Delivery Problem

Alexander S. Kulikov

Steklov Mathematical Institute at St. Petersburg, Russian Academy of Sciences
and
University of California, San Diego

Outline

Problem Statement

Brute Force Search

Nearest Neighbor

Branch and Bound

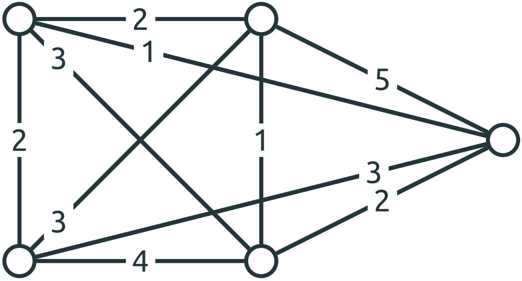
Dynamic Programming

Approximation Algorithm

Local Search

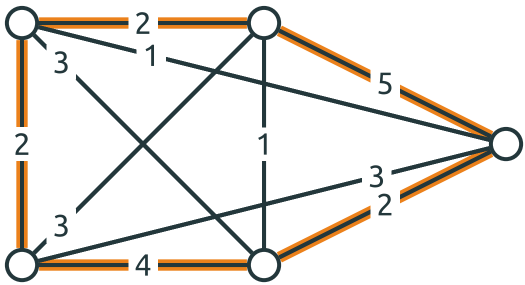
Traveling Salesman Problem

Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once



Traveling Salesman Problem

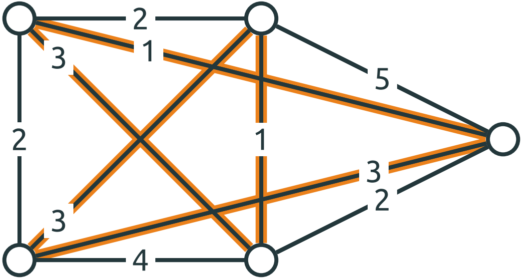
Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once



length: 15

Traveling Salesman Problem

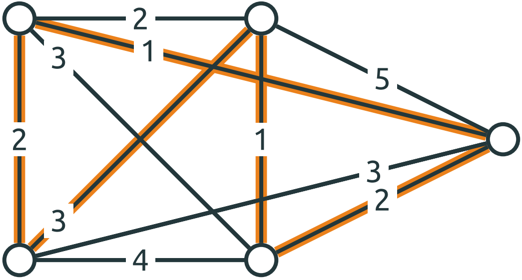
Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once



length: 11

Traveling Salesman Problem

Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once



length: 9

Status

- Classical optimization problem with countless number of real life applications (we'll see soon)

Status

- Classical optimization problem with countless number of real life applications (we’ll see soon)
- No polynomial time algorithms known

Status

- Classical optimization problem with countless number of real life applications (we'll see soon)
- No polynomial time algorithms known
- Goal of this project: develop efficient programs for solving TSP problem

Delivering Goods

Need to visit several points. What is the optimal order of visiting them?

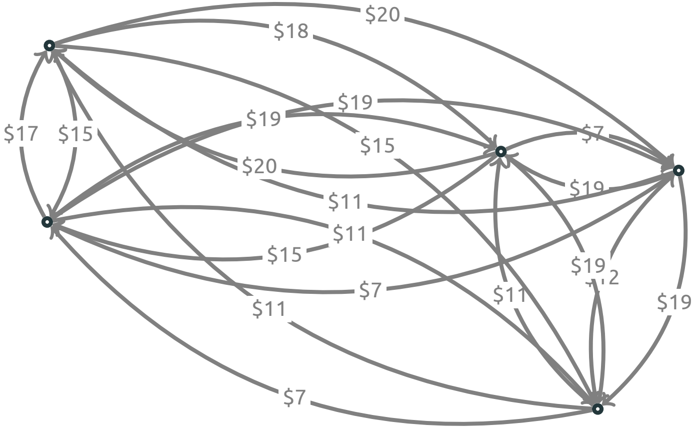
Traveling


United States
WASHINGTON
MONTANA
NORTH DAKOTA
MINNESOTA
SOUTH DAKOTA
WISCONSIN
MICHIGAN
OREGON
IDAHO
WYOMING
IOWA
NEBRASKA
NEVADA
UTAH
COLORADO
KANSAS
MISSOURI
ILLINOIS
INDIANA
OHIO
PENNSYLVANIA
NEW YORK
CALIFORNIA
ARIZONA
NEW MEXICO
OKLAHOMA
ARKANSAS
TENNESSEE
KENTUCKY
VIRGINIA
WEST VIRGINIA
NORTH CAROLINA
SOUTH CAROLINA
MISSISSIPPI
ALABAMA
GEORGIA
TEXAS
LOUISIANA
FLORIDA
MAINE
VERMONT
NEW HAMPSHIRE
MASSACHUSETTS
Seattle
Portland
Minneapolis
Chicago
Detroit
Toronto
Ottawa
Montreal
Quebec City
New York
Philadelphia
Denver
Kansas City
Indianapolis
Sacramento
San Francisco
San Jose
Las Vegas
Los Angeles
San Diego
Phoenix
Tucson
El Paso
Dallas
Austin
San Antonio
Houston
New Orleans
Nashville
Charlotte
Atlanta
Jacksonville
Orlando
Tampa
Miami
SONORA
CHIHUAHUA
Google

Traveling

Traveling

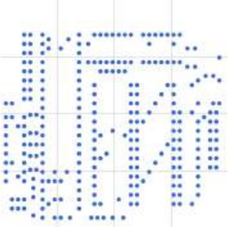
Traveling



Drilling a Circuit Board

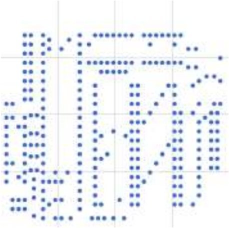
https://developers.google.com/optimization/routing/tsp/tsp

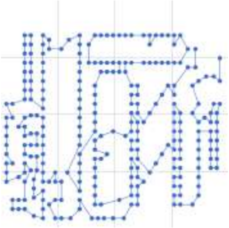
Drilling a Circuit Board



https://developers.google.com/optimization/routing/tsp/tsp

Drilling a Circuit Board





https://developers.google.com/optimization/routing/tsp/tsp

Euclidean TSP

- Euclidean TSP: instead of a complete graph, the input consists of n points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ on the plane

Euclidean TSP

- Euclidean TSP: instead of a complete graph, the input consists of n points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ on the plane
- Weights are given implicitly:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Euclidean TSP

- Euclidean TSP: instead of a complete graph, the input consists of n points $p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)$ on the plane
- Weights are given implicitly:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- Weights are symmetric: $d(p_i, p_j) = d(p_j, p_i)$

Euclidean TSP

- Euclidean TSP: instead of a complete graph, the input consists of n points $p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)$ on the plane
- Weights are given implicitly:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- Weights are symmetric: $d(p_i, p_j) = d(p_j, p_i)$
- Weights satisfy the triangle inequality: $d(p_i, p_j) \leq d(p_i, p_k) + d(p_k, p_j)$

Processing Components

There are n mechanical components to be processed on a complex machine. After processing the i-th component, it takes t_{ij} units of time to reconfigure the machine so that it is able to process the j-th component. What is the minimum processing cost?

