Analysis of Algorithms

Dealing with NP Problems: Intelligent Exponential Search and Approximation Algorithms

Andres Mendez-Vazquez

April 30, 2018

Outline

Outline

The Dilemma

We face the following

- NP problems need solutions in real-life!!!
- We only know exponential algorithms for them!!!
- What do we do?

The Dilemma

We face the following

- NP problems need solutions in real-life!!!
- We only know exponential algorithms for them!!!
- What do we do?

The Dilemma

We face the following

- NP problems need solutions in real-life!!!
- We only know exponential algorithms for them!!!
- What do we do?

Accuracy

Accuracy issues

- NP problems are often optimization problems.
- It's hard to find the EXACT answer.
- Maybe we just want to know if our answer is close to the exact answer.

Accuracy

Accuracy issues

- NP problems are often optimization problems.
- It's hard to find the EXACT answer.
- Maybe we just want to know if our answer is close to the exact answer.

Accuracy

Accuracy issues

- NP problems are often optimization problems.
- It's hard to find the EXACT answer.
- Maybe we just want to know if our answer is close to the exact answer.

Near optimal solutions

Ways of dealing with NP problems

- if the actual input is small, exponential running time may be perfectly satisfactory.
- It may be possible to isolate special cases solvable in polynomial time.

Near optimal solutions

Ways of dealing with NP problems

- if the actual input is small, exponential running time may be perfectly satisfactory.
- It may be possible to isolate special cases solvable in polynomial time.

Ways of dealing with NP problems

- Third, it may be possible to use some form of intelligent search to avoid almost all the time the worst case.
- Fourth, it may still be possible to find near-optimal solutions in polynomial time (either in the worst case or on average).

Near optimal solutions

Ways of dealing with NP problems

- if the actual input is small, exponential running time may be perfectly satisfactory.
- It may be possible to isolate special cases solvable in polynomial time.

Ways of dealing with NP problems

- Third, it may be possible to use some form of intelligent search to avoid almost all the time the worst case.
- Fourth, it may still be possible to find near-optimal solutions in polynomial time (either in the worst case or on average).

Near optimal solutions

Ways of dealing with NP problems

- if the actual input is small, exponential running time may be perfectly satisfactory.
- It may be possible to isolate special cases solvable in polynomial time.

Ways of dealing with NP problems

- Third, it may be possible to use some form of intelligent search to avoid almost all the time the worst case.
- Fourth, it may still be possible to find near-optimal solutions in polynomial time (either in the worst case or on average).

Outline

Three main branches attacking the NP Problems

Intelligent Exponential Search

Procedures that are exponential time in the worst-case, but with the right design can be very efficient on typical instances.

- They are a trade-off between optimality, completeness, accuracy and precision and running time

Three main branches attacking the NP Problems

Intelligent Exponential Search

Procedures that are exponential time in the worst-case, but with the right design can be very efficient on typical instances.

Approximation algorithms

Algorithms guaranteed to find a "near optimal" solution, under a certain bound, in polynomial time.

Three main branches attacking the NP Problems

Intelligent Exponential Search

Procedures that are exponential time in the worst-case, but with the right design can be very efficient on typical instances.

Approximation algorithms

Algorithms guaranteed to find a "near optimal" solution, under a certain bound, in polynomial time.

Heuristics

- Useful algorithmic procedures that provide approximated solutions in polynomial time.
- They are a trade-off between optimality, completeness, accuracy and precision and running time.

Three main branches attacking the NP Problems

Intelligent Exponential Search

Procedures that are exponential time in the worst-case, but with the right design can be very efficient on typical instances.

Approximation algorithms

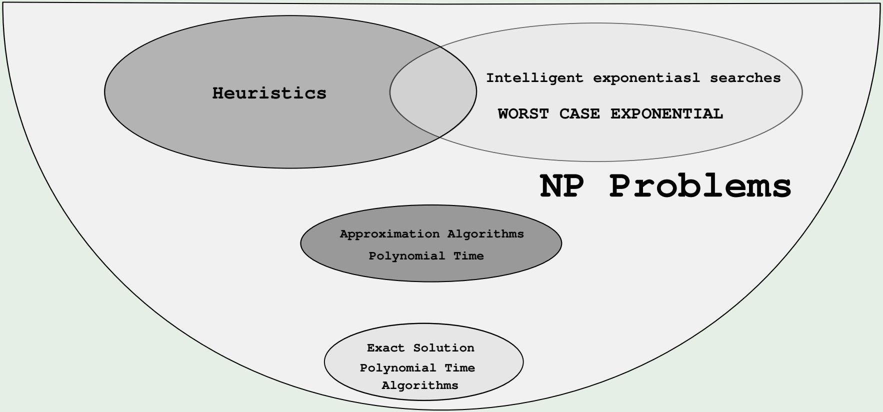
Algorithms guaranteed to find a "near optimal" solution, under a certain bound, in polynomial time.

Heuristics

- Useful algorithmic procedures that provide approximated solutions in polynomial time.
- They are a trade-off between optimality, completeness, accuracy and precision and running time.

We have something like this

As sets



Outline

Backtracking

Something Notable

Backtracking is based on that it is often possible to reject a solution by looking at just a small portion of it.

Example

If an instance of SAT contains the clause $C_i = (x_1 \vee x_2)$, then all assignments with $x_1 = x_2 = 0$ can be instantly eliminated.

Backtracking

Something Notable

Backtracking is based on that it is often possible to reject a solution by looking at just a small portion of it.

Example

If an instance of SAT contains the clause $C_i = (x_1 \vee x_2)$, then all assignments with $x_1 = x_2 = 0$ can be instantly eliminated.

Outline

Example

Pruning Example

Given the possible values that you can give to two literals:

x_1	x_2
1	1
1	0
0	1
0	**0**

It is possible to prune a quarter of the entire search space... Can this be systematically exploited?

