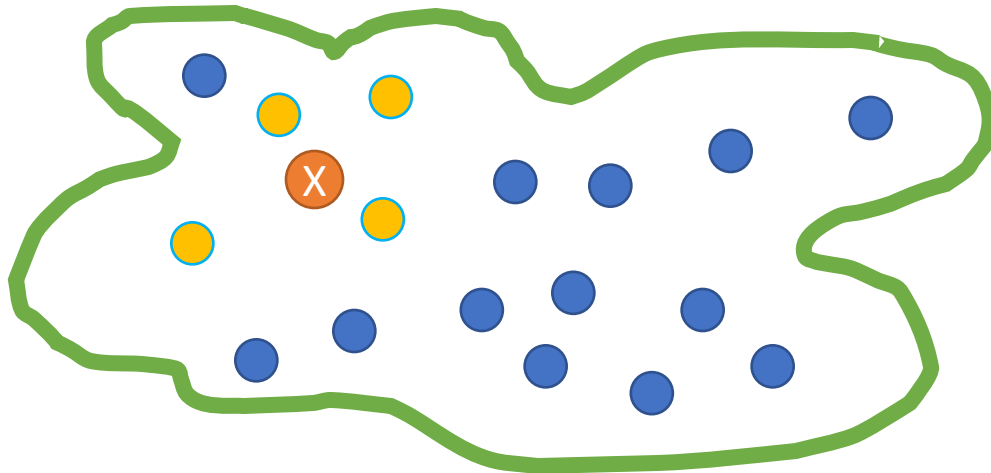


Local Search

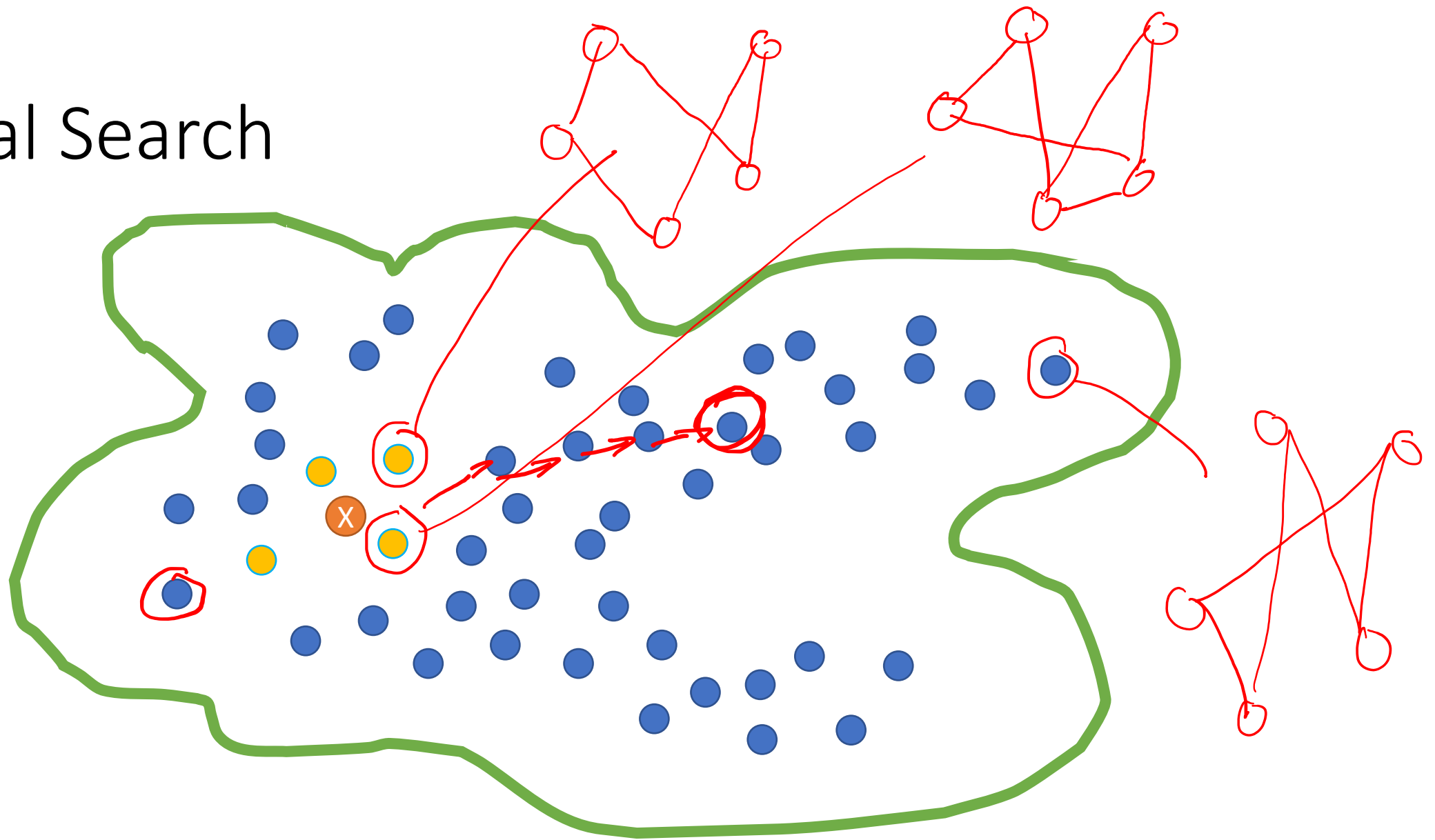
- Let X be a set of candidate solutions to a problem
- For example, let it be all possible tours of a graph