Shortest Common Superstring

- The shortest common superstring problem (SCS): given a set $\{s_1, \ldots, s_n\}$ of n strings find a shortest string containing each s_i as a substring

Shortest Common Superstring

- The shortest common superstring problem (SCS): given a set $\{s_1, \dots, s_n\}$ of n strings find a shortest string containing each s_i as a substring
- Practical applications: data storage, data compression, genome assembly

Shortest Common Superstring

- The shortest common superstring problem (SCS): given a set $\{s_1, \ldots, s_n\}$ of n strings find a shortest string containing each s_i as a substring
- Practical applications: data storage, data compression, genome assembly
- At the first look, it is not at all clear how this problem is related to TSP

SCS: Example

- Consider the following instance:

  ABE, DFA, DAB, CBD, ECA, ACB

SCS: Example

- Consider the following instance:

  ABE, DFA, DAB, CBD, ECA, ACB

- To get a superstring, just concatenate them:

  ABEDFADABCBDECAACB

SCS: Example

- Consider the following instance:

  ABE, DFA, DAB, CBD, ECA, ACB

- To get a superstring, just concatenate them:

  ABEDFADABCBDECAACB

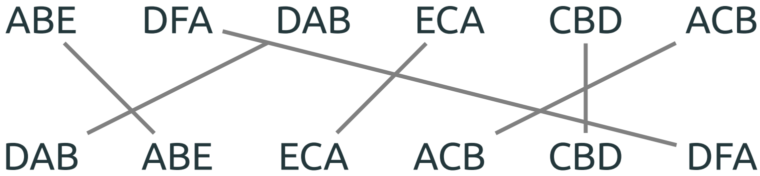
- But the strings ECA and ACB have a non-empty overlap. One can get a shorter superstring by overlapping them:

