Algorithmics and Complexity
Lecture 7/7 : Approaches for Hard Problems

CentraleSupélec – Gif

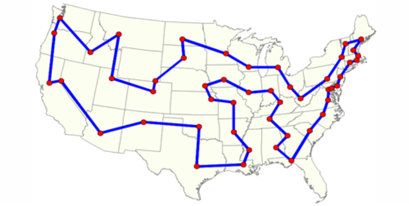
ST2 – Gif

Plan

Problem

Concrete problem

- Consider a set of cities and the distances between them, what is the shortest possible route that visits each city once and returns to the departure city?



This is the Travelling Salesman Problem (TSP) (In french *le problème du voyageur de commerce*).

Traveling Salesman Problem (TSP)

Optimization Problem

- Instance :
  - $G = (V, E)$ a complete and undirected graph with $|V| = n$
  - $d : E \rightarrow \mathbb{R}$ a weight function that associates a distance to each edge

Traveling Salesman Problem (TSP)

Optimization Problem

- Instance :
  - $G = (V, E)$ a complete and undirected graph with $|V| = n$
  - $d : E \rightarrow \mathbb{R}$ a weight function that associates a distance to each edge

- Question :
  - Find $S = [s_1, ..., s_n]$ a list of elements of V, such that

Traveling Salesman Problem (TSP)

Optimization Problem

- Instance :
  - $G = (V, E)$ a complete and undirected graph with $|V| = n$
  - $d : E \to \mathbb{R}$ a weight function that associates a distance to each edge
- Question :
  - Find $S = [s_1, ..., s_n]$ a list of elements of V, such that
- Constraints:
  - Each element of V appears exactly once in S
  - We **minimize** $Score(S) = \sum_{s_i \in S} d(s_i, s_{i+1})$

(we set $s_{n+1} = s_1$ to simplify the notations)

What to do at this stage?

1. Browse a catalog of known problems to learn about existing results
   - example: the *compendium* of Viggo Kann

What to do at this stage?

1. Browse a catalog of known problems to learn about existing results
   - example: the *compendium* of Viggo Kann
2. Suppose (which is false) that this problem does not exist in the literature, we should study it starting by this question:
   ➜ Is it in *NP*?

TSP as a decision problem

TSP as a decision problem

Decision Problem

- Instance :
  - $G = (V, E)$ a complete undirected graph with $|V| = n$
  - $d : E \to \mathbb{R}$ a weight function that associates a distance to each edge
  - $B \in \mathbb{R}$ an upper bound
- Question :
  - is there $S = [s_1, ..., s_n]$ a list of elements of V, such that
- Constraints:
  - Each element of V appears exactly once in S
  - $\sum_{s_i \in S} d(s_i, s_{i+1}) \leq B$

TSP as a decision problem

TSP is in *NP*?

The algorithm verifying a solution S of some positive instance (V, E) of the problem should check the two constraints of the problem:

- Each element of V appears exactly once in S
- $\sum_{s_i \in S} d(s_i, s_{i+1}) \leq B$

TSP as a decision problem

TSP is in *NP*?

The algorithm verifying a solution S of some positive instance (V, E) of the problem should check the two constraints of the problem:

- Each element of V appears exactly once in S
  - ➜ can be done in $\mathcal{O}(n)$ ($n = |V|$)
- $\sum_{s_i \in S} d(s_i, s_{i+1}) \leq B$
  - ➜ can be done in $\mathcal{O}(n)$ (using an adjacency matrix)

TSP as a decision problem

TSP is in *NP*?

The algorithm verifying a solution S of some positive instance (V, E) of the problem should check the two constraints of the problem:

- Each element of V appears exactly once in S
  - ➜ can be done in $\mathcal{O}(n)$ $(n = |V|)$
- $\sum_{s_i \in S} d(s_i, s_{i+1}) \leq B$
  - ➜ can be done in $\mathcal{O}(n)$ (using an adjacency matrix)

Thus we have a polynomial algorithm to check a solution.

Conclusion

➜ The Traveling Salesman Problem is indeed in *NP*.

TSP as a decision problem

Is TSP *NP*-complete?

We know that the problem is *NP*, now we should either:

- Find a polynomial solving algorithm;
- or Show that the problem is *NP*-complete.

TSP as a decision problem

Is TSP *NP*-complete?

We know that the problem is *NP*, now we should either:

- Find a polynomial solving algorithm;
- or **Show that the problem is *NP*-complete**.
  ...by performing a polynomial reduction from a known problem

List of problems already addressed

- Clique
- Stable
- **HAM/D-HAM**
- SAT
- ...

Hamiltonian cycle

Hamiltonian cycle problem — HAM

Instance :

- $G = (V, E)$ a undirected graph with $|V| = n$

Question :

- Is there $S = [s_1, ..., s_n]$ an ordered list of element of V, such that

Constraints :

- Each element of V occurs exactly once in S
- $\forall s_i \in S, \{s_i, s_{i+1}\} \in E$

Hamiltonian cycle

Hamiltonian cycle problem — HAM

Instance :

- $G = (V, E)$ a undirected graph with $|V| = n$

Question :

- Is there $S = [s_1, ..., s_n]$ an ordered list of element of V, such that

Constraints :

- Each element of V occurs exactly once in S
- $\forall s_i \in S, \{s_i, s_{i+1}\} \in E$