An example of exploiting this idea in SAT solvers

Consider the following Boolean formula $\phi(w, x, y, z)$

$$(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$$

We start branching in one variable, we can choose w

Note: This selection does not violate any of the clauses of $\phi(w, x, y, z)$

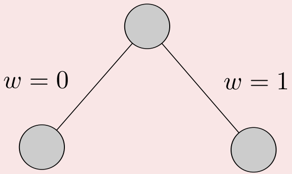
An example of exploiting this idea in SAT solvers

Consider the following Boolean formula $\phi(w, x, y, z)$

$(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$

We start branching in one variable, we can choose w

Initial formula ϕ

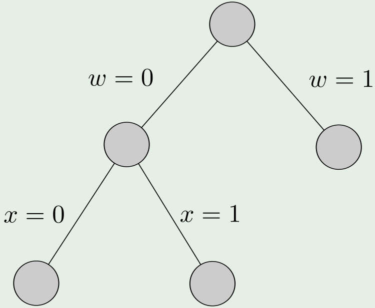


Note: This selection does not violate any of the clauses of $\phi(w, x, y, z)$

Now

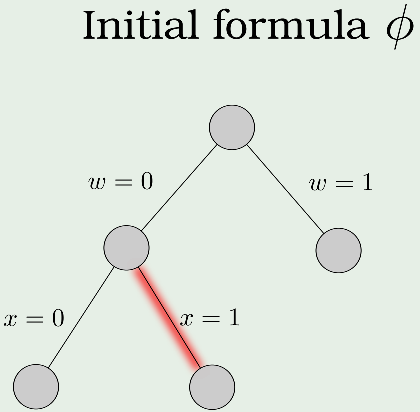
The partial assignment $w = 0, x = 1$ violates the clause $(w \lor \neg x)$

Initial formula ϕ



Now

Then, we prune that branch



In addition

What if $w = 0, x = 0$

Then, the following clauses are satisfied

1. $\neg w = 1$
2. $\neg x = 1$

Thus, we have the following left

1. Before
   1. $(w \lor x \lor y \lor z) \land (w \lor \neg x) \land (x \lor \neg y) \land (y \lor \neg z) \land (z \lor \neg w) \land (\neg w \lor \neg z)$
2. After
   1. $(0 \lor 0 \lor y \lor z) \land (0 \lor 1) \land (0 \lor \neg y) \land (y \lor \neg z) \land (z \lor 1) \land (1 \lor \neg z)$

In addition

What if $w = 0, x = 0$

Then, the following clauses are satisfied

1. $\neg w = 1$
2. $\neg x = 1$

Thus, we have the following left

1. Before
   1. $(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
2. After
   1. $(0 \vee 0 \vee y \vee z) \wedge (0 \vee 1) \wedge (0 \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee 1) \wedge (1 \vee \neg z)$

Finally

We have the following reduced number of equations

$(y \vee z), (1), (\neg y), (y \vee \neg z), (1), (1) \Leftrightarrow \boldsymbol{(y \vee z), (\neg y), (y \vee \neg z)}$

What if $w = 0, x = 1$

1. Before
   1. $(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
2. After
   1. $(1) \wedge (0) \wedge (1) \wedge (y \vee \neg z) \wedge (1) \wedge (1)$

Finally

We have the following reduced number of equations

$(y \vee z), (1), (\neg y), (y \vee \neg z), (1), (1) \Leftrightarrow \boldsymbol{(y \vee z), (\neg y), (y \vee \neg z)}$

What if $w = 0, x = 1$

1. Before
   1. $(w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
2. After
   1. $(1) \wedge (0) \wedge (1) \wedge (y \vee \neg z) \wedge (1) \wedge (1)$

Thus

We have something no satisfiable

$$(1) \wedge (0) \wedge (1) \wedge (y \vee \neg z) \wedge (1) \wedge (1) \Leftrightarrow (), (y \vee \neg z)$$

Clearly

We prune that part of the search tree.

Note: we use "()≡(0)" to point out to a "empty clause" ruling out satisfiability.

Thus

We have something no satisfiable

$$(1) \wedge (0) \wedge (1) \wedge (y \vee \neg z) \wedge (1) \wedge (1) \Leftrightarrow (), (y \vee \neg z)$$

Clearly

We prune that part of the search tree.

Note we use "()≡(0)" to point out to a "empty clause" ruling out satisfiability.

The decisions we need to make in backtracking

First

Which subproblem to expand next.

Second

Which branching variable to use.

Remark

The benefit of backtracking lies in its ability to eliminate portions of the search space.

The decisions we need to make in backtracking

First

Which subproblem to expand next.

Second

Which branching variable to use.

Remark

The benefit of backtracking lies in its ability to eliminate portions of the search space.

The decisions we need to make in backtracking

First

Which subproblem to expand next.

Second

Which branching variable to use.

Remark

The benefit of backtracking lies in its ability to eliminate portions of the search space.

Choosing

Something Notable

A classic strategy:

- You choose the subproblem that contains the smallest clause.
- Then, you branch on a variable in that clause.

Choosing

Something Notable

A classic strategy:

- You choose the subproblem that contains the smallest clause.
- Then, you branch on a variable in that clause.

Then

If the clause is a singleton then at least one of the resulting branches will be terminated.

Choosing

Something Notable

A classic strategy:

- You choose the subproblem that contains the smallest clause.
- Then, you branch on a variable in that clause.

Then

If the clause is a singleton then at least one of the resulting branches will be terminated.

Choosing

Something Notable

A classic strategy:

- You choose the subproblem that contains the smallest clause.
- Then, you branch on a variable in that clause.

Then

If the clause is a singleton then at least one of the resulting branches will be terminated.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

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

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

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

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

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

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

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

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

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

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

The Backtracking Test

The test needs to look at the subproblem to declare quickly if

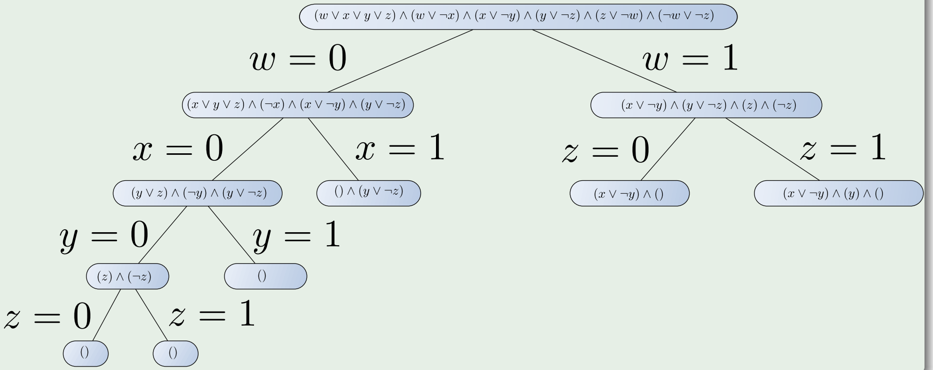
1. **Failure**: the subproblem has no solution.
2. **Success**: a solution to the subproblem is found.
3. **Uncertainty**.

What about SAT

- The test declares failure if there is an empty clause
- The test declares success if there are no clauses
- Uncertainty Otherwise.

Example

We have the following



Outline

Pseudo-code for Backtracking

We have

BACKTRACKING(P_0)

1. Start with some problem P_0
2. Let $\mathcal{S} = \{P_0\}$, the set if active subproblems
3. While $\mathcal{S} \neq \emptyset$
4. **choose** a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
5. **expand** it into smaller subproblems $P_1, P_2, ..., P_k$
6. For each P_i
7. if test(P_i) succeeds: halt and return the branch solution
8. if test(P_i) fails: discard P_i
9. Otherwise: add P_i to $\mathcal{S}$
10. return there is no solution.