  ECACB

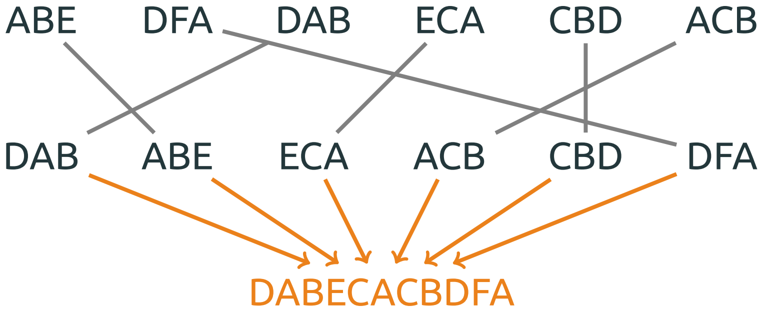
SCS: Permutation Problem

ABE DFA DAB ECA CBD ACB

SCS: Permutation Problem



SCS: Permutation Problem

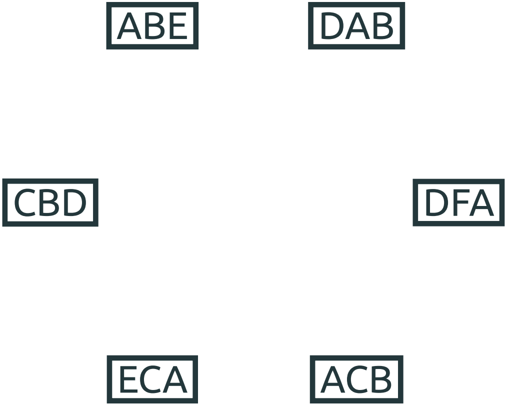


Overlap Graph: SCS$\rightarrow$MAX-ATSP

ABE DFA DAB CBD ECA ACB

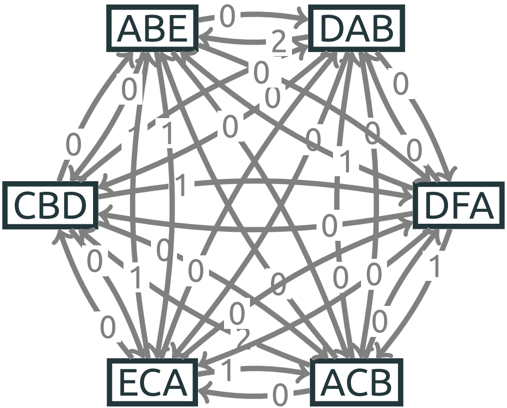
Overlap Graph: SCS→MAX-ATSP

ABE DFA DAB CBD ECA ACB



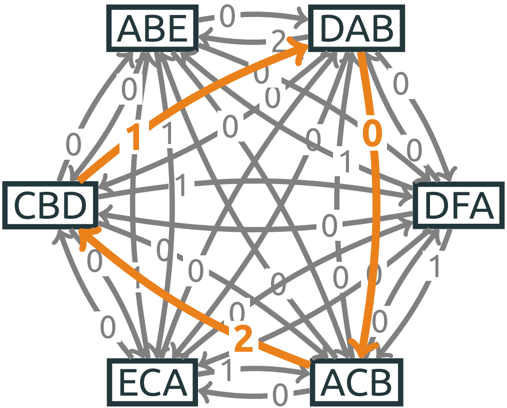
Overlap Graph: SCS→MAX-ATSP

ABE DFA DAB CBD ECA ACB



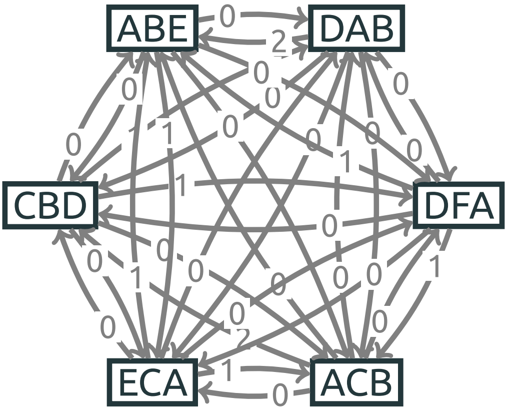
Overlap Graph: SCS→MAX-ATSP

ABE DFA DAB CBD ECA ACB



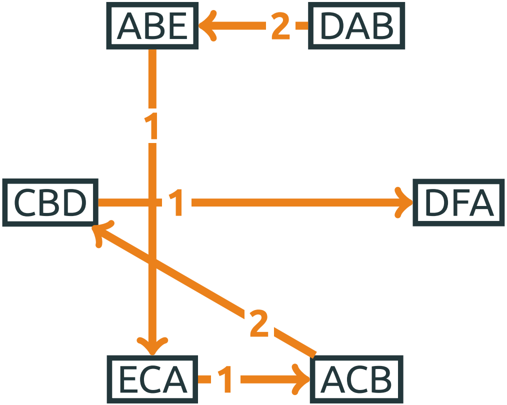
Overlap Graph: SCS→MAX-ATSP

ABE DFA DAB CBD ECA ACB



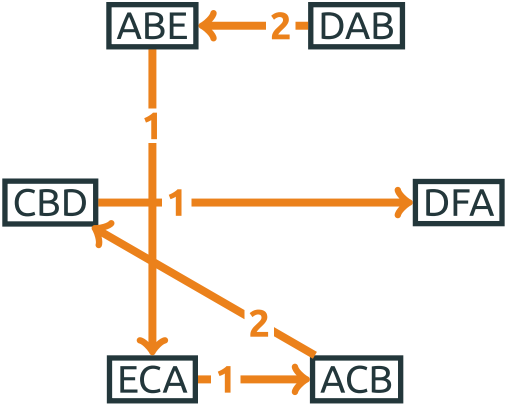
Overlap Graph: SCS→MAX-ATSP

ABE DFA DAB CBD ECA ACB



Overlap Graph: SCS→MAX-ATSP

ABE DFA DAB CBD ECA ACB



DABECACBDFA

Outline

Problem Statement

Brute Force Search

Nearest Neighbor

Branch and Bound

Dynamic Programming

Approximation Algorithm

Local Search

Enumerating all Permutations

- Finding the best permutation is easy: simply iterate through all of them and select the best one

Enumerating all Permutations

- Finding the best permutation is easy: simply iterate through all of them and select the best one
- But the number of permutations of n objects is $n!$

$n!$: Growth Rate

n	$n!$
5	120
8	40320
10	3628800
13	6227020800
20	2432902008176640000
30	265252859812191058636308480000000

Random Permutation

- OK, in most cases, we cannot afford going through all permutations

Random Permutation

- OK, in most cases, we cannot afford going through all permutations
- What if we just generate a random permutation?

Random Permutation

- OK, in most cases, we cannot afford going through all permutations
- What if we just generate a random permutation?
- The length of a random permutation may be much worse than the minimum length, even for Euclidean TSP

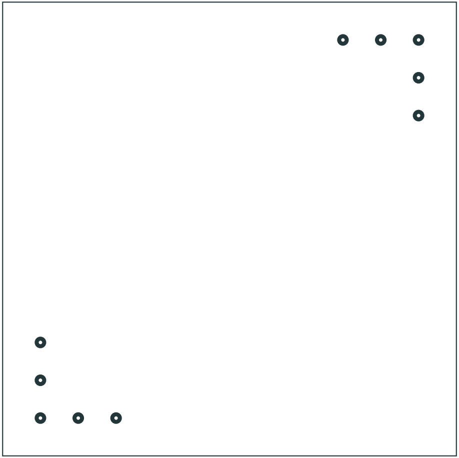
Expected Length

Lemma

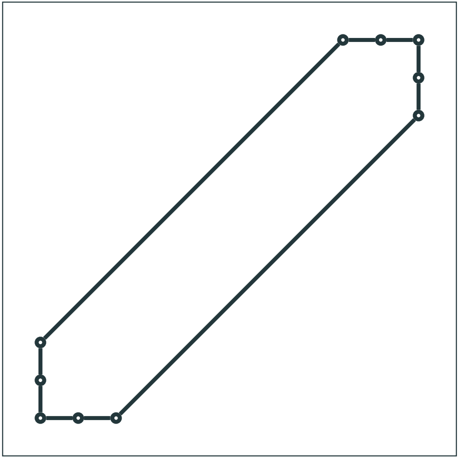
For a complete directed graph G, the expected length of a random permutation is

$$\frac{1}{n-1} \cdot \sum_{u,v \in V(G)} w(u,v)$$

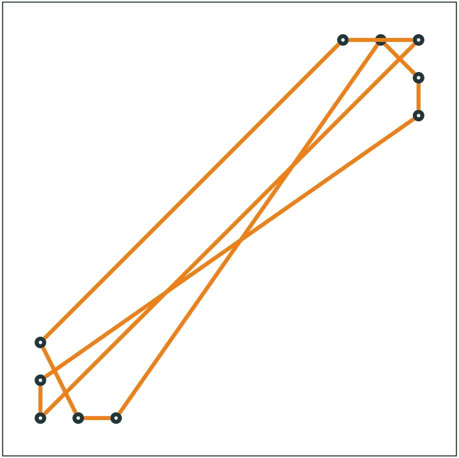
Bad Case



Bad Case



Bad Case



Outline

Speculating

- Sampling a random permutation is perhaps too naive

Speculating

- Sampling a random permutation is perhaps too naive
- What about going to the nearest yet unvisited node at every iteration?

Speculating

- Sampling a random permutation is perhaps too naive
- What about going to the nearest yet unvisited node at every iteration?
- Efficient, works reasonably well in practice

Speculating

- Sampling a random permutation is perhaps too naive
- What about going to the nearest yet unvisited node at every iteration?
- Efficient, works reasonably well in practice
- For general graphs, may produce a cycle that is much worse than an optimal one

Speculating

- Sampling a random permutation is perhaps too naive
- What about going to the nearest yet unvisited node at every iteration?
- Efficient, works reasonably well in practice
- For general graphs, may produce a cycle that is much worse than an optimal one
- For Euclidean instances, the resulting cycle may be about $\log n$ times worse than an optimal one

Nearest Neighbors: Bad Case

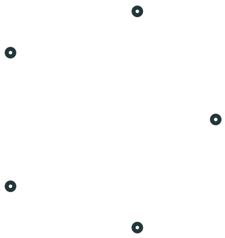
- How to fool the nearest neighbors heuristic?

Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2

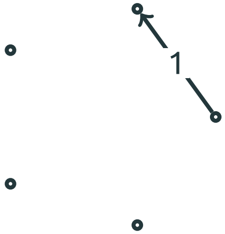
Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:



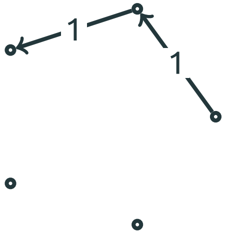
Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:



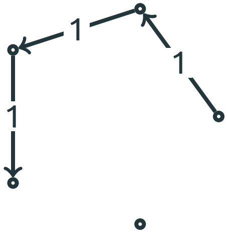
Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:



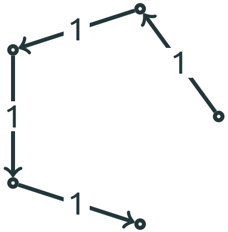
Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:



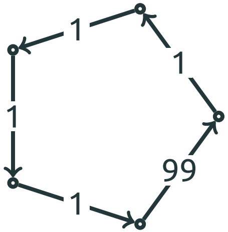
Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:

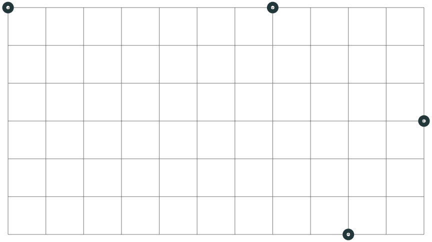


Nearest Neighbors: Bad Case

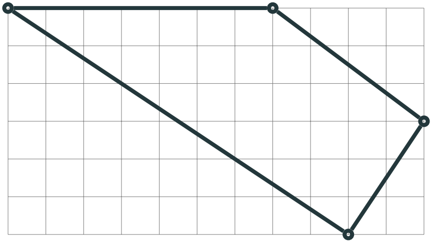
- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:



Suboptimal Solution for Euclidean TSP

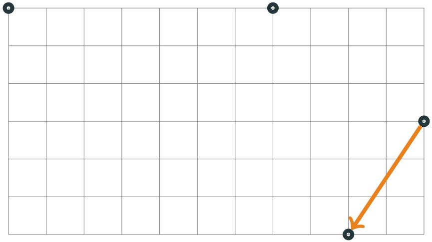


Suboptimal Solution for Euclidean TSP



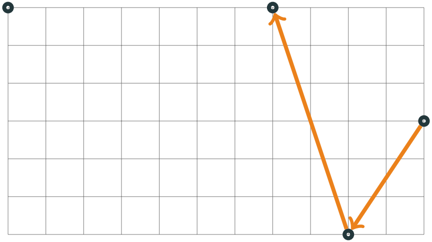
$$\mathrm{OPT} \approx 26.42$$

Suboptimal Solution for Euclidean TSP



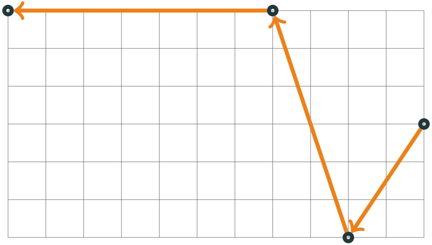
$$\mathrm{OPT} \approx 26.42$$

Suboptimal Solution for Euclidean TSP



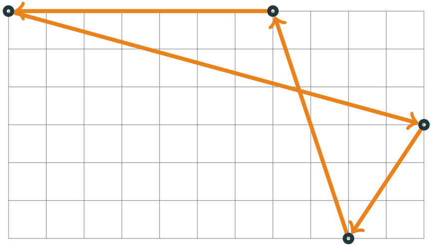
$\mathrm{OPT} \approx 26.42$

Suboptimal Solution for Euclidean TSP



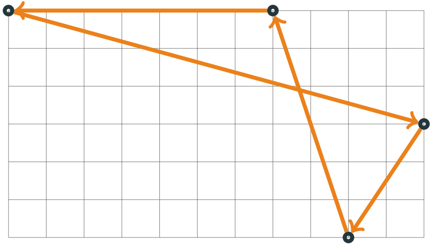
$$\text{OPT} \approx 26.42$$

Suboptimal Solution for Euclidean TSP



$$\text{OPT} \approx 26.42$$

Suboptimal Solution for Euclidean TSP



$$\text{OPT} \approx 26.42$$

$$\text{NN} \approx 28.33$$

Outline

Problem Statement

Brute Force Search

Nearest Neighbor

Branch and Bound

Dynamic Programming

Approximation Algorithm

Local Search

Main Ideas

- Start with some node

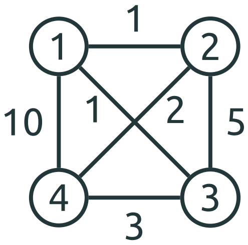
Main Ideas

- Start with some node
- At every iteration try to extend (recursively) the current path by every yet unvisited node

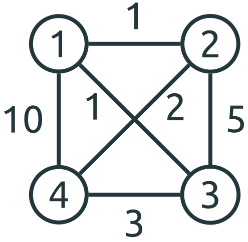
Main Ideas

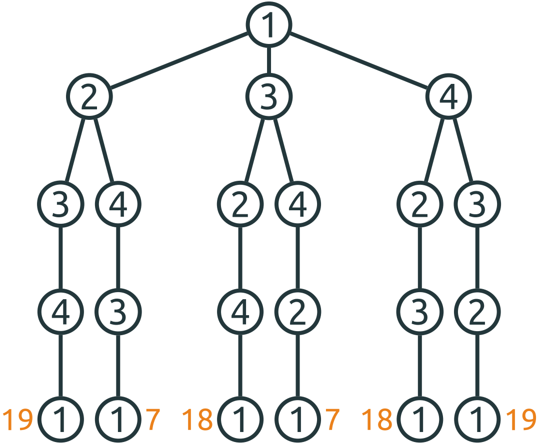
- Start with some node
- At every iteration try to extend (recursively) the current path by every yet unvisited node
- But don't continue extending the path, if it is already clear that it cannot be extended to an optimal cycle

