Algorithms (XVII)

YuYu
Shanghai Jiaotong University

Chapter 9. Coping with NP-completeness

Intelligent exhaustive search

Backtracking

Backtracking

It is often possible to *reject a solution* by looking at just *a small portion of it*.

Abstract formulation

Abstract formulation

A backtracking algorithm requires a **test** that looks at a subproblem and quickly declares one of three outcomes:

Abstract formulation

A backtracking algorithm requires a **test** that looks at a subproblem and quickly declares one of three outcomes:

1. Failure: the subproblem has no solution.

Abstract formulation

A backtracking algorithm requires a **test** that looks at a subproblem and quickly declares one of three outcomes:

1. Failure: the subproblem has no solution.
2. Success: a solution to the subproblem is found.

Abstract formulation

A backtracking algorithm requires a **test** that looks at a subproblem and quickly declares one of three outcomes:

1. Failure: the subproblem has no solution.
2. Success: a solution to the subproblem is found.
3. Uncertainty.

Abstract formulation

A backtracking algorithm requires a **test** that looks at a subproblem and quickly declares one of three outcomes:

1. Failure: the subproblem has no solution.
2. Success: a solution to the subproblem is found.
3. Uncertainty.

For SAT, this test declares failure if there is an *empty clause*,

Abstract formulation

A backtracking algorithm requires a **test** that looks at a subproblem and quickly declares one of three outcomes:

1. Failure: the subproblem has no solution.
2. Success: a solution to the subproblem is found.
3. Uncertainty.

For SAT, this test declares failure if there is an *empty clause*, success if there are *no clauses*,

Abstract formulation

A backtracking algorithm requires a **test** that looks at a subproblem and quickly declares one of three outcomes:

1. Failure: the subproblem has no solution.
2. Success: a solution to the subproblem is found.
3. Uncertainty.

For SAT, this test declares failure if there is an *empty clause*, success if there are *no clauses*, and uncertainty otherwise.

Abstract formulation (cont'd)

Abstract formulation (cont'd)

Start with some problem P_0
Let $\mathcal{S} = \{P_0\}$, the set of active subproblems
Repeat while $\mathcal{S}$ is nonempty:
    choose a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
    expand it into smaller subproblems $P_1, P_2, \ldots, P_k$
    for all P_i:
        if test(P_i) succeeds then halt
          and announce this solution
        if test(P_i) fails then discard P_i
        else add P_i to $\mathcal{S}$
Announce that there is no solution.

Abstract formulation (cont'd)

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
Repeat while S is nonempty:
    choose a subproblem P ∈ S and remove it from S
    expand it into smaller subproblems P_1, P_2, ..., P_k
    for all P_i:
        if test(P_i) succeeds then halt
          and announce this solution
        if test(P_i) fails then discard P_i
        else add P_i to S
Announce that there is no solution.
```

For SAT, the choose procedure picks a clause,

Abstract formulation (cont'd)

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
Repeat while S is nonempty:
    choose a subproblem P ∈ S and remove it from S
    expand it into smaller subproblems P_1, P_2, ..., P_k
    for all P_i:
        if test(P_i) succeeds then halt
          and announce this solution
        if test(P_i) fails then discard P_i
        else add P_i to S
Announce that there is no solution.
```

For SAT, the choose procedure picks a clause, and expand picks a variable within that clause.

Branch-and-bound

Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems.

Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems.

For concreteness, let's say we have a **minimization problem**;

Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems.

For concreteness, let's say we have a **minimization problem**; maximization will follow the same pattern.

Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems.

For concreteness, let's say we have a **minimization problem**; maximization will follow the same pattern.

We will deal with *partial solutions*,

Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems.

For concreteness, let's say we have a **minimization problem**; maximization will follow the same pattern.

We will deal with *partial solutions*, each represents a subproblem: *what is the (cost of the) best way to complete this solution?*

Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems.

For concreteness, let's say we have a **minimization problem**; maximization will follow the same pattern.

We will deal with *partial solutions*, each represents a subproblem: *what is the (cost of the) best way to complete this solution?*

And as before, we need a basis for *eliminating* partial solutions, since there is no other source of efficiency in our method.

Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems.

For concreteness, let's say we have a **minimization problem**; maximization will follow the same pattern.