Pseudo-code for Backtracking

We have

BACKTRACKING(P_0)

1. Start with some problem P_0
2. Let $\mathcal{S} = \{P_0\}$, the set if active subproblems
3. While $\mathcal{S} \neq \emptyset$
4. **choose** a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
5. **expand** it into smaller subproblems $P_1, P_2, ..., P_k$
6. For each P_i
7. if test(P_i) succeeds: halt and return the branch solution
8. if test(P_i) fails: discard P_i
9. Otherwise: add P_i to $\mathcal{S}$
10. return there is no solution.

Pseudo-code for Backtracking

We have

BACKTRACKING(P_0)

1. Start with some problem P_0
2. Let $\mathcal{S} = \{P_0\}$, the set if active subproblems
3. While $\mathcal{S} \neq \emptyset$
4. **choose** a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
5. **expand** it into smaller subproblems $P_1, P_2, ..., P_k$
6. For each P_i
7. if test(P_i) succeeds: halt and return the branch solution
8. if test(P_i) fails: discard P_i
9. Otherwise: add P_i to $\mathcal{S}$
10. return there is no solution.

Pseudo-code for Backtracking

We have

BACKTRACKING(P_0)

1. Start with some problem P_0
2. Let $\mathcal{S} = \{P_0\}$, the set if active subproblems
3. While $\mathcal{S} \neq \emptyset$
4. **choose** a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
5. **expand** it into smaller subproblems $P_1, P_2, ..., P_k$
6. For each P_i
7. if test(P_i) succeeds: halt and return the branch solution
8. if test(P_i) fails: discard P_i
9. Otherwise: add P_i to $\mathcal{S}$
10. return there is no solution.

Choose and Expand

For SAT

1. The choose procedure picks a clause,
2. Expand picks a variable within that clause.

There has been already

A discussion on how to make such choices.

Choose and Expand

For SAT

1. The choose procedure picks a clause,
2. Expand picks a variable within that clause.

There has been already

A discussion on how to make such choices.

Notes

With the right test, expand, and choose routines

- Backtracking can be remarkably effective in practice

Further

- The backtracking algorithm we showed for SAT is the basis of many successful satisfiability programs

For Example, 2SAT problems

- It is a conjunction (a Boolean and operation) of clauses,
- Where each clause is a disjunction (a Boolean or operation) of two variables or negated variables.

cinvestav
Unidad Guadalajara

Notes

With the right test, expand, and choose routines

- Backtracking can be remarkably effective in practice

Further

- The backtracking algorithm we showed for SAT is the basis of many successful satisfiability programs

For Example, 2SAT problems

- It is a conjunction (a Boolean and operation) of clauses,
- Where each clause is a disjunction (a Boolean or operation) of two variables or negated variables.

Notes

With the right test, expand, and choose routines

- Backtracking can be remarkably effective in practice

Further

- The backtracking algorithm we showed for SAT is the basis of many successful satisfiability programs

For Example, 2SAT problems

- It is a conjunction (a Boolean and operation) of clauses,
- Where each clause is a disjunction (a Boolean or operation) of two variables or negated variables.

Then

Backtracking

- If presented with a 2 SAT instance,
  - it will always find a satisfying assignment, if one exists, in polynomial time!!!

Something Notable

- Therefore, we depend on the constraints!!!

These problems are known as

- Constraint Satisfaction Problems!!!

Then

Backtracking

- If presented with a 2 SAT instance,
  - it will always find a satisfying assignment, if one exists, in polynomial time!!!

Something Notable

- Therefore, we depend on the constraints!!!

These problems are known as

- Constraint Satisfaction Problems!!!

Then

Backtracking

- If presented with a 2 SAT instance,
  - it will always find a satisfying assignment, if one exists, in polynomial time!!!

Something Notable

- Therefore, we depend on the constraints!!!

These problems are known as

- Constraint Satisfaction Problems!!!

Outline

Another Exponential Search

Branch-and-Bound

This is a search better fitted to optimization problems

Another Exponential Search

Branch-and-Bound

This is a search better fitted to optimization problems

How do we reject subproblems?

First, imagine a minimization problem

- To reject a subproblem, its cost must exceeds the best cost found so far.
- PROBLEM!!! This is unknown!!!
- Thus, we use a quick *lower bound* on this cost.

Another Exponential Search

Branch-and-Bound

This is a search better fitted to optimization problems

How do we reject subproblems?

First, imagine a minimization problem

- To reject a subproblem, its cost must exceeds the best cost found so far.
- PROBLEM!!! This is unknown!!!
- Thus, we use a quick *lower bound* on this cost.

Another Exponential Search

Branch-and-Bound

This is a search better fitted to optimization problems

How do we reject subproblems?

First, imagine a minimization problem

- To reject a subproblem, its cost must exceeds the best cost found so far.
- PROBLEM!!! This is unknown!!!
- Thus, we use a quick *lower bound* on this cost.

Another Exponential Search

Branch-and-Bound

This is a search better fitted to optimization problems

How do we reject subproblems?

First, imagine a minimization problem

- To reject a subproblem, its cost must exceeds the best cost found so far.
- PROBLEM!!! This is unknown!!!
- Thus, we use a quick *lower bound* on this cost.

Lower Bound

We use the info in the problem

To design an efficient way to approximate the solution.

Outline

Pseudo-code for Branch-and-Bound

We have

BRANCH-AND-BOUND(P_0)

1. *Start with some problem* P_0
2. *Let* $\mathcal{S} = \{P_0\}$*, the set if active subproblems*
3. *bestsofar*$=\infty$
4. *While* $\mathcal{S} \neq \emptyset$
5. **choose** *a subproblem (Partial Solution)* $P \in \mathcal{S}$ *and remove it from* $\mathcal{S}$
6. **expand** *it into smaller subproblems* $P_1, P_2, ..., P_k$
7. *For each* P_i
8. *if* P_i *is a complete solution:*
9. *update bestsofar*
10. *else*
11. *if lowerbound*(P_i) $<$*bestsofar: add* P_i *to* $\mathcal{S}$
12. *return bestsofar*

Pseudo-code for Branch-and-Bound

We have

BRANCH-AND-BOUND(P_0)