Example: Brute Force Search

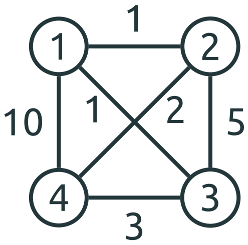


Example: Brute Force Search

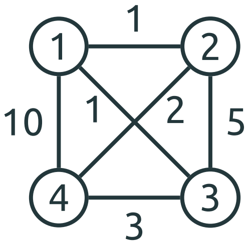


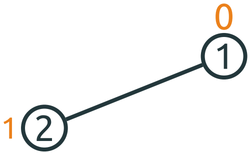


Example: Pruned Search

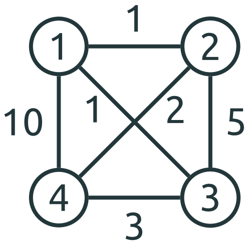


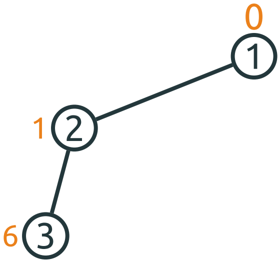
Example: Pruned Search



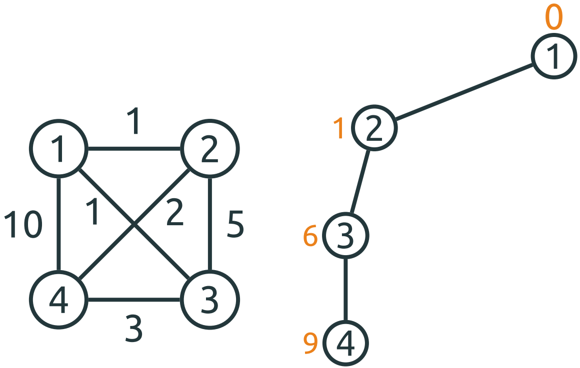


Example: Pruned Search

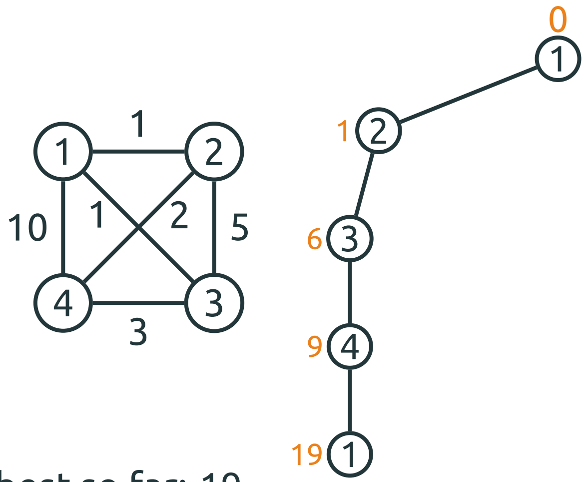




Example: Pruned Search

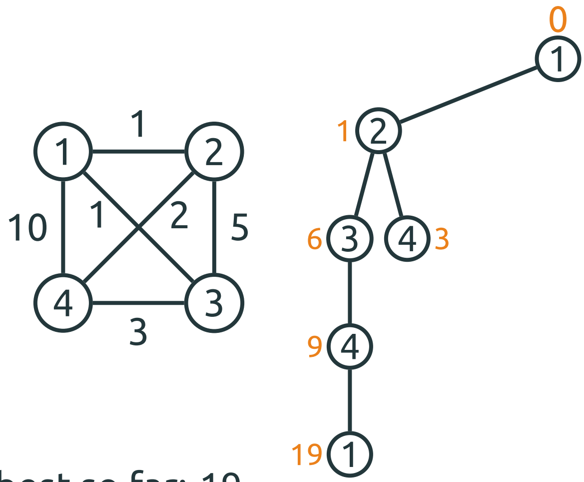


Example: Pruned Search



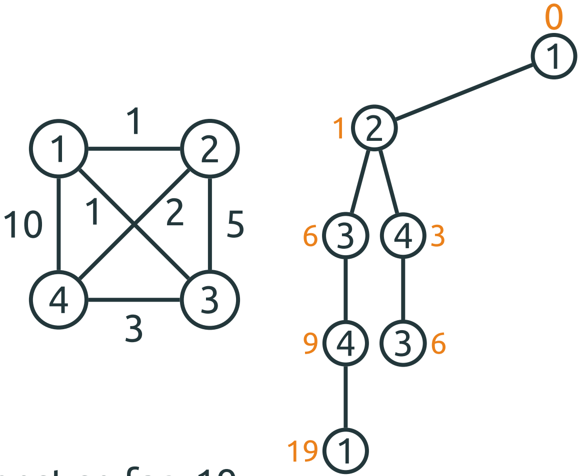
best so far: 19

Example: Pruned Search



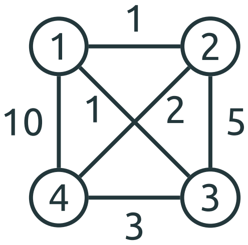
best so far: 19

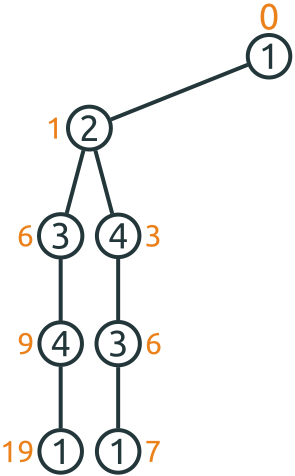
Example: Pruned Search



best so far: 19

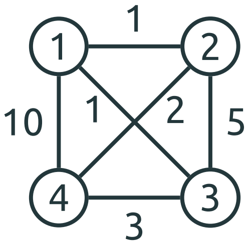
Example: Pruned Search

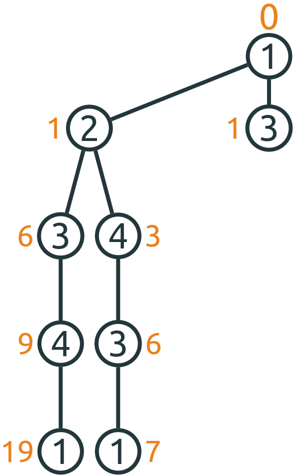




best so far: 7

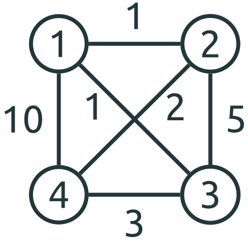
Example: Pruned Search

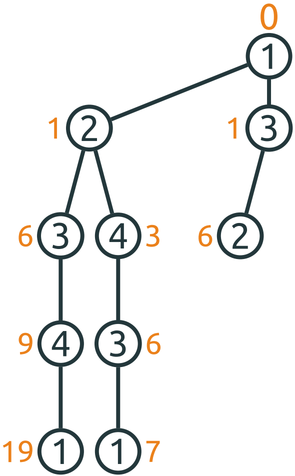




best so far: 7

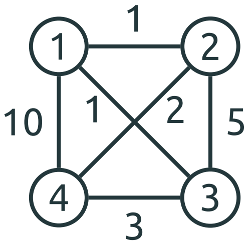
Example: Pruned Search

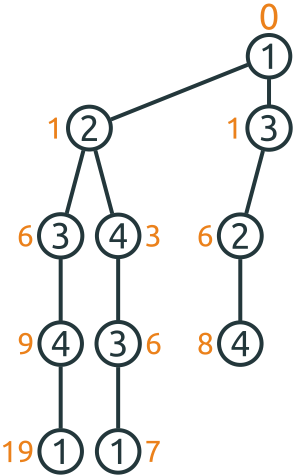




best so far: 7

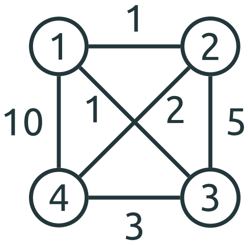
Example: Pruned Search

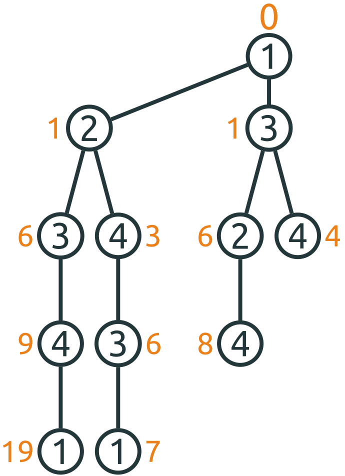




best so far: 7

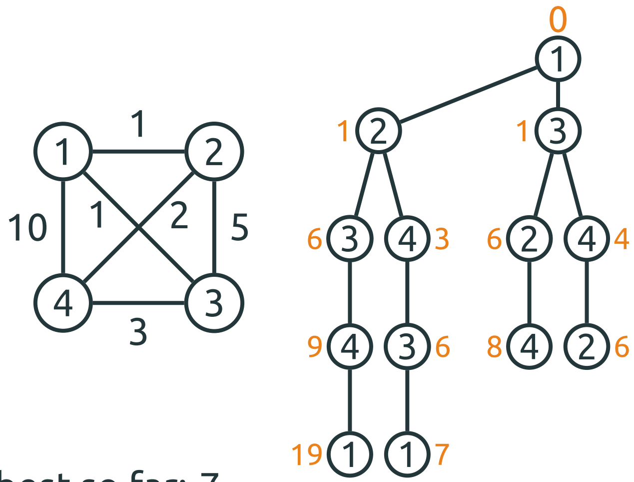
Example: Pruned Search





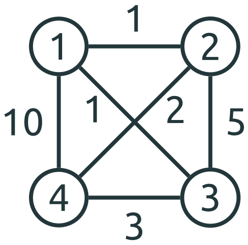
best so far: 7

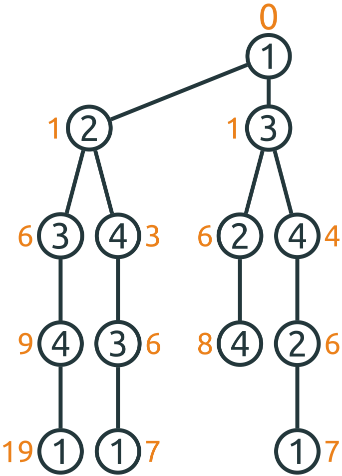
Example: Pruned Search



best so far: 7

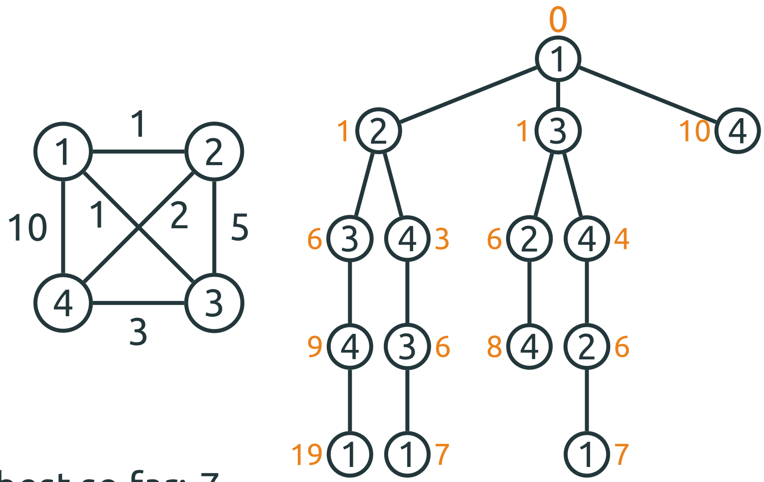
Example: Pruned Search





best so far: 7

Example: Pruned Search



best so far: 7

Lower Bound

- We used the simplest possible lower bound: any extension of a path has length at least the length of the path

Lower Bound

- We used the simplest possible lower bound: any extension of a path has length at least the length of the path
- Modern TSP-solvers use smarter lower bounds to solve instances with thousands of vertices

Example: Lower Bounds (Still Simple)

The length of an optimal TSP cycle is at least

- $\frac{1}{2} \sum_{v \in V}$(two min length edges adj to v)

Example: Lower Bounds (Still Simple)

The length of an optimal TSP cycle is at least

- $\frac{1}{2}\sum_{v \in V}$(two min length edges adj to v)
- the length of a minimum spanning tree (by taking out any edge of a TSP cycle, one gets a spanning tree)

Branch and Bound: Summary

- Main two heuristics of branch and bound:

Branch and Bound: Summary

- Main two heuristics of branch and bound:
    - Branch: order of yet unvisited nodes (say, select closer neighbors first)

Branch and Bound: Summary

- Main two heuristics of branch and bound:
  - Branch: order of yet unvisited nodes (say, select closer neighbors first)
  - Bound: lower bounding the length of a path

Branch and Bound: Summary

- Main two heuristics of branch and bound:
  - Branch: order of yet unvisited nodes (say, select closer neighbors first)
  - Bound: lower bounding the length of a path
- Finds an optimal solution

Branch and Bound: Summary

- Main two heuristics of branch and bound:
  - Branch: order of yet unvisited nodes (say, select closer neighbors first)
  - Bound: lower bounding the length of a path
- Finds an optimal solution
- The running time depends on the heuristics used as well as on the instance itself

Branch and Bound: Summary

- Main two heuristics of branch and bound:
    - Branch: order of yet unvisited nodes (say, select closer neighbors first)
    - Bound: lower bounding the length of a path
- Finds an optimal solution
- The running time depends on the heuristics used as well as on the instance itself
- Used by state-of-the-art TSP-solvers that can handle instances with thousands of nodes!

Outline

Problem Statement

Brute Force Search

Nearest Neighbor

Branch and Bound

Dynamic Programming

Approximation Algorithm

Local Search

Dynamic Programming

- Dynamic programming is one of the most powerful algorithmic techniques

Dynamic Programming

- Dynamic programming is one of the most powerful algorithmic techniques
- Rough idea: express a solution for a problem through solutions for smaller subproblems

Dynamic Programming

- Dynamic programming is one of the most powerful algorithmic techniques
- Rough idea: express a solution for a problem through solutions for smaller subproblems
- Solve subproblems one by one. Store solutions to subproblems in a table to avoid recomputing the same thing again

Subproblems

- For a subset of nodes $S \subseteq \{0, \ldots, n-1\}$ containing the node 0 and a node $i \in S$, let $C(i, S)$ be the length of the shortest path that starts at 0, ends at i, and visits all nodes from S exactly once

Subproblems

- For a subset of nodes $S \subseteq \{0, \ldots, n-1\}$ containing the node 0 and a node $i \in S$, let $C(i, S)$ be the length of the shortest path that starts at 0, ends at i, and visits all nodes from S exactly once
- $C(0, \{0\}) = 0$ and $C(0, S) = +\infty$ when $|S| > 1$

Recurrence Relation

- Consider the second-to-last node j on the required shortest path from 0 to i visiting all nodes from S

Recurrence Relation

- Consider the second-to-last node j on the required shortest path from 0 to i visiting all nodes from S
- The subpath from 0 to j is the shortest one visiting all vertices from $S - \{i\}$ exactly once

Recurrence Relation

- Consider the second-to-last node j on the required shortest path from 0 to i visiting all nodes from S
- The subpath from 0 to j is the shortest one visiting all vertices from $S - \{i\}$ exactly once
- Hence $C(i, S) = \min\{C(j, S - \{i\}) + w(j, i)\}$, where the minimum is over all $j \in S$ such that $j \neq i$

Implementation Remark

- How to iterate through all subsets of $\{0, \ldots, n-1\}$?