We will deal with *partial solutions*, each represents a subproblem: *what is the (cost of the) best way to complete this solution?*

And as before, we need a basis for *eliminating* partial solutions, since there is no other source of efficiency in our method.

To reject a subproblem, we must be certain that its cost exceeds that of some other solution we have already encountered.

Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems.

For concreteness, let's say we have a **minimization problem**; maximization will follow the same pattern.

We will deal with *partial solutions*, each represents a subproblem: *what is the (cost of the) best way to complete this solution?*

And as before, we need a basis for *eliminating* partial solutions, since there is no other source of efficiency in our method.

To reject a subproblem, we must be certain that its cost exceeds that of some other solution we have already encountered. But its exact cost is unknown to us and is generally not efficiently computable.

Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems.

For concreteness, let's say we have a **minimization problem**; maximization will follow the same pattern.

We will deal with *partial solutions*, each represents a subproblem: *what is the (cost of the) best way to complete this solution?*

And as before, we need a basis for *eliminating* partial solutions, since there is no other source of efficiency in our method.

To reject a subproblem, we must be certain that its cost exceeds that of some other solution we have already encountered. But its exact cost is unknown to us and is generally not efficiently computable.

So instead we use a *quick lower bound on this cost*.

Abstract formulation

Abstract formulation

Start with some problem P_0
Let $\mathcal{S} = \{P_0\}$, the set of active subproblems
bestsofar $= \infty$
Repeat while $\mathcal{S}$ is nonempty:
    choose a subproblem (partial solution) $P \in \mathcal{S}$
      and remove it from $\mathcal{S}$
    expand it into smaller subproblems $P_1, P_2, \ldots, P_k$
    for all P_i:
        if P_i is a complete solution then update bestsofar
        else if lowerbound$(P_i) <$ bestsofar then add P_i to $\mathcal{S}$
return bestsofar.

Application to TSP

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$,

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b.

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b.

We can denote such a partial solution by the tuple $[a, S, b]$

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b.

We can denote such a partial solution by the tuple $[a, S, b]$–in fact, a will be fixed throughout the algorithm.

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b.

We can denote such a partial solution by the tuple $[a, S, b]$–in fact, a will be fixed throughout the algorithm.

The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V \setminus S$.

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b.

We can denote such a partial solution by the tuple $[a, S, b]$–in fact, a will be fixed throughout the algorithm.

The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V \setminus S$.

The initial problem is of the form $[a; \{a\}, a]$ for any $a \in V$ of our choosing.

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b.

We can denote such a partial solution by the tuple $[a, S, b]$–in fact, a will be fixed throughout the algorithm.

The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V \setminus S$.

The initial problem is of the form $[a; \{a\}, a]$ for any $a \in V$ of our choosing.

At each step of the branch-and-bound algorithm, we extend a particular partial solution $[a, S, b]$ by *a single edge* (b, x),

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b.

We can denote such a partial solution by the tuple $[a, S, b]$–in fact, a will be fixed throughout the algorithm.

The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V \setminus S$.

The initial problem is of the form $[a; \{a\}, a]$ for any $a \in V$ of our choosing.

At each step of the branch-and-bound algorithm, we extend a particular partial solution $[a, S, b]$ by *a single edge* (b, x), where $x \in V \setminus S$.

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b.

We can denote such a partial solution by the tuple $[a, S, b]$–in fact, a will be fixed throughout the algorithm.

The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V \setminus S$.

The initial problem is of the form $[a; \{a\}, a]$ for any $a \in V$ of our choosing.

At each step of the branch-and-bound algorithm, we extend a particular partial solution $[a, S, b]$ by *a single edge* (b, x), where $x \in V \setminus S$.

There can be up to $|V \setminus S|$ ways to do this,

Application to TSP

TSP on a graph $G = (V, E)$ with edge lengths $d_e > 0$.

A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b.

We can denote such a partial solution by the tuple $[a, S, b]$–in fact, a will be fixed throughout the algorithm.

The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V \setminus S$.

The initial problem is of the form $[a; \{a\}, a]$ for any $a \in V$ of our choosing.

At each step of the branch-and-bound algorithm, we extend a particular partial solution $[a, S, b]$ by *a single edge* (b, x), where $x \in V \setminus S$.

There can be up to $|V \setminus S|$ ways to do this, and each of these branches leads to a subproblem of the form $[a, S \cup \{x\}, x]$.