1. *Start with some problem* P_0
2. *Let* $\mathcal{S} = \{P_0\}$*, the set if active subproblems*
3. *bestsofar*$=\infty$
4. *While* $\mathcal{S} \neq \emptyset$
5.   ***choose*** *a subproblem (Partial Solution)* $P \in \mathcal{S}$ *and remove it from* $\mathcal{S}$
6.   ***expand*** *it into smaller subproblems* $P_1, P_2, ..., P_k$
7.   *For each* P_i
8.   *if* P_i *is a complete solution:*
9.   *update bestsofar*
10.   *else*
11.   *if lowerbound*(P_i) $<$*bestsofar: add* P_i *to* $\mathcal{S}$
12. *return bestsofar*

Pseudo-code for Branch-and-Bound

We have

BRANCH-AND-BOUND(P_0)

1. *Start with some problem* P_0
2. *Let* $\mathcal{S} = \{P_0\}$*, the set if active subproblems*
3. *bestsofar*$=\infty$
4. *While* $\mathcal{S} \neq \emptyset$
5.   ***choose*** *a subproblem (Partial Solution)* $P \in \mathcal{S}$ *and remove it from* $\mathcal{S}$
6.   ***expand*** *it into smaller subproblems* $P_1, P_2, ..., P_k$
7.   *For each* P_i
8.   *if* P_i *is a complete solution:*
9.   *update bestsofar*
10.   *else*
11.   *if lowerbound*(P_i) $<$*bestsofar: add* P_i *to* $\mathcal{S}$
12. *return bestsofar*

Pseudo-code for Branch-and-Bound

We have

BRANCH-AND-BOUND(P_0)

1. *Start with some problem* P_0
2. *Let* $\mathcal{S} = \{P_0\}$*, the set if active subproblems*
3. *bestsofar*$=\infty$
4. *While* $\mathcal{S} \neq \emptyset$
5.   ***choose*** *a subproblem (Partial Solution)* $P \in \mathcal{S}$ *and remove it from* $\mathcal{S}$
6.   ***expand*** *it into smaller subproblems* $P_1, P_2, ..., P_k$
7.   *For each* P_i
8.   *if* P_i *is a complete solution:*
9.   *update bestsofar*
10.   *else*
11.   *if lowerbound*(P_i) $<$*bestsofar: add* P_i *to* $\mathcal{S}$
12. *return bestsofar*

Pseudo-code for Branch-and-Bound

We have

BRANCH-AND-BOUND(P_0)

Start with some problem P_0
Let $\mathcal{S} = \{P_0\}$, *the set if active subproblems*
bestsofar$=\infty$
While $\mathcal{S} \neq \emptyset$
    choose *a subproblem (Partial Solution)* $P \in \mathcal{S}$ *and remove it from* $\mathcal{S}$
    expand *it into smaller subproblems* $P_1, P_2, ..., P_k$
    For each P_i
        if P_i *is a complete solution:*
            update bestsofar
        else
            if lowerbound(P_i) $<$*bestsofar: add* P_i *to* $\mathcal{S}$
return bestsofar

Outline

Example: The Traveling Salesman Problem

A partial solution to the TSP

It is a simple path $a \rightsquigarrow b$ passing through some vertices's $S \subseteq V$ with $a, b \in S$.

- Denote this as $[a, S, b]$

The subproblem

It is necessary to find the best completion of the tour i.e. the cheapest complementary path $b \rightsquigarrow a$ with intermediate vertices in $V - S$.

Example: The Traveling Salesman Problem

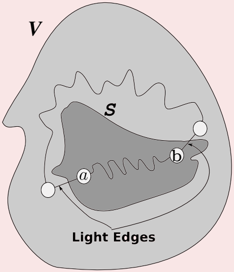
A partial solution to the TSP

It is a simple path $a \rightsquigarrow b$ passing through some vertices's $S \subseteq V$ with $a, b \in S$.

- Denote this as $[a, S, b]$

The subproblem

It is necessary to find the best completion of the tour i.e. the cheapest complementary path $b \rightsquigarrow a$ with intermediate vertices in $V - S$.



How do we establish a lower bound?

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

- A partial solution is a simple path $a \rightsquigarrow b$
  - Which pass through a series of vertices's $S \subseteq V$

Here

- S includes the endpoints a and b.

We denote such solution as

- We extend a particular solution $[a, S, b]$
  - in fact, a will be fixed throughout the algorithm

How do we establish a lower bound?

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

- A partial solution is a simple path $a \rightsquigarrow b$
  - Which pass through a series of vertices's $S \subseteq V$

Here

- S includes the endpoints a and b.

We denote such solution as

- We extend a particular solution $[a, S, b]$
  - in fact, a will be fixed throughout the algorithm

How do we establish a lower bound?

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

- A partial solution is a simple path $a \rightsquigarrow b$
  - Which pass through a series of vertices's $S \subseteq V$

Here

- S includes the endpoints a and b.

We denote such solution as

- We extend a particular solution $[a, S, b]$
  - in fact, a will be fixed throughout the algorithm

Then

The Subproblem

- The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path

$$b \rightsquigarrow y \rightsquigarrow a, \text{ such that } y \in V - S \text{ (Intermediate Vertex)}$$

Initial problem is the form

$$[a, \{a\}, a] \textbf{ For any } a \in V$$

At each step of the branch-and-bound

- We extend a particular solution $[a, S, b]$ by a single edge (b, x) where $x \in V - S$

Then

The Subproblem

- The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path

$$b \rightsquigarrow y \rightsquigarrow a, \text{ such that } y \in V - S \text{ (Intermediate Vertex)}$$

Initial problem is the form

$$[a, \{a\}, a] \ \textbf{For any } a \in V$$

At each step of the branch-and-bound

- We extend a particular solution $[a, S, b]$ by a single edge (b, x) where $x \in V - S$

Then

The Subproblem

- The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path

$$b \rightsquigarrow y \rightsquigarrow a, \text{ such that } y \in V - S \text{ (Intermediate Vertex)}$$

Initial problem is the form

$$[a, \{a\}, a] \textbf{ For any } a \in V$$

At each step of the branch-and-bound

- We extend a particular solution $[a, S, b]$ by a single edge (b, x) where $x \in V - S$

Thus, Given $a \rightsquigarrow b$

Leading to $|V - S|$ subproblems

- Of the form $[a, S \cup \{x\}, x]$

  - However, we will use a simpler one!!!

Thus, Given $a \rightsquigarrow b$

Leading to $|V - S|$ subproblems

- Of the form $[a, S \cup \{x\}, x]$

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

- Many sophisticated methods have been developed for this,
  - However, we will use a simple one!!!

Thus, Given $a \rightsquigarrow b$

Leading to $|V - S|$ subproblems

- Of the form $[a, S \cup \{x\}, x]$

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

- Many sophisticated methods have been developed for this,
  - However, we will use a simple one!!!