Implementation Remark

- How to iterate through all subsets of $\{0, \ldots, n-1\}$?
- There is a natural one-to-one correspondence between integers in the range from 0 to $2^n - 1$ and subsets of $\{0, \ldots, n-1\}$:

$$k \leftrightarrow \{i\colon\ i\text{-th bit of } k \text{ is } 1\}$$

Example

k	bin(k)	$\{i\colon\ i\text{-th bit of } k \text{ is } 1\}$
0	000	$\emptyset$
1	001	{0}
2	010	{1}
3	011	{0,1}
4	100	{2}
5	101	{0,2}
6	110	{1,2}
7	111	{0,1,2}

Flipping a Particular Bit

- If k corresponds to S, how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?

Flipping a Particular Bit

- If k corresponds to S, how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?
- For this, we need to flip the j-th bit of k (from 1 to 0)

Flipping a Particular Bit

- If k corresponds to S, how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?
- For this, we need to flip the j-th bit of k (from 1 to 0)
- For this, in turn, we compute a bitwise XOR of k and 2^j (that has 1 only in j-th position)

Flipping a Particular Bit

- If k corresponds to S, how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?
- For this, we need to flip the j-th bit of k (from 1 to 0)
- For this, in turn, we compute a bitwise XOR of k and 2^j (that has 1 only in j-th position)
- In `C/C++`, `Java`, `Python`:
  `k ^ (1 << j)`

Code

```
def dp(G):
  n = G.number_of_nodes()
  T = [[float("inf")] * (1 << n) for _ in range(n)]
  T[0][1] = 0
  for s in range(1 << n):
    if sum(((s >> j) & 1) for j in range(n)) <= 1 or not (s & 1):
      continue

    for i in range(1, n):
      if not ((s >> i) & 1):
        continue
      for j in range(n):
        if j == i or not ((s >> j) & 1):
          continue

        T[i][s] = min(T[i][s],
                      T[j][s ^ (1 << i)] + G[i][j]['weight'])

  return min(T[i][(1 << n) - 1] + G[0][i]['weight']
             for i in range(1, n))
```

Dynamic Programming: Summary

- The running time is about $n^2 2^n$

Dynamic Programming: Summary

- The running time is about $n^2 2^n$
- Better than $n!$, but still too slow (already for $n = 20$)

Outline

Problem Statement

Brute Force Search

Nearest Neighbor

Branch and Bound

Dynamic Programming

Approximation Algorithm

Local Search

Approximation

- Let's focus on the metric version of TSP: $w(u, v) = w(v, u)$ and $w(u, v) \leq w(u, z) + w(z, v)$ (in particular, Euclidean TSP is metric)
- We will design a 2-approximation algorithm: it quickly finds a cycle that is at most twice longer than an optimal one

Minimum Spanning Trees

Lemma

Let G be an undirected graph with non-negative edge weights. Then $\mathrm{MST}(G) \leq \mathrm{TSP}(G)$.

Minimum Spanning Trees

Lemma

Let G be an undirected graph with non-negative edge weights. Then $\mathrm{MST}(G) \leq \mathrm{TSP}(G)$.

Proof

By removing any edge from an optimum TSP cycle one gets a spanning tree of G. $\square$

Algorithm

- $T \leftarrow$ minimum spanning tree of G

Algorithm

- $T \leftarrow$ minimum spanning tree of G
- $D \leftarrow T$ with each edge doubled

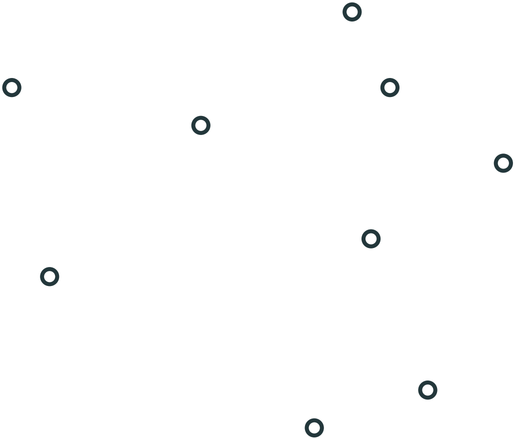
Algorithm

- $T \leftarrow$ minimum spanning tree of G
- $D \leftarrow T$ with each edge doubled
- find an Eulerian cycle C in D

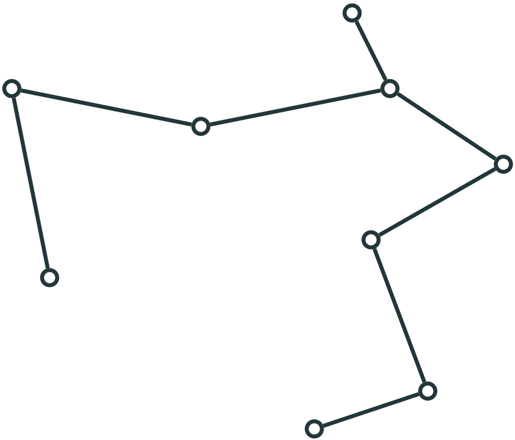
Algorithm

- $T \leftarrow$ minimum spanning tree of G
- $D \leftarrow T$ with each edge doubled
- find an Eulerian cycle C in D
- return a cycle that visits the nodes in the order of their first appearance in C