The key to local search is to define a neighborhood:

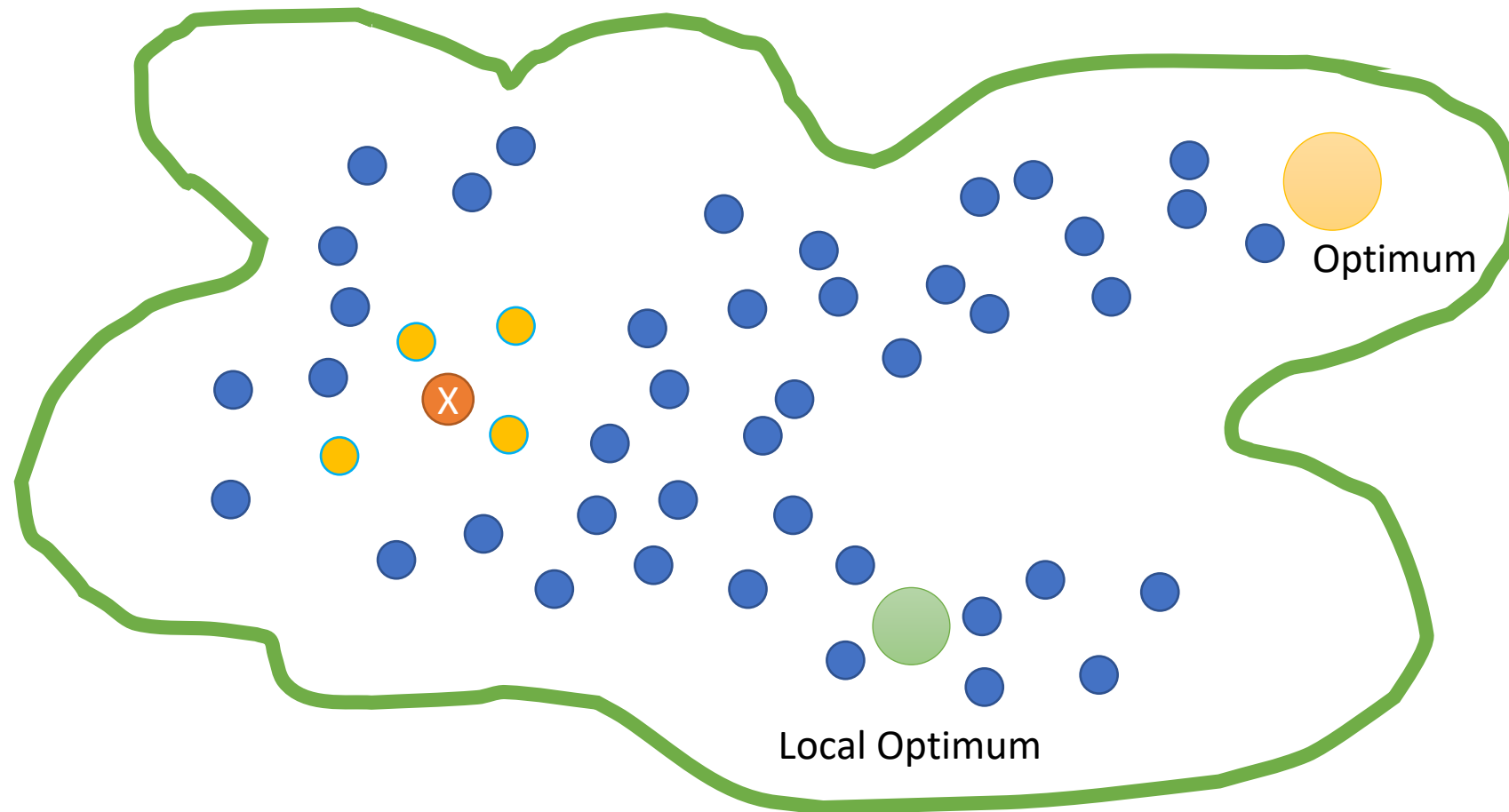
- For each x in X , specify which y in X are its “neighbors”



