Discrete Optimization
Trees, spanning trees

Ngày 12 tháng 10 năm 2011

Trees basics

Definition

A graph G whose connected components are trees is called a **forest**

Trees basics

Definition

A graph G whose connected components are trees is called a **forest**

Theorem (Basic tree properties.)

Trees basics

Definition

A graph G whose connected components are trees is called a **forest**

Theorem (Basic tree properties.)

- *A tree T_n with n vertices has $n - 1$ edges.*

Trees basics

Definition

A graph G whose connected components are trees is called a **forest**

Theorem (Basic tree properties.)

- *A tree T_n with n vertices has $n - 1$ edges.*
- *In a tree, between any two vertices there is a unique path.*

Trees basics

Definition

A graph G whose connected components are trees is called a **forest**

Theorem (Basic tree properties.)

- *A tree T_n with n vertices has $n - 1$ edges.*
- *In a tree, between any two vertices there is a unique path.*
- *A connected graph $G(V, E)$ with n vertices and $n - 1$ edges is a tree.*

Trees basics

Definition

A graph G whose connected components are trees is called a **forest**

Theorem (Basic tree properties.)

- *A tree T_n with n vertices has $n - 1$ edges.*
- *In a tree, between any two vertices there is a unique path.*
- *A connected graph $G(V, E)$ with n vertices and $n - 1$ edges is a tree.*
- *A tree contains at least two leaves.*

Trees basics

Definition

A graph G whose connected components are trees is called a **forest**

Theorem (Basic tree properties.)

- *A tree T_n with n vertices has $n-1$ edges.*
- *In a tree, between any two vertices there is a unique path.*
- *A connected graph $G(V, E)$ with n vertices and $n-1$ edges is a tree.*
- *A tree contains at least two leaves.*

Definition

The cost of a weighted graph is $\sum_{e \in E(G)} \omega(e)$ where $\omega(e)$ is the weight (cost) of the edge e.

Cayley's formula

Definition

Two labeled graphs G_1, G_2 are **isomorphic** *if their vertices are labeled by the same set of unique labels and $(i, j) \in E(G_1)$ if and only if $(i, j) \in E(G_2)$*

Cayley's formula

Definition

Two labeled graphs G_1, G_2 are **isomorphic** *if their vertices are labeled by the same set of unique labels and $(i, j) \in E(G_1)$ if and only if $(i, j) \in E(G_2)$*

Cayley's formula

Definition

Two labeled graphs G_1, G_2 are **isomorphic** *if their vertices are labeled by the same set of unique labels and $(i, j) \in E(G_1)$ if and only if $(i, j) \in E(G_2)$*

It is easy to see that there are 3 non-isomorphic labeled trees with 3 vertices. With slightly more efforts you can verify that there are 16 non-isomorphic trees with 4 vertices, or that a path of length n can be labeled in $(n-2)!$ non-isomorphic ways.

Cayley's formula

Definition

Two labeled graphs G_1, G_2 are **isomorphic** *if their vertices are labeled by the same set of unique labels and $(i, j) \in E(G_1)$ if and only if $(i, j) \in E(G_2)$*

It is easy to see that there are 3 non-isomorphic labeled trees with 3 vertices. With slightly more efforts you can verify that there are 16 non-isomorphic trees with 4 vertices, or that a path of length n can be labeled in $(n-2)!$ non-isomorphic ways. In 1860 the British mathematicain Arthur Cayley discovered a remarkable formula that counts the number of non-isomorphic labeled trees. There are many proofs of the formula. We shall look at an algorithmic proof due to Prüfer.

Prüfer's labeled trees coding

Let T be a labeled tree of order n and let the labels on its vertices be: $1, 2, \ldots n$.
We "code" T as with an $n-2$ list as follows:

Prüfer's labeled trees coding

Let T be a labeled tree of order n and let the labels on its vertices be: $1, 2, \ldots n$.
We "code" T as with an $n-2$ list as follows:

1. *If $|T| = 3$ then associate with T the list [k] where k is the label of the only non-leaf of T.*

Prüfer's labeled trees coding

Let T be a labeled tree of order n and let the labels on its vertices be: $1, 2, \ldots n$.
We "code" T as with an $n-2$ list as follows:

1. *If $|T| = 3$ then associate with T the list [k] where k is the label of the only non-leaf of T.*
2. *If $|T| = n+1$ let m be the largest label among all leaves of T (recall that T has at least two leaves). Let the vertex of T connected by an edge to m have the label $l(m)$.*