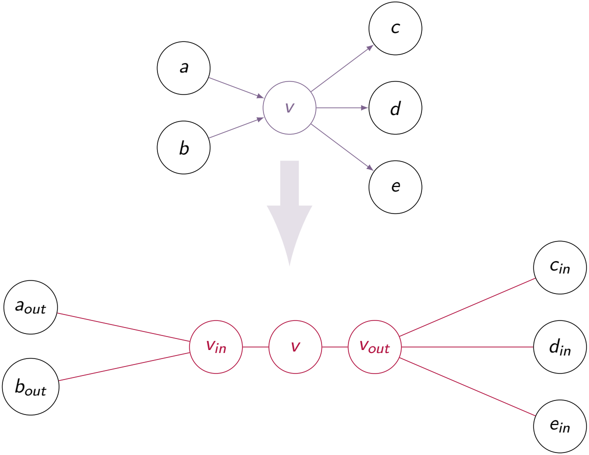
D-HAM $\leq$ HAM

D-HAM seen last lecture reduced to HAM

▸▸ skip reduction

D-HAM $\leq$ HAM

- A common reduction from a directed graph to a undirected one:



Reduction from the Hamiltonian cycle problem

Reduction from the Hamiltonian cycle problem

Can we reduce the Hamiltonian Cycle problem to the Traveling Salesman problem?

Reduction from the Hamiltonian cycle problem

Can we reduce the Hamiltonian Cycle problem to the Traveling Salesman problem?

We want to show that :

Hamiltonian cycle $\leq$ Traveling Salesman
HAM $\leq$ TSP

Reduction from the Hamiltonian cycle problem

Can we reduce the Hamiltonian Cycle problem to the Traveling Salesman problem?

We want to show that :

$$\text{Hamiltonian cycle} \leq \text{Traveling Salesman}$$
$$\text{HAM} \leq \text{TSP}$$

We present a polynomial reduction of each instance of the Hamiltonian Cycle to an instance of the Traveling Salesman problem

Polynomial Reduction

1/2

Reduction

Let $G = (V, E)$ be an instance of HAM, we construct $\langle G' = (V', E'), d, B \rangle$ with:

- $V' =$
- $E' =$

Polynomial Reduction

Reduction

Let $G = (V, E)$ be an instance of HAM, we construct $\langle G' = (V', E'), d, B\rangle$ with:

- $V' = V$
  We keep the same vertices...
- $E' =$

Polynomial Reduction 1/2

Reduction

Let $G = (V, E)$ be an instance of HAM, we construct $\langle G' = (V', E'), d, B \rangle$ with:

- $V' = V$
  We keep the same vertices...
- $E' = \{\{u, v\}, \forall u, v \in V \wedge u \neq v\}$
  ...but we construct a complete graph G'!

Polynomial Reduction 1/2

Reduction

Let $G = (V, E)$ be an instance of HAM, we construct $\langle G' = (V', E'), d, B \rangle$ with:

- $V' = V$
  We keep the same vertices...
- $E' = \{\{u, v\}, \forall u, v \in V \wedge u \neq v\}$
  ...but we construct a complete graph G'!
- $\forall e \in E'$,
  $d(e) =$
- $B =$

Polynomial Reduction 1/2

Reduction

Let $G = (V, E)$ be an instance of HAM, we construct $\langle G' = (V', E'), d, B \rangle$ with:

- $V' = V$
  We keep the same vertices...
- $E' = \{\{u, v\}, \forall u, v \in V \wedge u \neq v\}$
  ...but we construct a complete graph G'!
- $\forall e \in E'$,
  $d(e) = 0$ if $e \in E$,
  $d(e) = 1$ if $e \notin E$
- $B =$

Polynomial Reduction 1/2

Reduction

Let $G = (V, E)$ be an instance of HAM, we construct $\langle G' = (V', E'), d, B \rangle$ with:

- $V' = V$
  We keep the same vertices...
- $E' = \{\{u, v\}, \forall u, v \in V \wedge u \neq v\}$
  ...but we construct a complete graph G'!
- $\forall e \in E'$,
  $d(e) = 0$ if $e \in E$,
  $d(e) = 1$ if $e \notin E$
- $B = 0$

Polynomial Reduction 2/2

Let's show that it is indeed a polynomial reduction

NB: *The transformation is polynomial (check each operation)*

Polynomial Reduction 2/2

Let's show that it is indeed a polynomial reduction

NB: *The transformation is polynomial (check each operation)*

$\Longrightarrow$ Let $S = [s_1, ..., s_n]$ be a solution of the instance G of HAM

Polynomial Reduction 2/2

Let's show that it is indeed a polynomial reduction

***NB:** The transformation is polynomial (check each operation)*

$\Longrightarrow$ Let $S = [s_1, ..., s_n]$ be a solution of the instance G of HAM
S also defines a cycle in G' of weight 0 *(all $d(e)$ are 0)*

➜ So there is a solution for TSP.

Polynomial Reduction 2/2

Let's show that it is indeed a polynomial reduction

NB: *The transformation is polynomial (check each operation)*

$\Longrightarrow$ Let $S = [s_1, ..., s_n]$ be a solution of the instance G of HAM
S also defines a cycle in G' of weight 0 *(all $d(e)$ are 0)*

➜ So there is a solution for TSP.