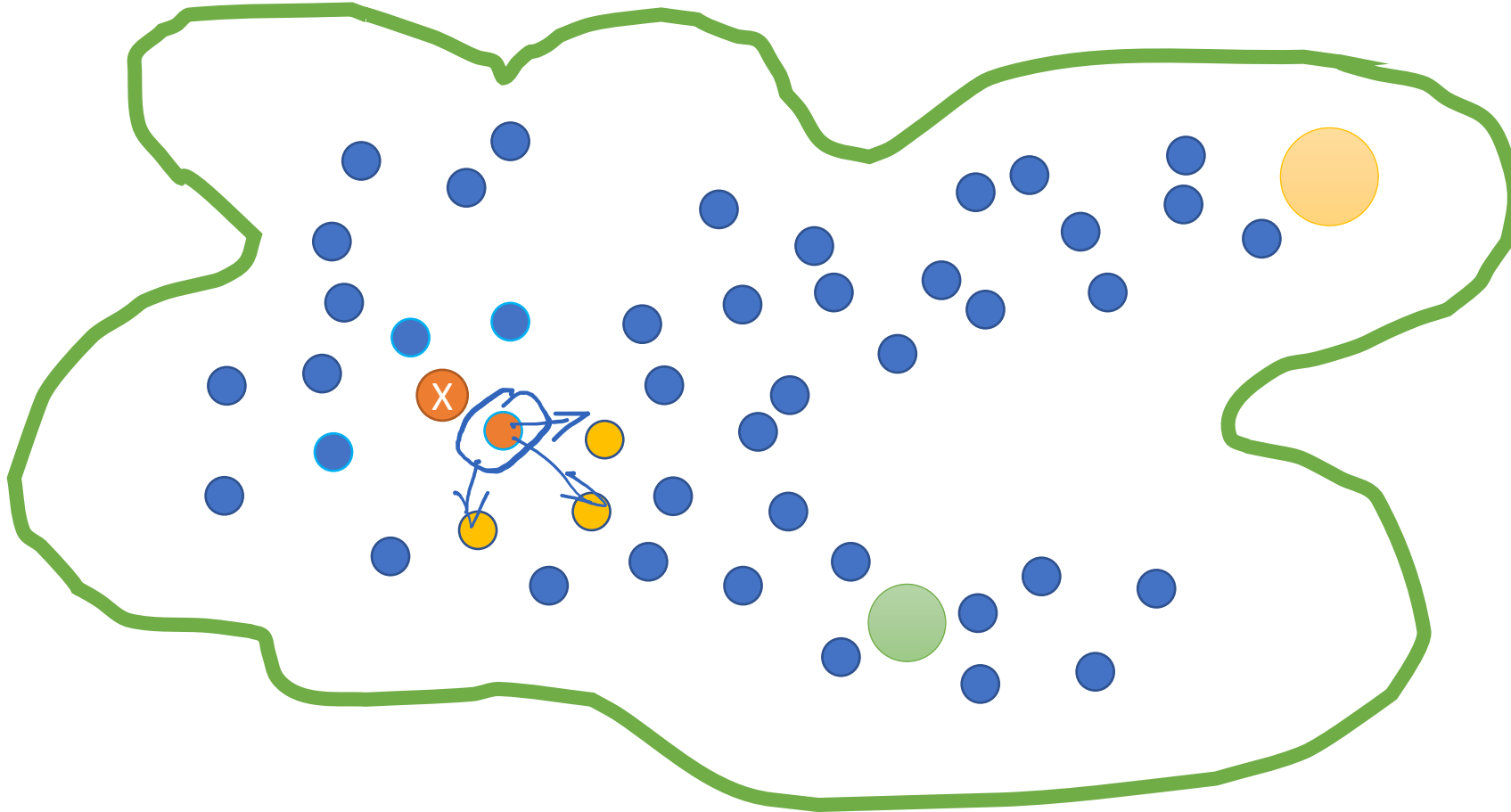
Local Search



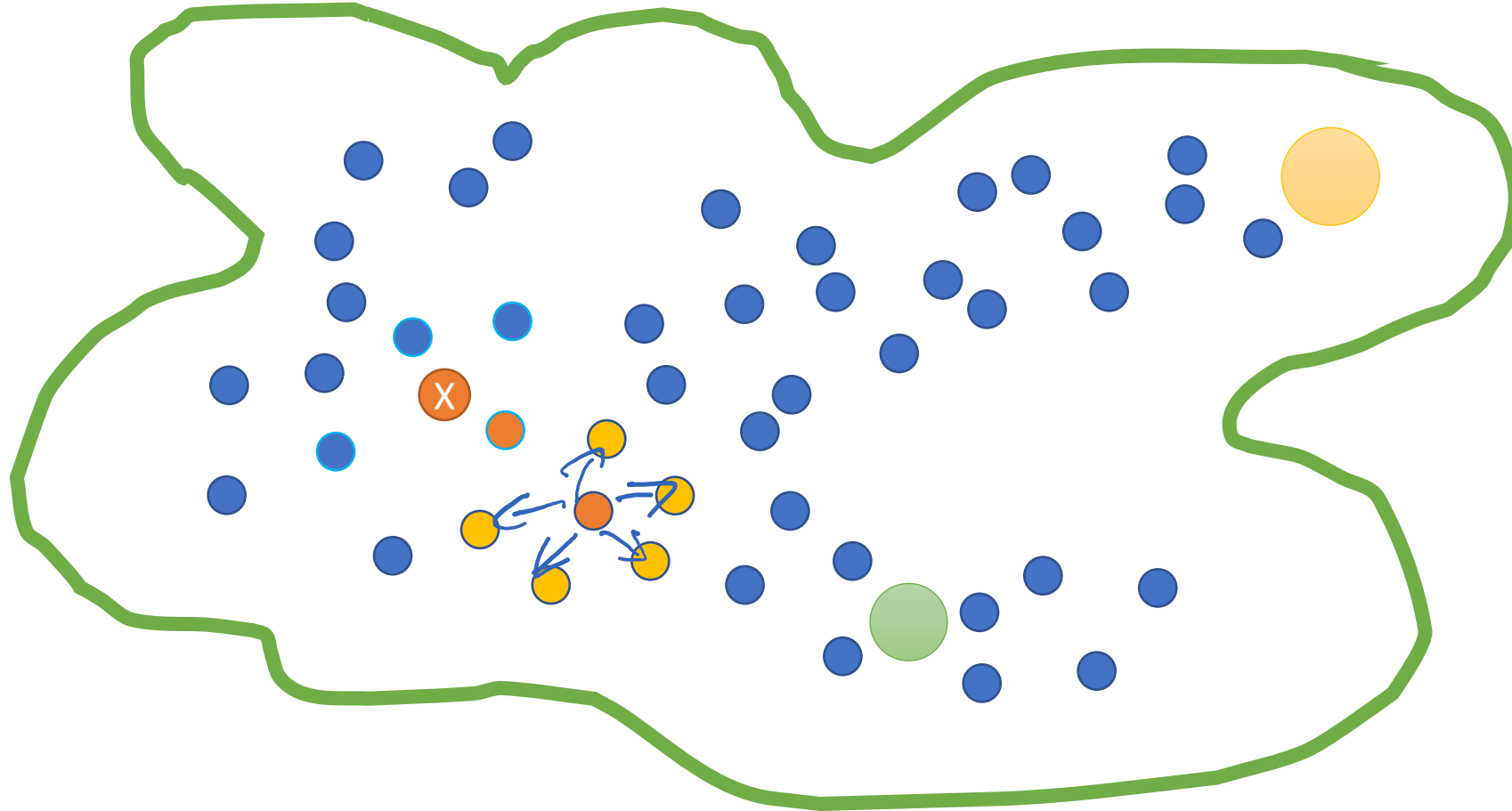
Local Search