Application to TSP (cont'd)

Application to TSP (cont'd)

How can we lower-bound the cost of completing a partial tour $[a, S, b]$?

Application to TSP (cont'd)

How can we lower-bound the cost of completing a partial tour $[a, S, b]$?

A simple solution:

Application to TSP (cont'd)

How can we lower-bound the cost of completing a partial tour $[a, S, b]$?

A simple solution:

The remainder of the tour consists of a path through $V \setminus S$,

Application to TSP (cont'd)

How can we lower-bound the cost of completing a partial tour $[a, S, b]$?

A simple solution:

The remainder of the tour consists of a path through $V \setminus S$, plus edges from a and b to $V \setminus S$.

Application to TSP (cont'd)

How can we lower-bound the cost of completing a partial tour $[a, S, b]$?

A simple solution:

The remainder of the tour consists of a path through $V \setminus S$, plus edges from a and b to $V \setminus S$.

1. The *lightest* edge from a to $V \setminus S$.

Application to TSP (cont'd)

How can we lower-bound the cost of completing a partial tour $[a, S, b]$?

A simple solution:

The remainder of the tour consists of a path through $V \setminus S$, plus edges from a and b to $V \setminus S$.

1. The *lightest* edge from a to $V \setminus S$.
2. The *lightest* edge from b to $V \setminus S$.

Application to TSP (cont'd)

How can we lower-bound the cost of completing a partial tour $[a, S, b]$?

A simple solution:

The remainder of the tour consists of a path through $V \setminus S$, plus edges from a and b to $V \setminus S$.

1. The *lightest* edge from a to $V \setminus S$.
2. The *lightest* edge from b to $V \setminus S$.
3. The *minimum spanning tree* of $V \setminus S$.

Application to TSP (cont'd)

How can we lower-bound the cost of completing a partial tour $[a, S, b]$?

A simple solution:

The remainder of the tour consists of a path through $V \setminus S$, plus edges from a and b to $V \setminus S$.

1. The *lightest* edge from a to $V \setminus S$.
2. The *lightest* edge from b to $V \setminus S$.
3. The *minimum spanning tree* of $V \setminus S$.

This lower bound can be computed quickly by a minimum spanning tree algorithm.

Approximation algorithms

Optimization problems

In an optimization problem we are given an instance I and are asked to find the optimum solution,

Optimization problems

In an optimization problem we are given an instance I and are asked to find the optimum solution,

- the one with the maximum gain if we have a maximization problem like INDEPENDENT SET,

Optimization problems

In an optimization problem we are given an instance I and are asked to find the optimum solution,

- the one with the maximum gain if we have a maximization problem like INDEPENDENT SET,
- or the minimum cost if we are dealing with a minimization problem such as the TSP.

Optimization problems

In an optimization problem we are given an instance I and are asked to find the optimum solution,

- the one with the maximum gain if we have a maximization problem like INDEPENDENT SET,
- or the minimum cost if we are dealing with a minimization problem such as the TSP.

For every instance I, $\mathrm{OPT}(I)$ denotes the value (benefit or cost) of the optimum solution.

Optimization problems

In an optimization problem we are given an instance I and are asked to find the optimum solution,

- the one with the maximum gain if we have a maximization problem like INDEPENDENT SET,
- or the minimum cost if we are dealing with a minimization problem such as the TSP.

For every instance I, $\mathrm{OPT}(I)$ denotes the value (benefit or cost) of the optimum solution.

We always assume $\mathrm{OPT}(I)$ is a positive integer.

Approximation ratio

Approximation ratio

We have seen the greedy algorithm for SET COVER:

Approximation ratio

We have seen the greedy algorithm for SET COVER:

For any instance I of size n, this greedy algorithm quickly finds a set cover of cardinality at most $\mathrm{OPT}(I) \cdot \log n$.

Approximation ratio

We have seen the greedy algorithm for SET COVER:

For any instance I of size n, this greedy algorithm quickly finds a set cover of cardinality at most $\mathrm{OPT}(I) \cdot \log n$.

This $\log n$ factor is known as the **approximation guarantee** of the algorithm.

Approximation ratio

We have seen the greedy algorithm for SET COVER:

For any instance I of size n, this greedy algorithm quickly finds a set cover of cardinality at most $\mathrm{OPT}(I) \cdot \log n$.