$\Longleftarrow$ Let $S = [s_1, ..., s_n]$ be a solution of the instance $\langle G', d, B\rangle$ of TSP *obtained by transformation from G*

Polynomial Reduction 2/2

Let's show that it is indeed a polynomial reduction

***NB:** The transformation is polynomial (check each operation)*

$\implies$ Let $S = [s_1, ..., s_n]$ be a solution of the instance G of HAM
S also defines a cycle in G' of weight 0 *(all $d(e)$ are 0)*

➜ So there is a solution for TSP.

$\impliedby$ Let $S = [s_1, ..., s_n]$ be a solution of the instance $\langle G', d, B\rangle$ of TSP *obtained by transformation from G*

$B = 0$ and $d(e) = 1$ for all $e \notin E$, so the solution only borrows edges from E!

Polynomial Reduction 2/2

Let's show that it is indeed a polynomial reduction

NB: *The transformation is polynomial (check each operation)*

$\Longrightarrow$ Let $S = [s_1, ..., s_n]$ be a solution of the instance G of HAM
S also defines a cycle in G' of weight 0 *(all $d(e)$ are 0)*

➜ So there is a solution for TSP.

$\Longleftarrow$ Let $S = [s_1, ..., s_n]$ be a solution of the instance $\langle G', d, B\rangle$ of TSP *obtained by transformation from G*

$B = 0$ and $d(e) = 1$ for all $e \notin E$, so the solution only borrows edges from E!

➜ S also defines a solution to the G instance of HAM

By contraposition: no sol. for HAM instance $\Rightarrow$ no sol. for TSP instance

Polynomial Reduction 2/2

Let's show that it is indeed a polynomial reduction

NB: *The transformation is polynomial (check each operation)*

$\Longrightarrow$ Let $S = [s_1, ..., s_n]$ be a solution of the instance G of HAM
S also defines a cycle in G' of weight 0 *(all $d(e)$ are 0)*

➜ So there is a solution for TSP.

$\Longleftarrow$ Let $S = [s_1, ..., s_n]$ be a solution of the instance $\langle G', d, B \rangle$ of TSP *obtained by transformation from G*

$B = 0$ and $d(e) = 1$ for all $e \notin E$, so the solution only borrows edges from E!

➜ S also defines a solution to the G instance of HAM

By contraposition: no sol. for HAM instance $\Rightarrow$ no sol. for TSP instance

$$\text{HAM} \leq \text{TSP}$$

TSP complexity

TSP is *NP*-Complete

- Travelling Salesman is in *NP*
- Hamiltonian Cycle is *NP*-complete
- Hamiltonian Cycle $\leq$ Travelling Salesman

TSP complexity

TSP is *NP*-Complete

- Travelling Salesman is in *NP*
- Hamiltonian Cycle is *NP*-complete
- Hamiltonian Cycle $\leq$ Travelling Salesman

➜ Travelling Salesman (decision problem) is *NP*-complete

TSP complexity

TSP is *NP*-Complete

- Travelling Salesman is in *NP*
- Hamiltonian Cycle is *NP*-complete
- Hamiltonian Cycle $\leq$ Travelling Salesman

→ Travelling Salesman (decision problem) is *NP*-complete

Conclusion

It is not possible to compute an optimal solution in polynomial time (unless P=NP . . .)

Plan

Handle NP-hard optimization problems

Exact methods

We look for the best solution ... by trying to be efficient!

Examples: Backtracking, Branch & Bound, Linear programming, ...

Handle NP-hard optimization problems

Exact methods

We look for the best solution ... by trying to be efficient!

Examples: Backtracking, Branch & Bound, Linear programming, ...

Methods not necessarily exact, but in polynomial time

- heuristics algorithms and approximation algorithms
  ➜ ex: greedy, see go further in the course
- Randomized algorithms (Monte Carlo, Las Vegas)
- General methods of exploring solution space
  ➜ metaheuristics (ex: simulated annealing, genetic algorithms ...)

Handle NP-hard optimization problems

Exact methods

We look for the best solution . . . by trying to be efficient!

Examples: **Brute Force**, **Backtracking**, Branch & Bound, Linear programming, . . .

Brute Force (exhaustive search)

Principle

1. Enumerate successively all configurations. All possible solutions!

   in TSP: all lists of $|V|$ nodes (all possible cycles)

2. Evaluate the score of each configuration

   in TSP: compute for each cycle the sum of edges' weights

3. Keep the best configuration

   in TSP: choose the cycle with the lowest score

Brute Force (exhaustive search)

Principle

1. Enumerate successively all configurations. All possible solutions!

   in TSP: all lists of $|V|$ nodes (all possible cycles)

   ➜ $\frac{|V-1|!}{2}$ *possible solutions*

2. Evaluate the score of each configuration

   in TSP: compute for each cycle the sum of edges' weights

3. Keep the best configuration

   in TSP: choose the cycle with the lowest score

Exponential complexity

Backtracking

Principle

- Iterative construction of solutions

→ Determine the set of possible configurations

Backtracking

Principle

- Iterative construction of solutions

➜ Determine the set of possible configurations

Example:

In TSP, at each step, we separate cases depending on :

- Option 1 : the next node to visit
- Option 2 : adding or eliminating an edge

Backtracking

Principle

- Iterative construction of solutions

➜ Determine the set of possible configurations

Example:

In TSP, at each step, we separate cases depending on :

- Option 1 : the next node to visit
- Option 2 : adding or eliminating an edge

- We **explore** the solutions space.