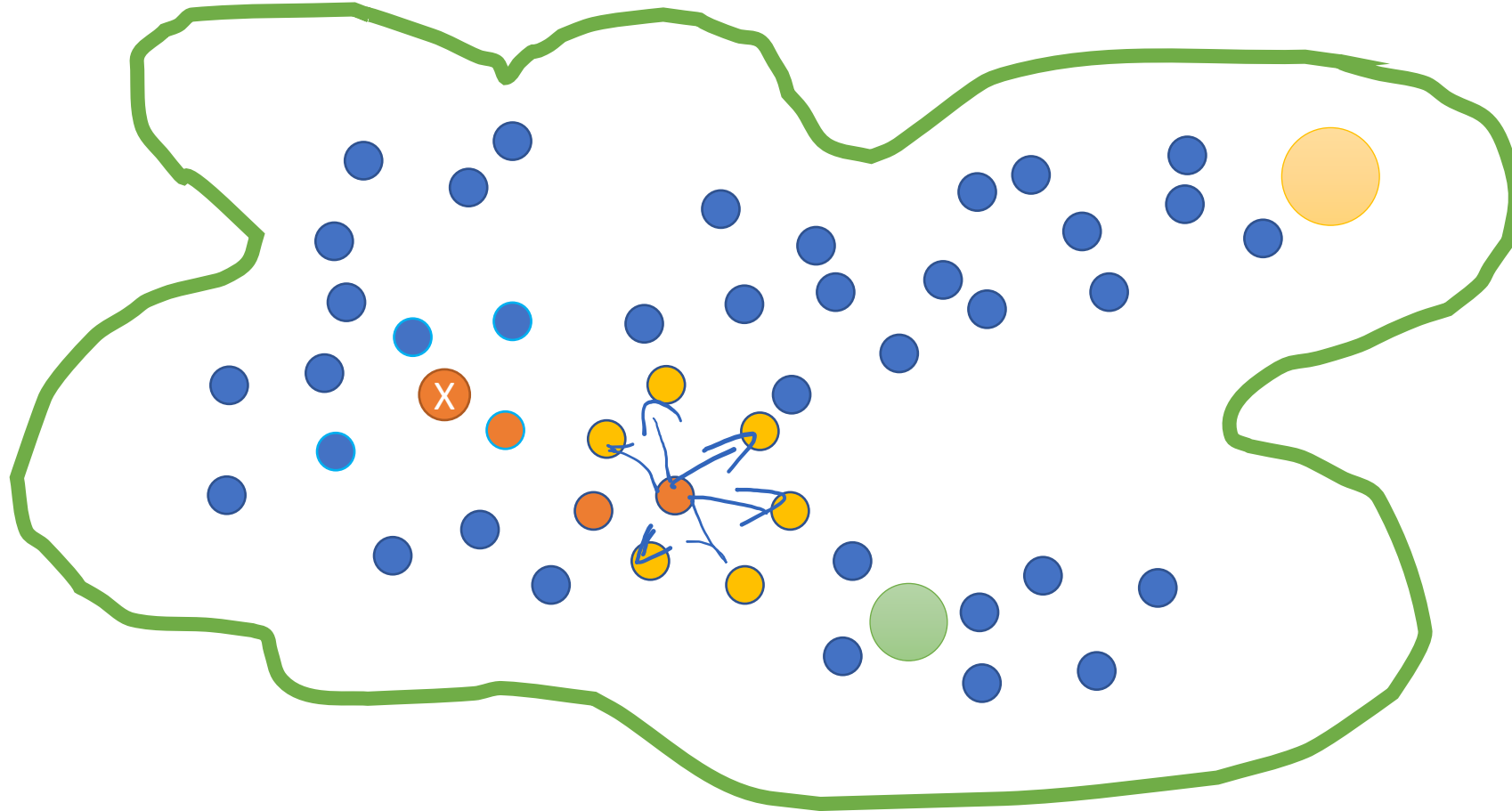
Local Search



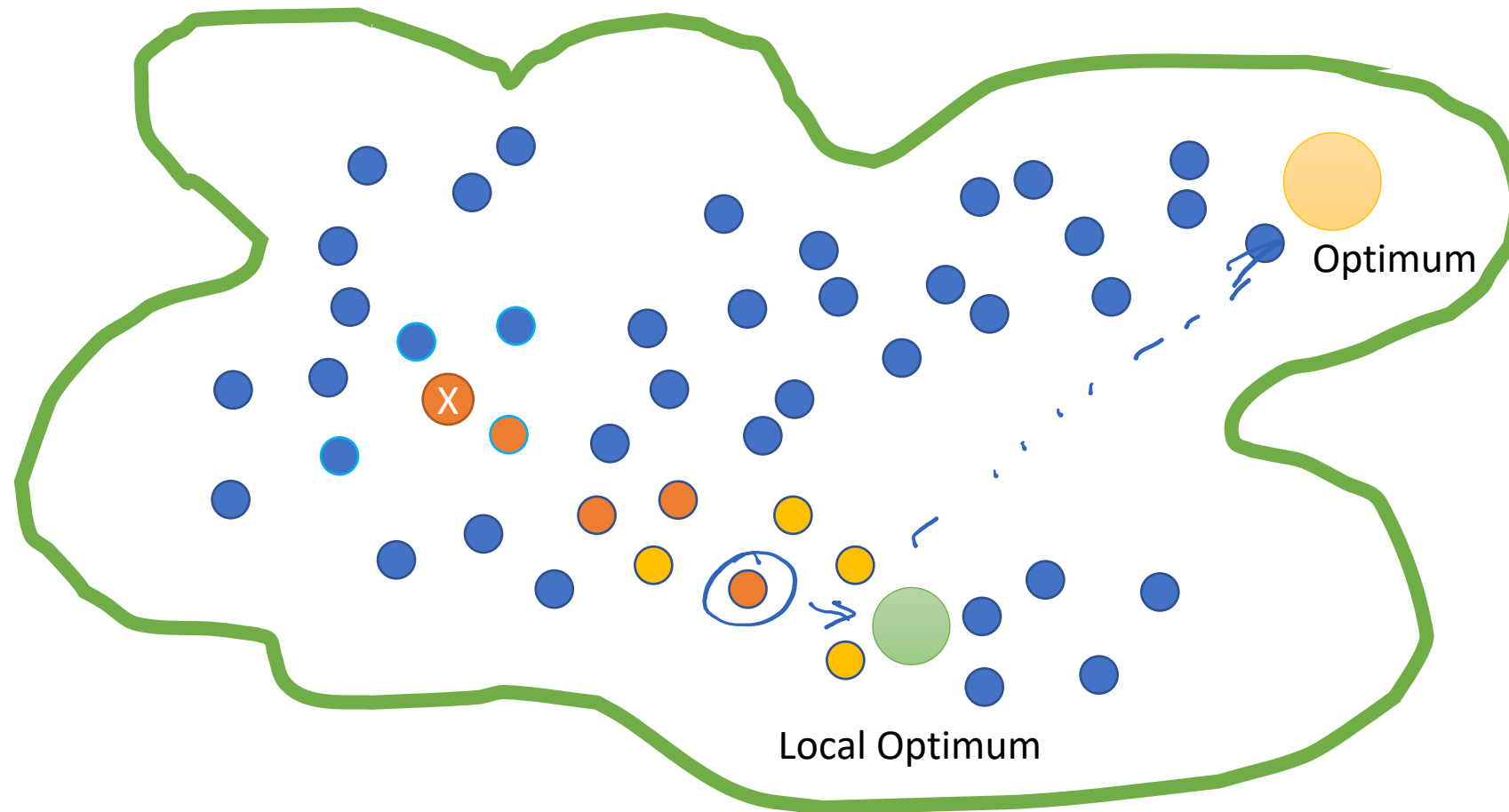
Local Search