This $\log n$ factor is known as the **approximation guarantee** of the algorithm.

Suppose now that we have an algorithm $\mathcal{A}$ for a minimization problem which, given an instance I, returns a solution with *value* $\mathcal{A}(I)$.

Approximation ratio

We have seen the greedy algorithm for SET COVER:

For any instance I of size n, this greedy algorithm quickly finds a set cover of cardinality at most $\mathrm{OPT}(I) \cdot \log n$.

This $\log n$ factor is known as the **approximation guarantee** of the algorithm.

Suppose now that we have an algorithm $\mathcal{A}$ for a minimization problem which, given an instance I, returns a solution with *value* $\mathcal{A}(I)$.

The approximation ratio of $\mathcal{A}$ is defined to be

$$\alpha_{\mathcal{A}} = \max_{I} \frac{\mathcal{A}(I)}{\mathrm{OPT}(I)}$$

Vertex cover

Vertex cover

We already know the VERTEX COVER problem is **NP**-hard.

Vertex cover

We already know the VERTEX COVER problem is **NP**-hard.

Input:	An undirected graph $G = (V, E)$.
Output:	A subset of the vertices $S \subseteq V$ that touches every edge.
Goal:	Minimize $\lvert S \rvert$.

Matching vs. Vertex Cover

Matching vs. Vertex Cover

Definition

A matching is a subset of edges that have no vertices in common.

Matching vs. Vertex Cover

Definition

A matching is a subset of edges that have no vertices in common.

Definition

A matching is *maximal* if no more edges can be added to it.

Matching vs. Vertex Cover

Definition

A matching is a subset of edges that have no vertices in common.

Definition

A matching is *maximal* if no more edges can be added to it.

Theorem

Any vertex cover of a graph G must be at least as large as the number of edges in any matching in G.

Matching vs. Vertex Cover

Definition
A matching is a subset of edges that have no vertices in common.

Definition
A matching is *maximal* if no more edges can be added to it.

Theorem
Any vertex cover of a graph G must be at least as large as the number of edges in any matching in G.

Theorem
Let S be a set that contains both endpoints of each edge in a maximal matching M.

Matching vs. Vertex Cover

Definition

A matching is a subset of edges that have no vertices in common.

Definition

A matching is *maximal* if no more edges can be added to it.

Theorem

Any vertex cover of a graph G must be at least as large as the number of edges in any matching in G.

Theorem

Let S be a set that contains both endpoints of each edge in a maximal matching M. Then S must be a vertex cover.

2-approximation of VERTEX COVER

2-approximation of VERTEX COVER

Find a maximal matching $M \subseteq E$
Return $S = \{$all endpoints of edges in $M\}$

Euclidean distance $d(x, y)$

Euclidean distance $d(x, y)$

1. $d(x, y) \geq 0$ for all x, y.

Euclidean distance $d(x, y)$

1. $d(x, y) \geq 0$ for all x, y.
2. $d(x, y) = 0$ if and only if $x - y$.

Euclidean distance $d(x, y)$

1. $d(x, y) \geq 0$ for all x, y.
2. $d(x, y) = 0$ if and only if $x - y$.
3. $d(x, y) = d(y, x)$.

Euclidean distance $d(x, y)$

1. $d(x, y) \geq 0$ for all x, y.
2. $d(x, y) = 0$ if and only if $x - y$.
3. $d(x, y) = d(y, x)$.
4. (**Triangle inequality**) $d(x, y) \leq d(x, z) + d(z, y)$.

Clustering

Clustering

k-CLUSTERING

Input: Points $X = \{x_1, \ldots, x_n\}$ with underlying distance metric $d(\cdot, \cdot)$; integer k.

Output: A partition of the points into k clusters $C_1, \ldots, C_k$.

Goal: Minimize the diameter of the cluster

$$\max_j \max_{x_a, x_b \in C_j} d(x_a, x_b).$$

.

Clustering

k-CLUSTERING

Input: Points $X = \{x_1, \ldots, x_n\}$ with underlying distance metric $d(\cdot, \cdot)$; integer k.

Output: A partition of the points into k clusters $C_1, \ldots, C_k$.

Goal: Minimize the diameter of the cluster

$$\max_j \max_{x_a, x_b \in C_j} d(x_a, x_b).$$

.