Backtracking

Principle

- Iterative construction of solutions

➜ Determine the set of possible configurations

Example:

In TSP, at each step, we separate cases depending on :

- Option 1 : the next node to visit
- Option 2 : adding or eliminating an edge

- We **explore** the solutions space.

➜ Backtracking is an exploration by **branching** over the solutions space

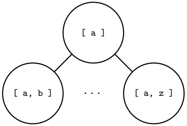
Exploration of the solutions space

Principle

➜ The solutions space can be seen as a tree

where branches correspond to the iterative construction of the solutions

Example: in TSP, append a new element to the list (option 1)



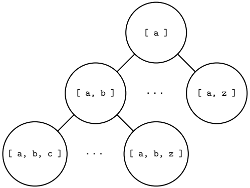
Exploration of the solutions space

Principle

➜ The solutions space can be seen as a tree

where branches correspond to the iterative construction of the solutions

Example: in TSP, append a new element to the list (option 1)



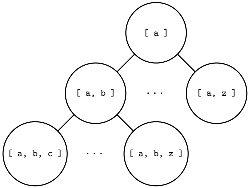
Exploration of the solutions space

Principle

➜ The solutions space can be seen as a tree

where branches correspond to the iterative construction of the solutions

Example: in TSP, append a new element to the list (option 1)

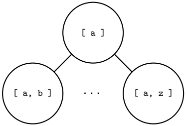


➜ Leaves contain possible solutions.

Exploration of the solutions space

Principle

Example: in TSP, append a new element to the list (option 1)



➜ Solution space seen as a tree.

- Enumerate solutions by a depth-first exploration of this tree.

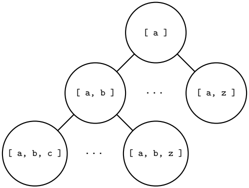
  → The aim is to choose an optimal solution by examining possible solutions in the leaves.

  → We say that this tree is implicit when it is built as the exploration progresses. We do not represent the solution tree in memory!

Exploration of the solutions space

Principle

Example: in TSP, append a new element to the list (option 1)



➜ Solution space seen as a tree.

- Enumerate solutions by a depth-first exploration of this tree.
  $\rightarrow$ The aim is to choose an optimal solution by examining possible solutions in the leaves.
  $\rightarrow$ We say that this tree is implicit when it is built as the exploration progresses. We do not represent the solution tree in memory!

TSP example (option 2)

Branching depending on edges

- At each step, we separate the set of Hamiltonian cycles, between those who will take a chosen edge $\{i, j\}$ and those who will not.

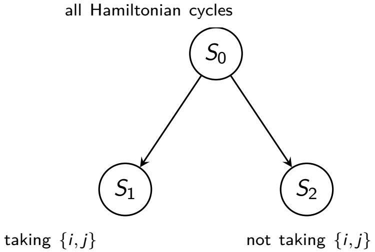
➜ Binary tree of height $|E|$

TSP example (option 2)

Branching depending on edges

- At each step, we separate the set of Hamiltonian cycles, between those who will take a chosen edge $\{i, j\}$ and those who will not.

→ Binary tree of height $|E|$

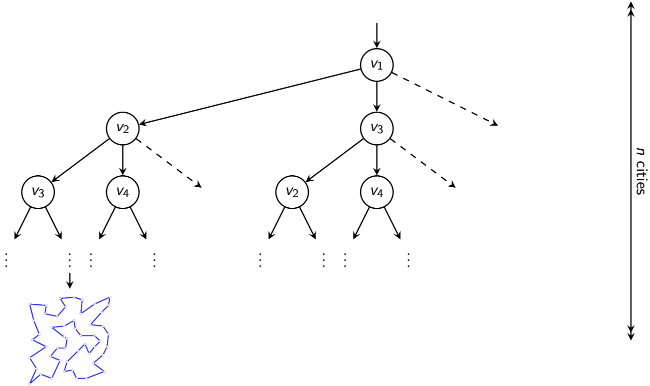


TSP example (option 1)

Branching depending on next nodes

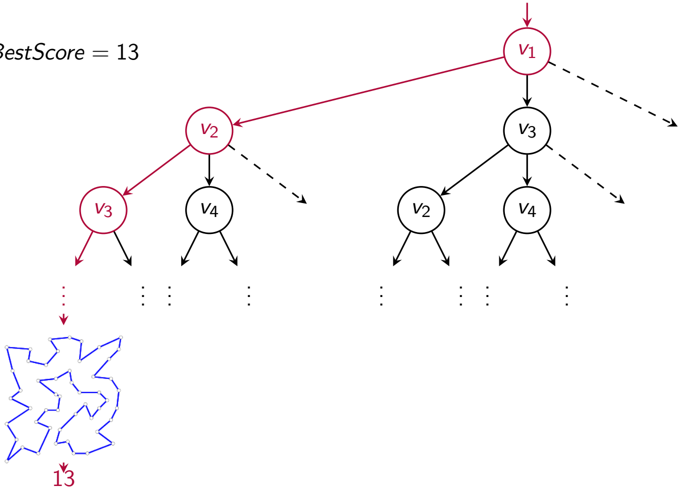
- At each step: choose a city among non-visited ones;

➜ Tree : each node has as many children as remaining nodes.