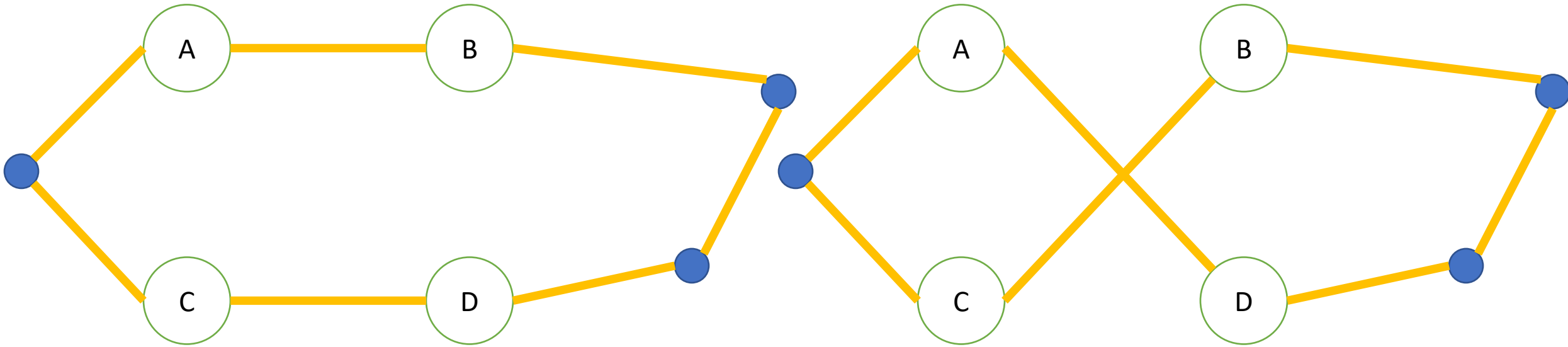
Local Search



Local Search



Neighborhood for TSP