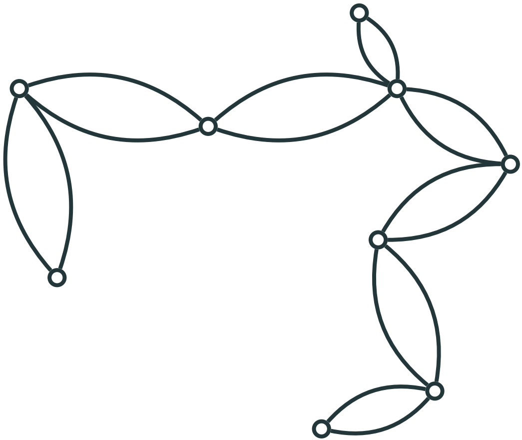
Example



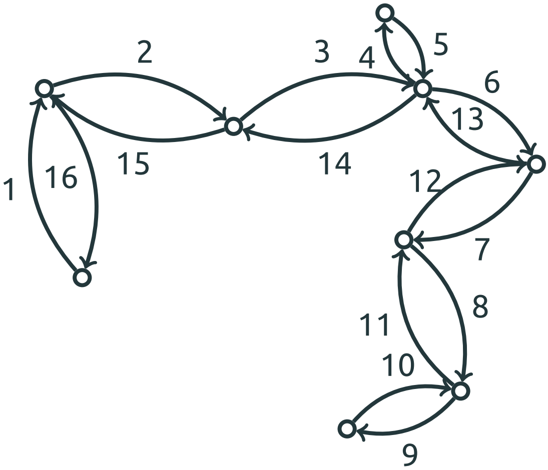
Example



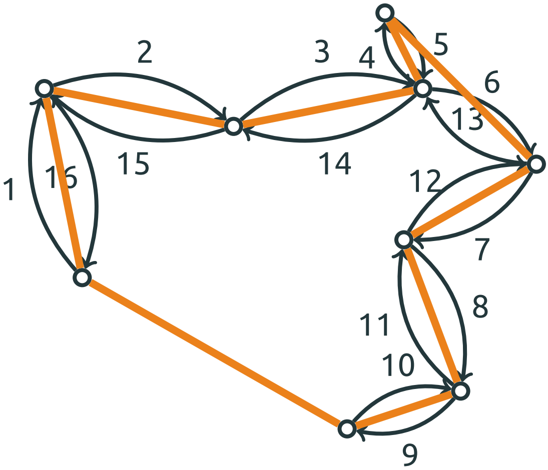
Example



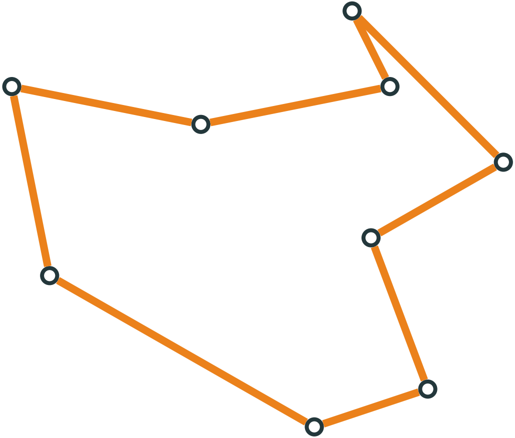
Example



Example



Example



Provable Guarantee

Lemma

The algorithm is 2-approximate.

Provable Guarantee

Lemma

The algorithm is 2-approximate.

Proof

- The total length of the MST T is at most OPT.

Provable Guarantee

Lemma

The algorithm is 2-approximate.

Proof

- The total length of the MST T is at most OPT.
- Bypasses can only decrease the total length. $\square$

Final Remarks

- The currently best known approximation algorithm for metric TSP is Christofides' algorithm that achieves a factor of 1.5

Final Remarks

- The currently best known approximation algorithm for metric TSP is Christofides' algorithm that achieves a factor of 1.5
- If P $\neq$ NP, then there is no α-approximation algorithm for the general version of TSP for any constant α

Outline

Problem Statement

Brute Force Search

Nearest Neighbor

Branch and Bound

Dynamic Programming

Approximation Algorithm

Local Search

Local Search

Local Search with parameter d:

- $s \leftarrow$ some initial solution

Local Search

Local Search with parameter d:

- $s \leftarrow$ some initial solution
- while it is possible to change d edges in s to get a better cycle s':

Local Search

Local Search with parameter d:

- $s \leftarrow$ some initial solution
- while it is possible to change d edges in s to get a better cycle s':
  - $s \leftarrow s'$

Local Search

Local Search with parameter d:

- $s \leftarrow$ some initial solution
- while it is possible to change d edges in s to get a better cycle s':
  - $s \leftarrow s'$
- return s

Properties

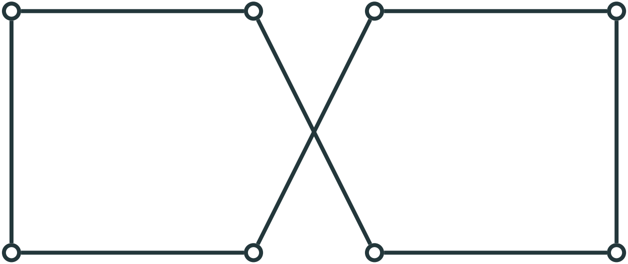
- Computes a local optimum instead of a global optimum

Properties

- Computes a local optimum instead of a global optimum
- The larger is d, the better is the resulting solution and the higher is the running time

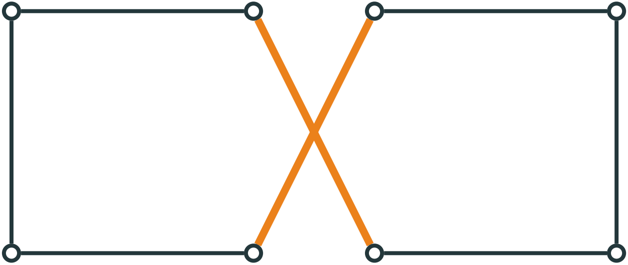
Example

Changing two edges in a suboptimal solution:



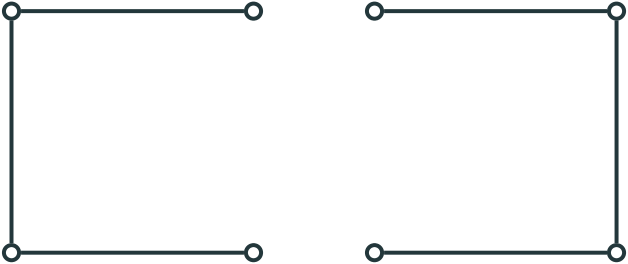
Example

Changing two edges in a suboptimal solution:



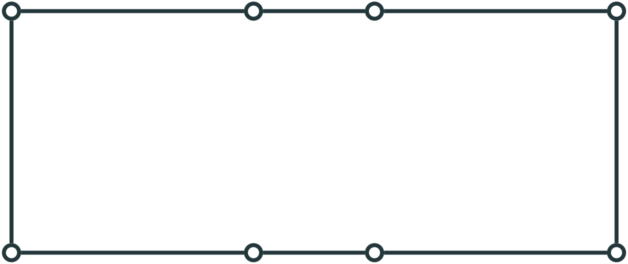
Example

Changing two edges in a suboptimal solution:



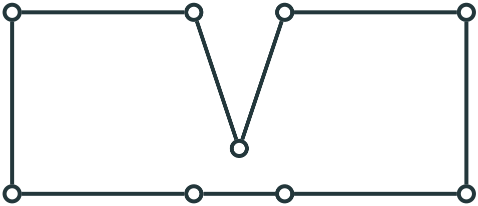
Example

Changing two edges in a suboptimal solution:



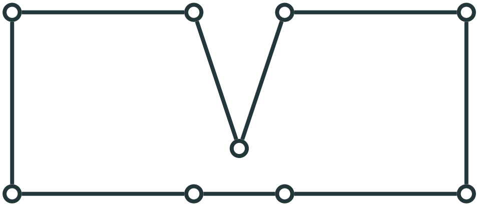
Example

A suboptimal solution that cannot be improved by changing two edges:



Example

A suboptimal solution that cannot be improved by changing two edges:



Need to allow changing three edges to improve this solution

Performance

- Trade-off between quality and running time of a single iteration

Performance

- Trade-off between quality and running time of a single iteration
- Still, the number of iterations may be exponential and the quality of the found cycle may be poor

Performance

- Trade-off between quality and running time of a single iteration
- Still, the number of iterations may be exponential and the quality of the found cycle may be poor
- But works well in practice

Summary

- Exact algorithms: brute force, branch and bound, dynamic programming

Summary

- Exact algorithms: brute force, branch and bound, dynamic programming
- Approximation algorithms: nearest neighbors, MST-based, local search