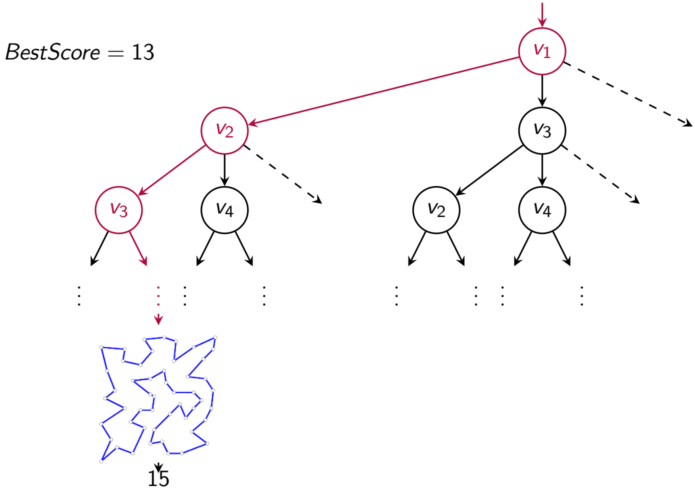


Backtracking algorithm

$BestScore = 13$

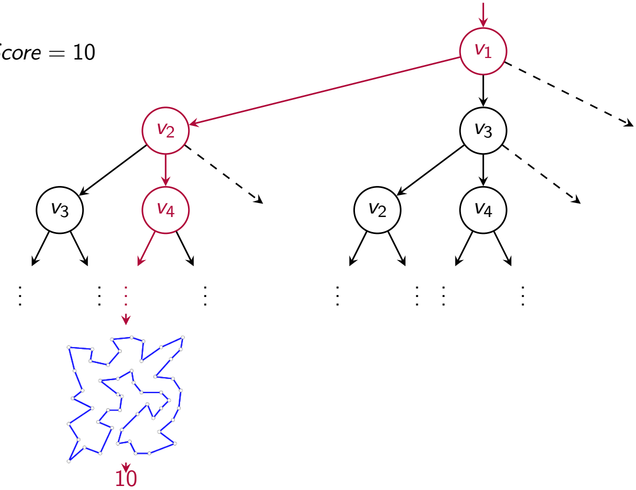


Backtracking algorithm



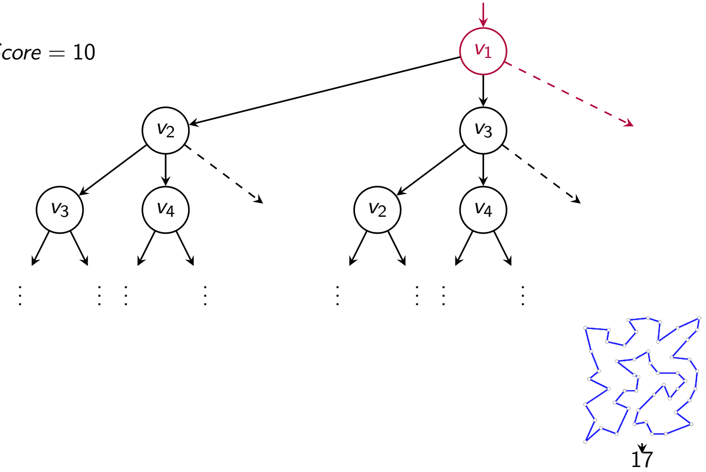
Backtracking algorithm

$BestScore = 10$



Backtracking algorithm

$BestScore = 10$



Backtracking algorithm

Principle

For any partial or terminal solution s, we assume that we have the following functions:

- $children(s)$: returns next step partial solutions of s
- $terminal(s)$: returns *true* if the solution is terminal, *false* otherwise
- $score(s)$: returns the score of the terminal solution s

Backtracking algorithm

code skeleton

```
bestScore = Inf
bestSol = None
def backtracking(s) :
    if terminal(s) :
        if score(s) < bestScore :
            bestScore = score(s)
            bestSol = s
    else :
        for c in children(s):
            backtracking(c)
```

Improving the algorithm

How to improve this algorithm?

Improving the algorithm

How to improve this algorithm?

Principle

We explore the space of solutions while pruning/cutting non-promising branches. **Please note: improvements do not reduce complexity, which will remain exponential!**

Improving the algorithm

How to improve this algorithm?

Principle

We explore the space of solutions while pruning/cutting non-promising branches. **Please note: improvements do not reduce complexity, which will remain exponential!**

Travelling Salesman case

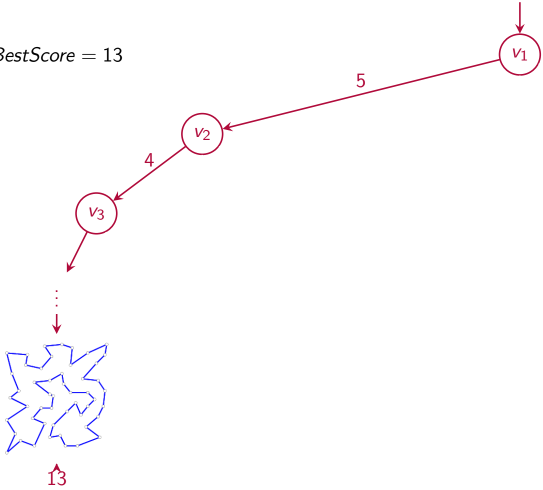
- We note the score of the current best terminal solution: *BestScore*
- A branch is a partial solution: $s = [s_1, \ldots, s_k]$ (the beginning of a Hamiltonian path)

➜ A non-promising branch is a partial solution that is already longer than *BestScore*: $\sum_{s_i \in s} d(s_i, s_{i+1}) > BestScore$

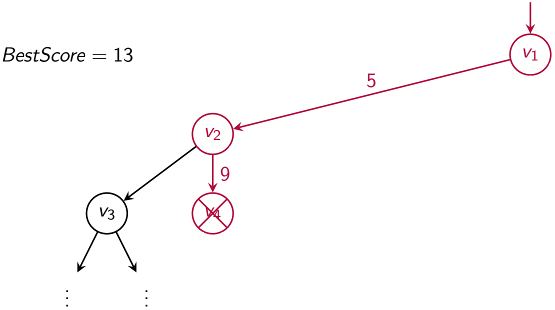
✗ We stop the exploration of that branch!