The Simple Strategy

The remainder of the tour consists

- A path through $V - S +$ edges from a and b to $V - S$

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

The Simple Strategy

The remainder of the tour consists

- A path through $V-S$ + edges from a and b to $V-S$

Therefore

We use the simple strategy, the lower-bound is the sum of the following quantities:

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

The Simple Strategy

The remainder of the tour consists

- A path through $V - S +$ edges from a and b to $V - S$

Therefore

We use the simple strategy, the lower-bound is the sum of the following quantities:

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

The Simple Strategy

The remainder of the tour consists

- A path through $V - S$ + edges from a and b to $V - S$

Therefore

We use the simple strategy, the lower-bound is the sum of the following quantities:

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

The Simple Strategy

The remainder of the tour consists

- A path through $V - S$ + edges from a and b to $V - S$

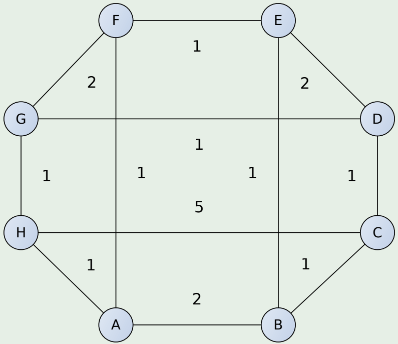
Therefore

We use the simple strategy, the lower-bound is the sum of the following quantities:

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

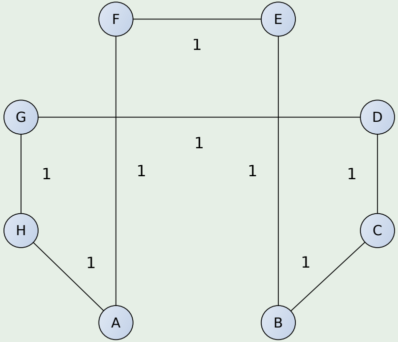
Example

A graph where a brute force approach will take 7!=5,040



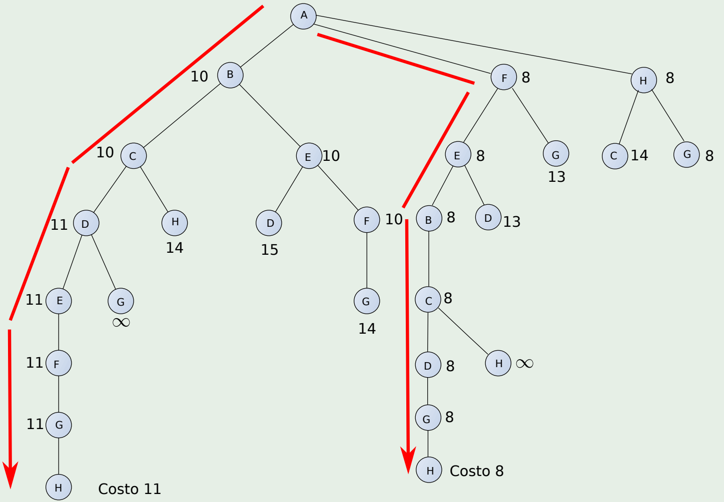
Example

TSP Solution



Example

Branch and Bound Solution



This is a quite simple lower bound

Clearly

- The lightest edges and the minimum spanning tree is a good estimation

Something Notable

- Notice how just 28 partial solutions are considered

Instead of

- The $7! = 5{,}040$ that would arise in a brute-force search.

This is a quite simple lower bound

Clearly

- The lightest edges and the minimum spanning tree is a good estimation

Something Notable

- Notice how just 28 partial solutions are considered

Instead of

- The $7! = 5{,}040$ that would arise in a brute-force search.

This is a quite simple lower bound

Clearly

- The lightest edges and the minimum spanning tree is a good estimation

Something Notable

- Notice how just 28 partial solutions are considered

Instead of

- The $7! = 5{,}040$ that would arise in a brute-force search.

Outline

Approximation Algorithms

Remarks

- In practice, near-optimality is often good.
- An algorithm that returns near-optimal solutions is called an approximation algorithm.

Approximation Algorithms

Remarks

- In practice, near-optimality is often good.
- An algorithm that returns near-optimal solutions is called an approximation algorithm.

Definition of approximation algorithms

Framework

Suppose that we are working on an optimization problem in which each potential solution has a positive cost.

- We wish to find a near-optimal solution.

Definition of approximation algorithms

Framework

Suppose that we are working on an optimization problem in which each potential solution has a positive cost.

- We wish to find a near-optimal solution.

Thus

- Max cost for a maximization problem.
- Min cost for a minimization problem.

Definition of approximation algorithms

Framework

Suppose that we are working on an optimization problem in which each potential solution has a positive cost.

- We wish to find a near-optimal solution.

Thus

- Max cost for a maximization problem.
- Min cost for a minimization problem.

Definition of approximation algorithms

Framework

Suppose that we are working on an optimization problem in which each potential solution has a positive cost.

- We wish to find a near-optimal solution.

Thus

- Max cost for a maximization problem.
- Min cost for a minimization problem.

Definition of approximation algorithms

Framework

Suppose that we are working on an optimization problem in which each potential solution has a positive cost.

- We wish to find a near-optimal solution.

Thus

- Max cost for a maximization problem.
- Min cost for a minimization problem.

Then

Definition

We say that an algorithm for a problem has an approximation ratio of $\rho(n)$ if:

- For any input of size n the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n)$$

Then

Definition

We say that an algorithm for a problem has an approximation ratio of $\rho(n)$ if:

- For any input of size n the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n)$$

Then

Definition

We say that an algorithm for a problem has an approximation ratio of $\rho(n)$ if:

- For any input of size n the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n)$$

Definition of approximation algorithms

Definition

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-**approximation algorithm.**

Further

The definitions of the approximation ratio and of a $\rho(n)$-**approximation algorithm** apply to both minimization and maximization problems.

Definition of approximation algorithms

Definition

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-**approximation algorithm.**

Further

The definitions of the approximation ratio and of a $\rho(n)$-**approximation algorithm** apply to both minimization and maximization problems.

The two possibilities

For a maximization problem

We have that $0 < C \leq C^*$, then the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

For a minimization problem

We have that $0 < C^* \leq C$, then the ratio C/C^* gives the factor by which the approximate solution is larger than the cost of an optimal solution.

Properties

The approximation ratio of an approximation algorithm is never less than 1.

The two possibilities

For a maximization problem

We have that $0 < C \leq C^*$, then the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

For a minimization problem

We have that $0 < C^* \leq C$, then the ratio C/C^* gives the factor by which the approximate solution is larger than the cost of an optimal solution.

Properties

The approximation ratio of an approximation algorithm is never less than 1.

The two possibilities

For a maximization problem

We have that $0 < C \leq C^{*}$, then the ratio ${C^{*}}/{C}$ gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

For a minimization problem

We have that $0 < C^{*} \leq C$, then the ratio ${C}/{C^{*}}$ gives the factor by which the approximate solution is larger than the cost of an optimal solution.