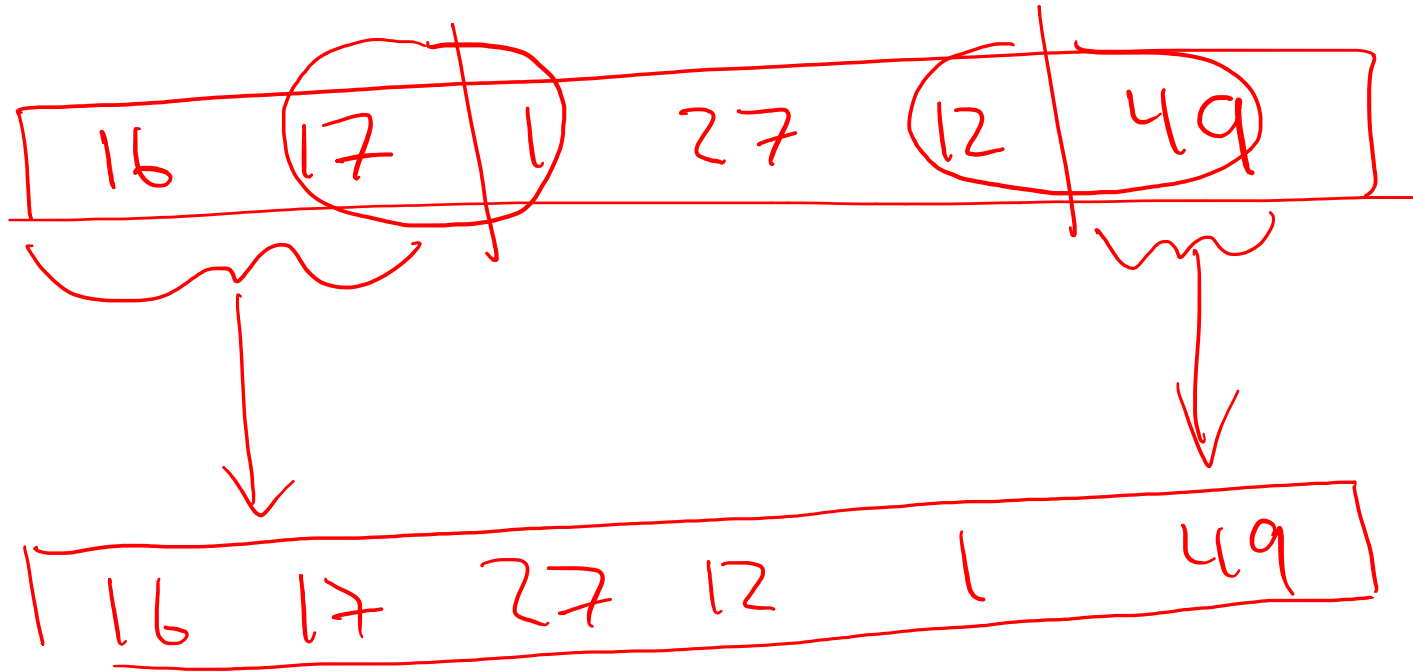
Let's say that two tours are neighbors if they differ by a minimal number of edges.