Backtracking improvement

$BestScore = 13$

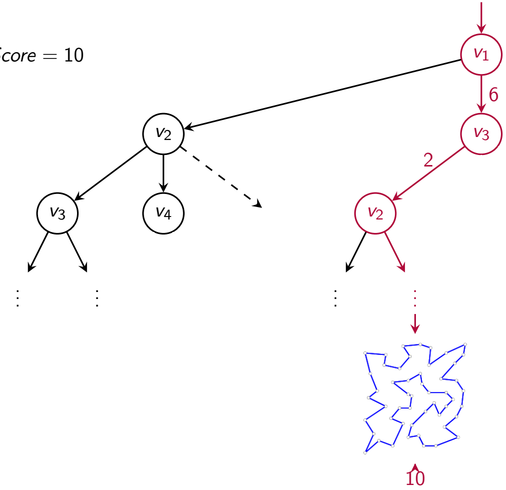


Backtracking improvement

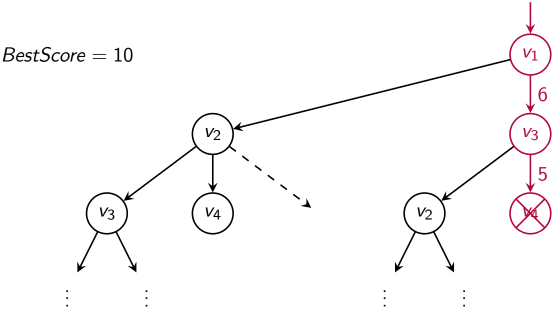


Backtracking improvement

$BestScore = 10$

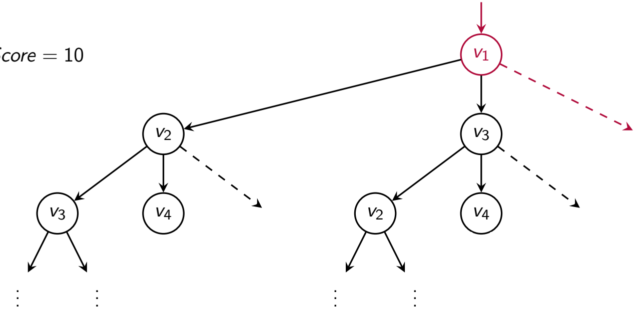


Backtracking improvement



Backtracking improvement

$BestScore = 10$



Plan

Polynomial time algorithms for hard problems

Problem

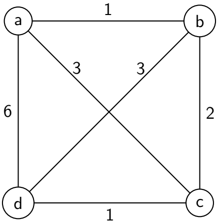
The algorithms producing an optimal solution have an exponential complexity

- they can handle small instances

For large instances, you have to be satisfied with a solution that will not necessarily be optimal . . .

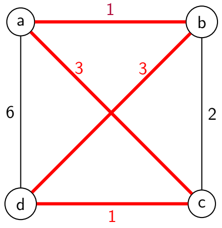
Let us take a TSP instance...

...so small that an optimal solution is obvious.



Let us take a TSP instance...

...so small that an optimal solution is obvious.



Optimal solution = 1+3+1+3=8

Greedy TSP

Idea of a greedy algorithm

1. Choose a starting vertex v arbitrarily
2. Repeat until the entire tour is made
    1. Chose among all neighbors of v the closest to it and not included in the tour under construction, v',
    2. v' becomes a new current vertex, $v \leftarrow v'$.

And its complexity

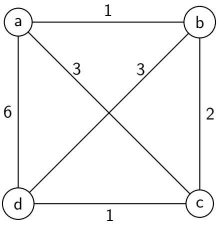
Our algorithm is in polynomial time (as it looks like as one of the graph traversals).

skip greedy in python

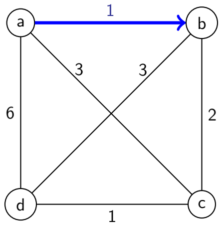
Its possible implementation (based upon DFS, the graphe definad as a matrix)

```
def ClstNeighbor_TSP(graph,v): # v - current
    tour.append(v)
    if len(tour)==len(graph):
        return tour  # complet tour

    min_dist = math.inf; candidat = None
    for n in graph[v]:
        if not n in tour:
            if graph[v][n]<min_dist:
               min_dist = graph[v][n]
               candidat = n

    ClstNeighbor_TSP(graph, candidat)

tour = [] # to collect the vertex order
ClstNeighbor_TSP(graph,arbitrary_start)
```

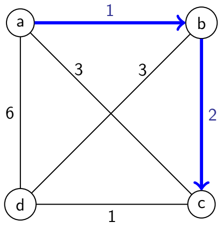
Solution according to “the closest neighbor” approach