Properties

The approximation ratio of an approximation algorithm is never less than 1.

Some characteristics of approximation algorithms

Approximation algorithms have the following characteristics

- They have polynomial times.
- They are not guarantee to obtain optimal solutions.
- However, they guarantee good solution within some factor of the optimum.

Some characteristics of approximation algorithms

Approximation algorithms have the following characteristics

- They have polynomial times.
- They are not guarantee to obtain optimal solutions.
- However, they guarantee good solution within some factor of the optimum.

Further

- They often use algorithms from related problems as subroutines.

Some characteristics of approximation algorithms

Approximation algorithms have the following characteristics

- They have polynomial times.
- They are not guarantee to obtain optimal solutions.
- However, they guarantee good solution within some factor of the optimum.

Further

- They often use algorithms from related problems as subroutines.

Some characteristics of approximation algorithms

Approximation algorithms have the following characteristics

- They have polynomial times.
- They are not guarantee to obtain optimal solutions.
- However, they guarantee good solution within some factor of the optimum.

Further

- They often use algorithms from related problems as subroutines.

Some characteristics of approximation algorithms

Approximation algorithms have the following characteristics

- They have polynomial times.
- They are not guarantee to obtain optimal solutions.
- However, they guarantee good solution within some factor of the optimum.

Further

- They often use algorithms from related problems as subroutines.

We will look at...

Examples

- Vertex Cover problem.
- The Traveling Salesman Problem.

We will look at...

Examples

- Vertex Cover problem.
- The Traveling Salesman Problem.

This barely scratches

All the theory of approximation algorithms.

We will look at...

Examples

- Vertex Cover problem.
- The Traveling Salesman Problem.

This barely scratches

All the theory of approximation algorithms.

Outline

Outline

Vertex Cover Problem

Definition of a vertex cover

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G. then either $u \in V'$ or $v \in V'$ (Or both).

Thus

The size of a vertex cover is the number of vertices in it.

Vertex Cover Problem

Definition of a **vertex cover**

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G. then either $u \in V'$ or $v \in V'$ (Or both).

Thus

The size of a vertex cover is the number of vertices in it.

Vertex Cover Problem

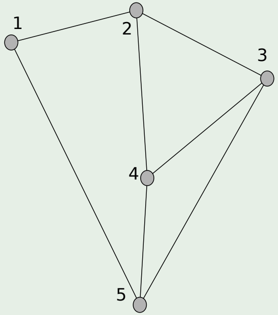
Definition

The vertex-cover problem is to find a vertex cover of minimum size in a given undirected graph.

Example of Vertex Cover Problem

Example

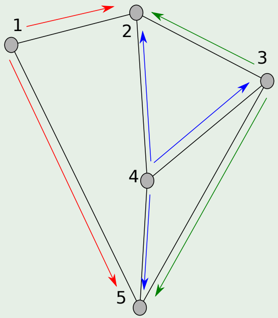
Determine the smallest subset of vertex that "Cover" the graph on the right.



Vertex cover

Example

Determine the smallest subset of vertex that "Cover" the graph on the right.



ANSWER: $\{1, 3, 4\}$

Approximation vertex cover algorithm

Now, we have

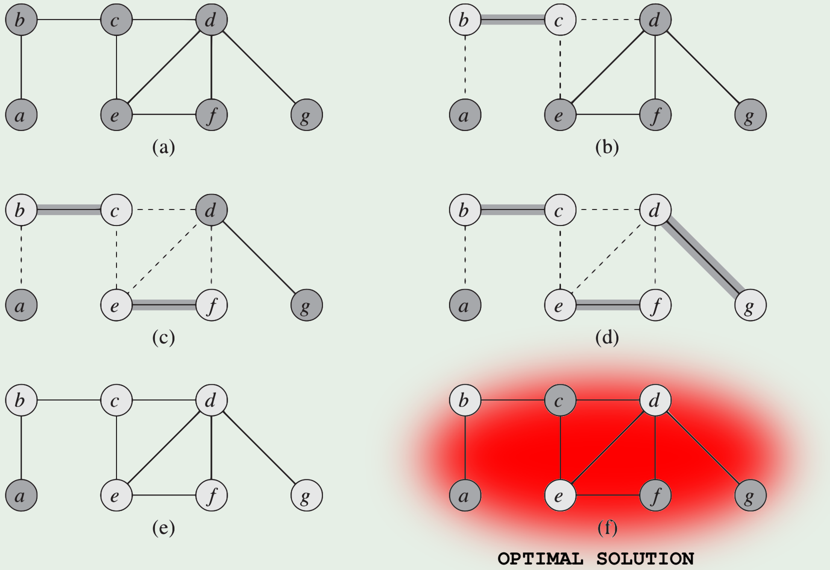
APPROX-VERTEX-COVER(G)

1. $C = \emptyset$
2. $E' = G.E$
3. while $E' \neq \emptyset$
4. let (u, v) be an arbitrary edge of E'
5. $C = C \cup \{u, v\}$
6. remove from E' every edge incident on either u or v
7. return C

Complexity $O(V + E)$

Example

We have



Theorem

Theorem 35.1

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Theorem

Theorem 35.1

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof

- We know that the algorithms return a vertex cover until it finishes, and it runs in poly-time.
- Now, we only need to prove that APPROX-VERTEX-COVER returns a vertex cover of at most twice the size of the optimal cover.

Theorem

Theorem 35.1

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof

- We know that the algorithms return a vertex cover until it finishes, and it runs in poly-time.
- Now, we only need to prove that APPROX-VERTEX-COVER returns a vertex cover of at most twice the size of the optimal cover.