General Local Search Process

1. Let x = some initial solution
2. While the current solution, x , has a superior neighbor y , replace x with y
3. Return the final value for y (the locally optimal solution)

Simulated Annealing Demo

four



The Max-Cut Problem

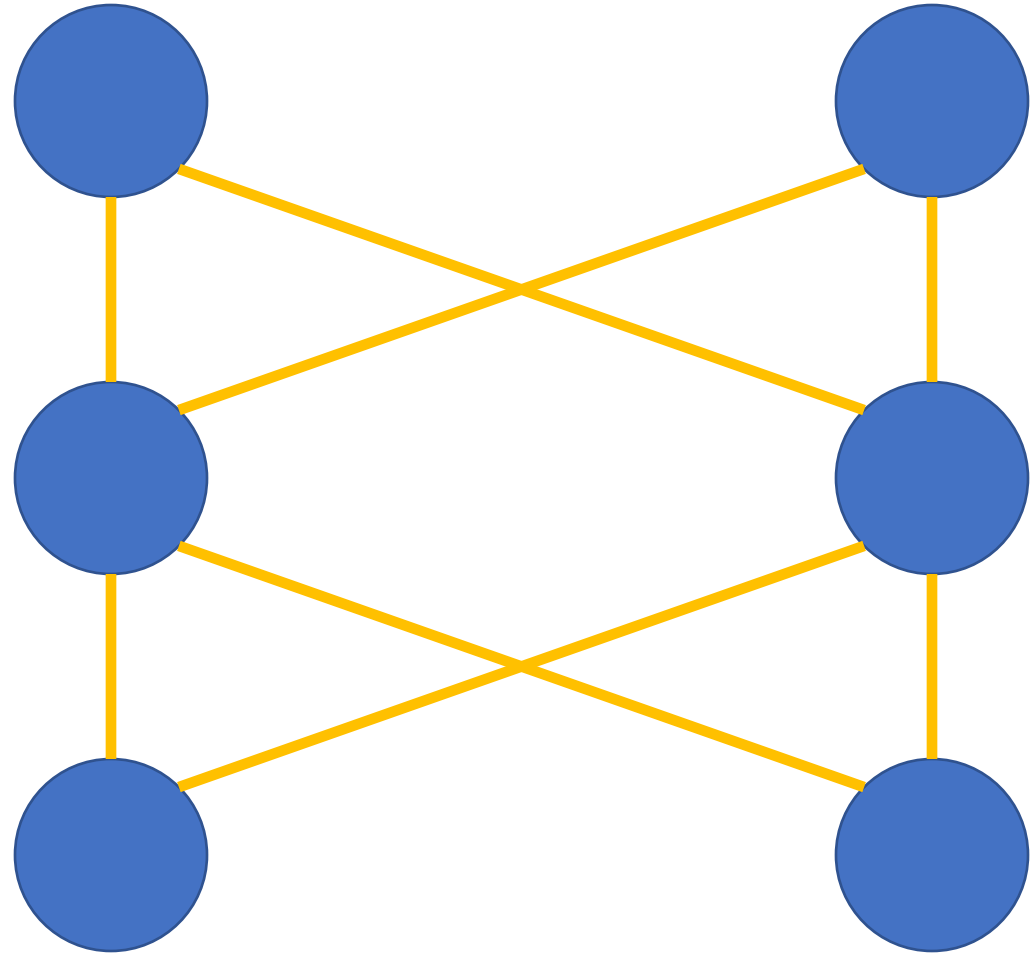
- Input: an undirected graph
- Output: a cut (A,B) that maximizes the number of crossing edges
- Reminder: a cut is a partition of the vertices into two non-empty sets
- How many possible cuts are there?

It turns out that:

- The min-cut problem is tractable (we have a polynomial time algorithm)
- The max-cut problem is NP-Complete

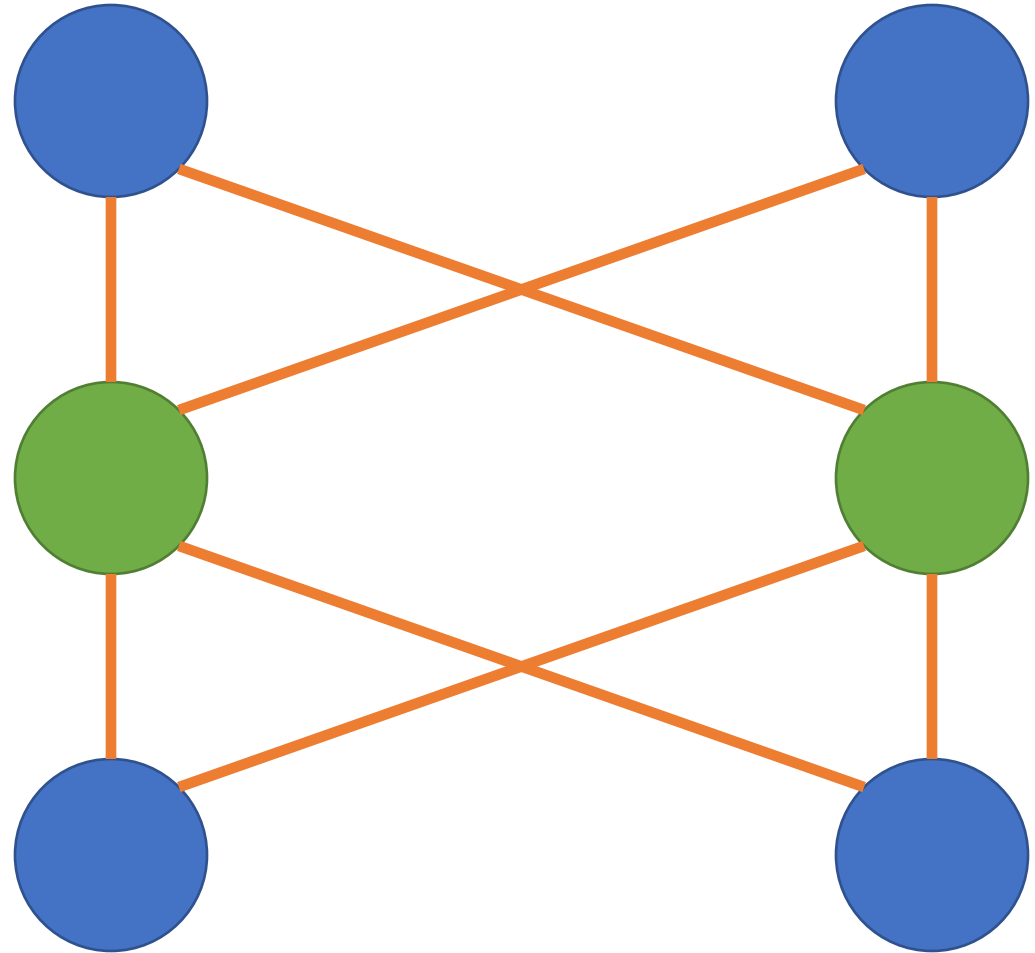
How many edges cross the max-cut?