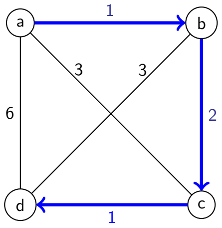
Solution according to “the closest neighbor” approach



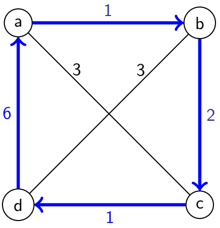
Solution according to “the closest neighbor” approach



Solution according to "the closest neighbor" approach

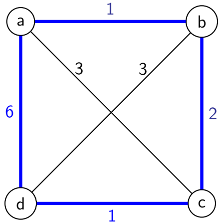


Solution according to “the closest neighbor” approach



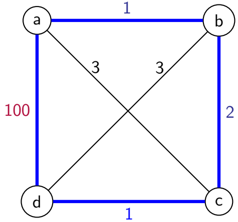
Greedy solution = 1+2+1+**6**=10

Solution quality of the “the closest neighbor” approach



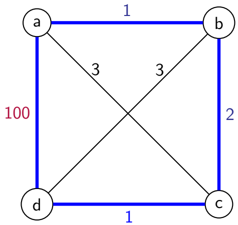
Greedy solution = 1+2+1+**6**=10

Solution quality of the "the closest neighbor" approach



Greedy solution = 1+2+1+**100**=104

Solution quality of the “the closest neighbor” approach

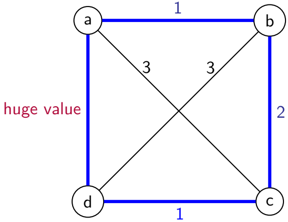


Greedy solution $= 1{+}2{+}1{+}\mathbf{100}{=}104$

The last edge determines the solution quality.

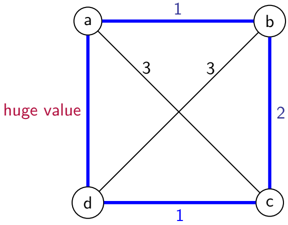
Replacing the distance **6** by a huge value degrades the solution quality.

Solution quality of the "the closest neighbor" approach



Greedy solution = 1+2+1+**huge value** = **huge value**

Solution quality of the “the closest neighbor” approach



Greedy solution = 1+2+1+**huge value** = **huge value**

Conclusion

The quality of a solution produced by our algorithm is not guaranteed.

Solution quality

Heuristics

A heuristic algorithm solves a difficult problem in polynomial time without guarantee of the solution quality.

Solution quality

Heuristics

A heuristic algorithm solves a difficult problem in polynomial time without guarantee of the solution quality.

Approximation algorithm

An approximation algorithm produces a solution to a difficult problem **in polynomial time** whose quality is known. We know **how many times at worst** the solution produced for a problem of

- **minimization** is **greater** than the optimal solution
- **maximization** is **smaller** than the optimal solution.

For an idea...

...of how to proceed we will present you an approximation algorithm solving TSP.

For an idea...

...of how to proceed we will present you an approximation algorithm solving TSP.

We restrict ourselves to metric spaces

where **the triangular inequality is valid** and we proceed by construction*.

*No, it will not be greedy this time!

For an idea...

...of how to proceed we will present you an approximation algorithm solving TSP.

We restrict ourselves to metric spaces

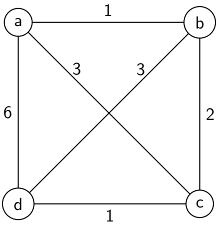
where **the triangular inequality is valid** and we proceed by construction*.

*No, it will not be greedy this time!

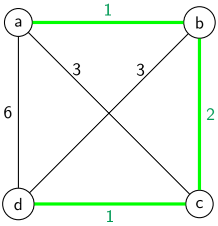
Before starting

Our algorithm is in three steps. In their description OPT denotes the length of the **optimal solution** of TSP that **we do not know**.

First step : Find a solution T to MST for G



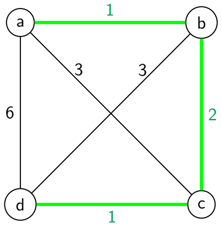
First step : Find a solution T to MST for G



Observation

$$\text{weight}(T) \leq \text{OPT}$$

First step : Find a solution T to MST for G



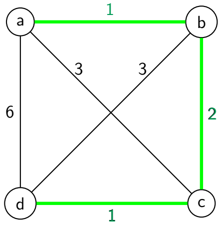
Observation

$$\text{weight}(T) \leq \text{OPT}$$

see lecture 3 on MST: the weight of an MST will never be greater than the length of an optimal TSP solution which is a cycle

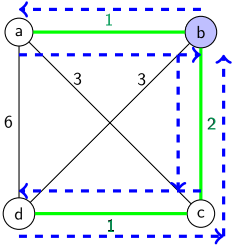
Complexity: polynomial (the same as Kruskal/Prim)

Second step: Do a DFS of T



T_D is a DFS tree of T such that any T edge is traversed **twice**.
Starting from b : $T_D = (b, a, b, c, d, c, b)$.

Second step: Do a DFS of T

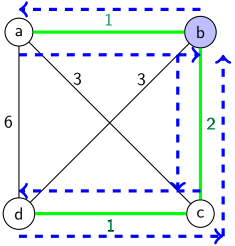


T_D is a DFS tree of T such that any T edge is traversed **twice**.
Starting from b : $T_D = (b, a, b, c, d, c, b)$.