Proof

First

- We call C as the set of vertices that is returned by the approximation algorithm.
- Now, given the set of edges A produced by the algorithm.
- Now, given an optimal cover C^*:
  - C^* must include at least one endpoint of each edge in A.
  - No two edges in A share an endpoint, thus no two edges are covered by the same vertex from C^*:

$$|C^*| \geq |A|$$

Proof

First

- We call C as the set of vertices that is returned by the approximation algorithm.
- Now, given the set of edges A produced by the algorithm.
- Now, given an optimal cover C^*:
  - C^* must include at least one endpoint of each edge in A.
  - No two edges in A share an endpoint, thus no two edges are covered by the same vertex from C^*:

$$|C^*| \geq |A|$$

Proof

First

- We call C as the set of vertices that is returned by the approximation algorithm.
- Now, given the set of edges A produced by the algorithm.
- Now, given an optimal cover C^*:
  - C^* must include at least one endpoint of each edge in A.
  - No two edges in A share an endpoint, thus no two edges are covered by the same vertex from C^*:

$$|C^*| \geq |A|$$

Proof

First

- We call C as the set of vertices that is returned by the approximation algorithm.
- Now, given the set of edges A produced by the algorithm.
- Now, given an optimal cover C^*:
  - C^* must include at least one endpoint of each edge in A.
  - No two edges in A share an endpoint, thus no two edges are covered by the same vertex from C^*:

$$|C^*| \geq |A|$$

Proof

First

- We call C as the set of vertices that is returned by the approximation algorithm.
- Now, given the set of edges A produced by the algorithm.
- Now, given an optimal cover C^*:
  - C^*must include at least one endpoint of each edge in A.
  - No two edges in A share an endpoint, thus no two edges are covered by the same vertex from C^*:

$$|C^*| \geq |A|$$

Proof

Second

Now, each execution in line 4 picks an edge for which neither of its endpoints is already in C, thus we have

$$
\begin{aligned}
|C| &= 2|A| \\
&\leq 2|C^*|
\end{aligned}
$$

Proof

Finally, we have

$$\frac{|C|}{|C^*|} \leq 2$$

Remarks

Did you notice the trick?

We do not know the size of C^*, but we obtain a lower bound for it

Actually

The set A is a **maximal matching** in the graph G.

Remarks

Did you notice the trick?

We do not know the size of C^*, but we obtain a lower bound for it

Actually

The set A is a **maximal matching** in the graph G.

What is a Maximal Matching?

Definition

Given a graph $G = (V, E)$, a matching M in G is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex.

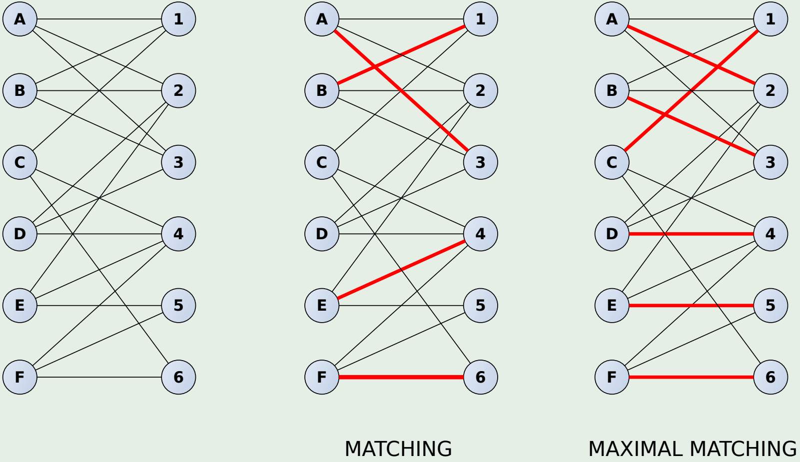
Maximal Matching

Definition

A Maximal Matching is a matching M of a graph G with the property that if any edge not in M is added to M, it is no longer a matching.

Example

Here, we can see an example



Outline

The Traveling-Salesman Problem

Given a complete undirected graph $G = (V, E)$

- With a no-negative integer cost function $c(u, v)$ associated to each edge $(u, v) \in E$
- We need to find a Hamiltonian cycle of G with minimum cost.

The Traveling-Salesman Problem

Given a complete undirected graph $G = (V, E)$

- With a no-negative integer cost function $c(u, v)$ associated to each edge $(u, v) \in E$
- We need to find a Hamiltonian cycle of G with minimum cost.

Extra notation

As an extension of our notation

Let $c(A)$ denote the total cost of the edges in a subset $A \subseteq E$

$$c(A) = \sum_{(u,v) \in A} c(u, v)$$

The extra assumption: The triangle inequality

Assume

:

We will assume that the cost function satisfies the triangle inequality for all its vertices $u, v, w \in V$ then:

$$c(u, w) \leq c(u, v) + c(v, w)$$

The 2-approximation algorithm

Pseudocode

APPROX-TSP-TOUR(G, c, r)

1. select a vertex $r \in G.V$ to be a "root" vertex.
2. compute a minimum spanning tree T for G from root r using the MST-PRIM(G, c, r).
3. Let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T.
4. return the Hamiltonian cycle H

The 2-approximation algorithm

Pseudocode

APPROX-TSP-TOUR(G, c, r)

1. select a vertex $r \in G.V$ to be a "root" vertex.
2. compute a minimum spanning tree T for G from root r using the MST-PRIM(G, c, r).
3. Let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T.
4. return the Hamiltonian cycle H

The 2-approximation algorithm

Pseudocode

APPROX-TSP-TOUR(G, c, r)

1. select a vertex $r \in G.V$ to be a "root" vertex.
2. compute a minimum spanning tree T for G from root r using the MST-PRIM(G, c, r).
3. Let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T.
4. return the Hamiltonian cycle H

The 2-approximation algorithm

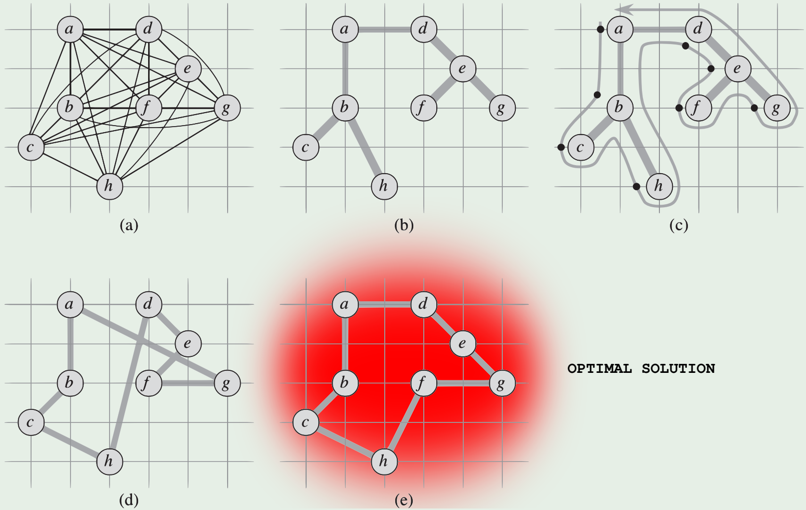
Pseudocode

APPROX-TSP-TOUR(G, c, r)

1. select a vertex $r \in G.V$ to be a "root" vertex.
2. compute a minimum spanning tree T for G from root r using the MST-PRIM(G, c, r).
3. Let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T.
4. return the Hamiltonian cycle H

Example

The root is a



(a) (b) (c) (d) (e)

Theorem that supports the claim

Theorem 35.2

APROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling salesman problem with the triangle inequality.