Theorem

k-CLUSTERING *is* **NP**-*hard.*

2-Approximation

2-Approximation

Pick any point $\mu_1 \in X$ as the first cluster center
for $i = 2$ to k do
    Let μ_i be the point in X that is farthest from $\mu_1, \ldots, \mu_{i-1}$
      (i.e., that maximizes $\min_{j<i} d(\cdot, \mu_j)$)
Create k clusters: $C_i = \{\text{all } x \in X \text{ whose closest center is } mu_i\}$

Proof for the approximation ratio 2:

Proof for the approximation ratio 2:

Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$,

Proof for the approximation ratio 2:

Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$, and r be its distance to its closest center.

Proof for the approximation ratio 2:

Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$, and r be its distance to its closest center.

Then every point in X must be within distance r of its cluster center.

Proof for the approximation ratio 2:

Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$, and r be its distance to its closest center.

Then every point in X must be within distance r of its cluster center. By the triangle inequality, this means that every cluster has diameter at most $2r$.

Proof for the approximation ratio 2:

Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$, and r be its distance to its closest center.

Then every point in X must be within distance r of its cluster center. By the triangle inequality, this means that every cluster has diameter at most $2r$.

we have identified $k+1$ points $\mu_1, \mu_2, \ldots, \mu_k, x\}$ that are all *at a distance at least r from each other*.

Proof for the approximation ratio 2:

Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$, and r be its distance to its closest center.

Then every point in X must be within distance r of its cluster center. By the triangle inequality, this means that every cluster has diameter at most $2r$.

we have identified $k+1$ points $\mu_1, \mu_2, \ldots, \mu_k, x\}$ that are all *at a distance at least r from each other*.

Any partition into k clusters must put two of these points in the same cluster and must therefore have diameter at least r.

□

TSP on metric space

TSP on metric space

The triangle inequality played a crucial role in making the k-CLUSTER problem approximable.

TSP on metric space

The triangle inequality played a crucial role in making the k-CLUSTER problem approximable.

It also helps with the TRAVELING SALESMAN PROBLEM:

TSP on metric space

The triangle inequality played a crucial role in making the k-CLUSTER problem approximable.

It also helps with the TRAVELING SALESMAN PROBLEM: if the distances between cities satisfy the metric properties, then there is an algorithm that outputs a tour of length *at most* 1.5 *times optimal*.

TSP on metric space

The triangle inequality played a crucial role in making the k-CLUSTER problem approximable.

It also helps with the TRAVELING SALESMAN PROBLEM: if the distances between cities satisfy the metric properties, then there is an algorithm that outputs a tour of length *at most* 1.5 *times optimal*.

We'll now look at a slightly weaker result that achieves *a factor of* 2.

Removing any edge from a traveling salesman tour leaves a path through all the vertices, which is *a spanning tree*.

Removing any edge from a traveling salesman tour leaves a path through all the vertices, which is *a spanning tree*.

Therefore,

$$\text{TSP cost} \geq \text{cost of this path} \geq \text{MST cost}$$

Removing any edge from a traveling salesman tour leaves a path through all the vertices, which is *a spanning tree*.

Therefore,

$$\text{TSP cost} \geq \text{cost of this path} \geq \text{MST cost}$$

If we can use each edge twice, then by following the shape of the MST we end up with a tour that visits all the cities, some of them *more than once*.

Removing any edge from a traveling salesman tour leaves a path through all the vertices, which is *a spanning tree*.

Therefore,

$$\text{TSP cost} \geq \text{cost of this path} \geq \text{MST cost}$$

If we can use each edge twice, then by following the shape of the MST we end up with a tour that visits all the cities, some of them *more than once*.

To fix the problem, the tour should simply skip any city it is about to revisit, and instead move directly to the next new city in its list.

Removing any edge from a traveling salesman tour leaves a path through all the vertices, which is *a spanning tree*.

Therefore,

$$\text{TSP cost} \geq \text{cost of this path} \geq \text{MST cost}$$

If we can use each edge twice, then by following the shape of the MST we end up with a tour that visits all the cities, some of them *more than once*.

To fix the problem, the tour should simply skip any city it is about to revisit, and instead move directly to the next new city in its list.
By the triangle inequality, these bypasses can only make the overall tour shorter.