- a. 4
- b. 6
- c. 8
- d. 10



How many edges cross the max-cut?

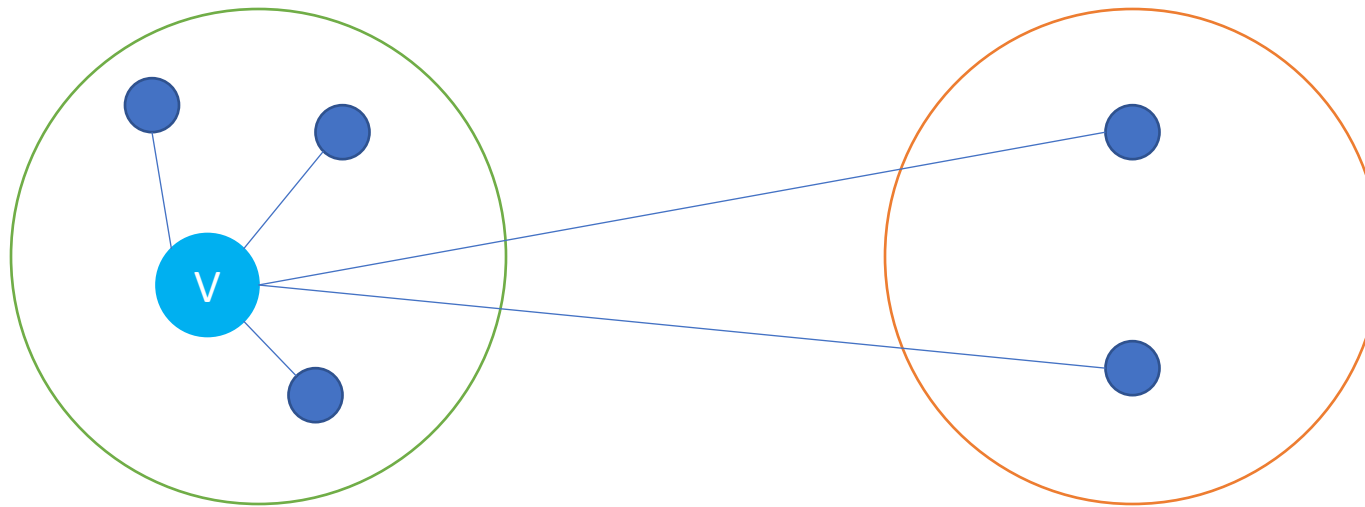
- a. 4
- b. 6
- c. 8
- d. 10



Local Search for Max-Cut

Notation: for a cut (A, B) and a vertex v :

- $C_v(A, B)$ = the number of edges incident on v that cross (A, B)
- $D_v(A, B)$ = the number of edges incident on v that don't cross (A, B)



$$C_v = 2$$
$$D_v = 3$$

Local Search for Max-Cut

1. Let (A,B) be some arbitrary cut of the graph G
2. While there is a vertex v with $D_v(A,B) > C_v(A,B)$
 1. move v to the other side of the cut
3. Return the final cut (A,B)

About this algorithm

- This algorithm runs in polynomial time (quadratic)
- This algorithm is **not** guaranteed to give the optimal cut
- This algorithm outputs a cut which is **at least** 50% of the maximum possible

About Local Search Algorithms

How do you pick the initial solution?

- Use a heuristic
- “this type of solution is usually a good place to start”
- Use a random choice

Which superior neighbor should you choose?

- Use a heuristic
- Choose the neighbor at random

- Choose the neighbor that yields the most improvement
- How do you define the neighborhood?

Can you think of some simple techniques for improving local search?

- Run the algorithm multiple times with some random choices!
- Independent trials.
- Combine **good** solutions.