Proof

First

- Let H^* denote an optimal tour for a set of vertices,
  - H^* is a cycle.
  - If from H^* we erase an edge, H^* becomes a tree.
- Let T the minimum spanning tree computed at line 2

Then:

$$c(T) \leq c(H^*)$$

Proof

First

- Let H^* denote an optimal tour for a set of vertices,
  - H^* is a cycle.
  - If from H^* we erase an edge, H^* becomes a tree.
- Let T the minimum spanning tree computed at line 2

Then:

$$c(T) \leq c(H^*)$$

Proof

First

- Let H^* denote an optimal tour for a set of vertices,
  - H^* is a cycle.
  - If from H^* we erase an edge, H^* becomes a tree.
- Let T the minimum spanning tree computed at line 2

Then:

$$c(T) \leq c(H^*)$$

Proof

First

- Let H^* denote an optimal tour for a set of vertices,
  - H^* is a cycle.
  - If from H^* we erase an edge, H^* becomes a tree.
- Let T the minimum spanning tree computed at line 2

Then:

$$c(T) \leq c(H^*)$$

Proof

First

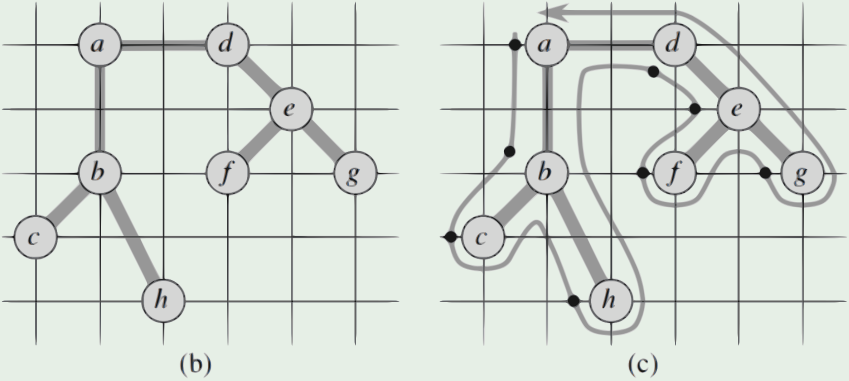
- Let H^* denote an optimal tour for a set of vertices,
  - H^* is a cycle.
  - If from H^* we erase an edge, H^* becomes a tree.
- Let T the minimum spanning tree computed at line 2

Then:

$$c(T) \leq c(H^*)$$

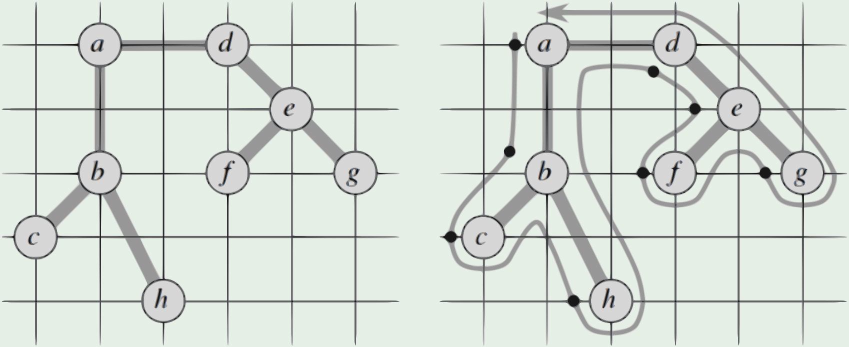
Full walk

A full walk W of T lists how many times a vertex has been visited



Full walk

A full walk $W = a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$



(b) (c)

Then, we have

Since the full walk traverses every edge of T exactly twice

$$c(W) \leq 2c(T)$$

In addition

However, W is not a tour... What can we do?

Then, we have

Since the full walk traverses every edge of T exactly twice

$$c(W) \leq 2c(T)$$

In addition

However, W is not a tour... What can we do?

Use triangle inequality

Something Notable

By the triangle inequality, however, we can delete a visit to any vertex from W.

Use triangle inequality

Something Notable

By the triangle inequality, however, we can delete a visit to any vertex from W.

IMPORTANT

Using the triangle inequality the cost does not increase.

- The resulting ordering specifies going directly from u and v

Use triangle inequality

Something Notable

By the triangle inequality, however, we can delete a visit to any vertex from W.

IMPORTANT

Using the triangle inequality the cost does not increase.

Deletion of vertices

- Delete a vertex v from W between visits to u and w.
  - The resulting ordering specifies going directly from u and w.

Use triangle inequality

Something Notable

By the triangle inequality, however, we can delete a visit to any vertex from W.

IMPORTANT

Using the triangle inequality the cost does not increase.

Deletion of vertices

- Delete a vertex v from W between visits to u and w.
  - The resulting ordering specifies going directly from u and w.

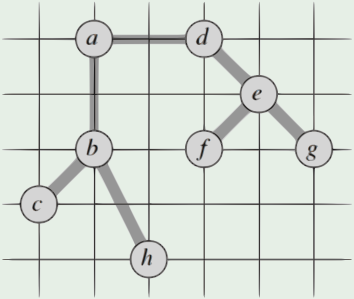
Thus

We get

a, b, c, h, d, e, f, g

Minimum Spanning Tree

From our example



(b)

Notice the following

We have

The previous visiting order is equal to a preorder walking on the previous tree T.

Notice the following

We have

The previous visiting order is equal to a preorder walking on the previous tree T.

Now

- Let H the cycle obtained by this preorder walking.
- It is a Hamiltonian cycle, since every vertex is visited exactly one

Notice the following

We have

The previous visiting order is equal to a preorder walking on the previous tree T.

Now

- Let H the cycle obtained by this preorder walking.
- It is a Hamiltonian cycle, since every vertex is visited exactly one

Thus

$$c(H) \leq c(W) \leq 2c(T) \leq 2c(H^*)$$

Notice the following

We have

The previous visiting order is equal to a preorder walking on the previous tree T.

Now

- Let H the cycle obtained by this preorder walking.
- It is a Hamiltonian cycle, since every vertex is visited exactly one

Thus

$$c(H) \leq c(W) \leq 2c(T) \leq 2c(H^{*})$$

Finally

We have that

$$\frac{c\left(H\right)}{c\left(H^{*}\right)} \leq 2$$

However

Given the General Traveling-Salesman Problem

If we drop the assumption that the cost function c satisfies the triangle inequality.

Theorem 35.3

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling-salesman problem.

However

Given the General Traveling-Salesman Problem

If we drop the assumption that the cost function c satisfies the triangle inequality.

Theorem 35.3

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling-salesman problem.

Excercises

From Cormen's book solve

- 35.1-1
- 35.1-2
- 35.1-4
- 35.2-1
- 35.2-2
- 35.2-3
- 35.2-5