Observation

$$\text{length}(T_D) \leq 2\text{OPT}$$

Second step: Do a DFS of T



T_D is a DFS tree of T such that any T edge is traversed **twice**.
Starting from b : $T_D = (b, a, b, c, d, c, b)$.

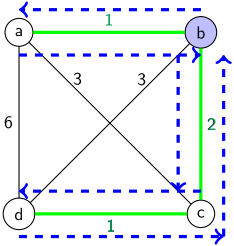
Observation

$$\text{length}(T_D) \leq 2\text{OPT}$$

the length of the T_D traversal is at most **twice** as large as the weight of T.
Complexity: polynomial (the same as DFS).

Third step: Transform T_D into a cycle C

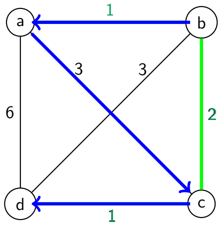
Keep only the first occurrence of a vertex in T_D:
$T_D = (b, a, b, c, d, c, b) \rightarrow (b, a, \not{b}, c, d, \not{c}, \not{b}) \rightarrow (b, a, c, d) = C.$



Third step: Transform T_D into a cycle C

Keep only the first occurrence of a vertex in T_D:

$T_D = (b, a, b, c, d, c, b) \rightarrow (b, a, \not{b}, c, d, \not{c}, \not{b}) \rightarrow (b, a, c, d) = C.$

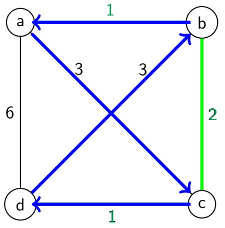


Observation

Third step: Transform T_D into a cycle C

Keep only the first occurrence of a vertex in T_D:

$T_D = (b, a, b, c, d, c, b) \rightarrow (b, a, \not{b}, c, d, \not{c}, \not{b}) \rightarrow (b, a, c, d) = C.$

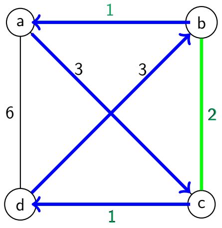


Observation

Thanks to the triangular inequality: $\text{length}(C) \leq \text{length}(T_D)$

Third step: Transform T_D into a cycle C

Keep only the first occurrence of a vertex in T_D:
$T_D = (b, a, b, c, d, c, b) \rightarrow (b, a, \not{b}, c, d, \not{c}, \not{b}) \rightarrow (b, a, c, d) = C.$

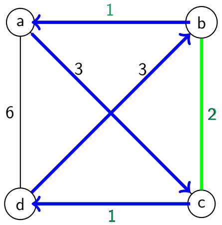


Observation

Thanks to the triangular inequality: $\text{length}(C) \leq \text{length}(T_D)$
consequently: $\text{length}(C) \leq \text{length}(T_D) \leq 2\text{OPT}$.

Third step: Transform T_D into a cycle C

Keep only the first occurrence of a vertex in T_D:
$T_D = (b, a, b, c, d, c, b) \rightarrow (b, a, \not{b}, c, d, \not{c}, \not{b}) \rightarrow (b, a, c, d) = C.$



Observation

Thanks to the triangular inequality: length$(C) \leq$ length(T_D)
consequently: length$(C) \leq$ length$(T_D) \leq 2$OPT.
Complexity: polynomial (linear in $|V|$).

To end with TSP

Recap

- our approximation algorithm produces a solution never **twice** longer than an optimal solution in a space where the triangular inequality holds,
- there is an approximation algorithm taking advantage of the triangular inequality that yields a solution never $\frac{3}{2}$ times longer than an optimal solution (Christofides' algorithm: a five-step construction, its first step being a solution to MST),
- it is impossible to find an approximation algorithm for TSP in non-metric spaces.

Approximation formally

Let $\mathcal{P}$ be an optimization problem, f the function for evaluating the solutions of $\mathcal{P}$ and $\mathcal{A}$ an approximation algorithm.

Approximation formally

Let $\mathcal{P}$ be an optimization problem, f the function for evaluating the solutions of $\mathcal{P}$ and $\mathcal{A}$ an approximation algorithm.

Definition

Let I be an instance of $\mathcal{P}$, and S be a solution for I produced by $\mathcal{A}$ and S^* be one optimal solution of I.

- the approximation ratio of S on I is: $\rho(I, S) = \frac{f(S)}{f(S^*)}$
- the algorithm $\mathcal{A}$ is a ρ -approximation for $\mathcal{P}$ if and only if: $\rho(I, S) \leq \rho$

Pros and cons of an approximation algorithm

Pros and cons of an approximation algorithm

- ✓ Polynomial time
- ✓ Solution quality guaranteed
- ✓ In practice, a solution obtained is potentially better than the theoretical guarantee
- ✗ Impossible to find such an algorithm*
- ✗ Polynomial time, but execution in practice too long§
- ✗ Difficult to implement§
- ✗ Solutions produced "far from" an exact solution§

---

*for some problems they do not exist
§possible

Plan

Keep in mind

NP-hard optimization problem ...

- Exact solution only for small instances

  because the complexity of an exact algorithm is a priori exponential

- Large instances → approximate method

  ➜ We will look in polynomial time for *solutions* " *which are not that bad* " (but they are not necessarily optimal).