Prüfer's labeled trees coding

Let T be a labeled tree of order n and let the labels on its vertices be: $1, 2, \ldots n$.
We "code" T as with an $n-2$ list as follows:

1. *If $|T| = 3$ then associate with T the list [k] where k is the label of the only non-leaf of T.*
2. *If $|T| = n+1$ let m be the largest label among all leaves of T (recall that T has at least two leaves). Let the vertex of T connected by an edge to m have the label $l(m)$.*
3. *The list associated with T is constructed recursively: we start with $[l(m)]$.*

Prüfer's labeled trees coding

Let T be a labeled tree of order n and let the labels on its vertices be: $1, 2, \ldots n$.
We "code" T as with an $n-2$ list as follows:

1. *If $|T| = 3$ then associate with T the list [k] where k is the label of the only non-leaf of T.*
2. *If $|T| = n+1$ let m be the largest label among all leaves of T (recall that T has at least two leaves). Let the vertex of T connected by an edge to m have the label $l(m)$.*
3. *The list associated with T is constructed recursively: we start with $[l(m)]$.*
4. *Remove from T the leaf m and continue recursively until you reach a tree with 3 vertices.*

Prüfer's labeled trees coding

Let T be a labeled tree of order n and let the labels on its vertices be: $1, 2, \ldots n$.
We "code" T as with an $n-2$ list as follows:

1. *If $|T| = 3$ then associate with T the list [k] where k is the label of the only non-leaf of T.*
2. *If $|T| = n+1$ let m be the largest label among all leaves of T (recall that T has at least two leaves). Let the vertex of T connected by an edge to m have the label $l(m)$.*
3. *The list associated with T is constructed recursively: we start with $[l(m)]$.*
4. *Remove from T the leaf m and continue recursively until you reach a tree with 3 vertices.*
5. *Add at the end of the tree the non-leaf of this tree.*

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.
2. Given a list L of $n-2$ integers there is unique labeled tree of order n whose list is L.

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.
2. Given a list L of $n-2$ integers there is unique labeled tree of order n whose list is L.
3. There is a bijection between the n^{n-2} different lists and and labeled trees of order n.

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.
2. Given a list L of $n-2$ integers there is unique labeled tree of order n whose list is L.
3. There is a bijection between the n^{n-2} different lists and and labeled trees of order n.

Observation

To reconstruct T from a given list we proceed as follows:

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.
2. Given a list L of $n-2$ integers there is unique labeled tree of order n whose list is L.
3. There is a bijection between the n^{n-2} different lists and and labeled trees of order n.

Observation

To reconstruct T from a given list we proceed as follows:

- *Let* $\mathbf{L} = [l_1, l_2, \ldots, l_{n-2}]$ *be a list.*

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.
2. Given a list L of $n-2$ integers there is unique labeled tree of order n whose list is L.
3. There is a bijection between the n^{n-2} different lists and and labeled trees of order n.

Observation

To reconstruct T from a given list we proceed as follows:

- *Let* $\mathbf{L} = [l_1, l_2, \ldots, l_{n-2}]$ *be a list.*
- *Let* $\{t_1, t_2, \ldots, t_k\}$ *be the labels not in* $\mathbf{L}$ *arranged in decreasing order (these are the leaves of* T*).*

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.
2. Given a list L of $n-2$ integers there is unique labeled tree of order n whose list is L.
3. There is a bijection between the n^{n-2} different lists and and labeled trees of order n.

Observation

To reconstruct T from a given list we proceed as follows:

- *Let* $\mathbf{L} = [l_1, l_2, \ldots, l_{n-2}]$ *be a list.*
- *Let* $\{t_1, t_2, \ldots, t_k\}$ *be the labels not in* $\mathbf{L}$ *arranged in decreasing order (these are the leaves of* T*).*
- *Start with the empty tree* T_0*.*

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.
2. Given a list L of $n-2$ integers there is unique labeled tree of order n whose list is L.
3. There is a bijection between the n^{n-2} different lists and and labeled trees of order n.

Observation

To reconstruct T from a given list we proceed as follows:

- *Let* $\mathbf{L} = [l_1, l_2, \ldots, l_{n-2}]$ *be a list.*
- *Let* $\{t_1, t_2, \ldots, t_k\}$ *be the labels not in* $\mathbf{L}$ *arranged in decreasing order (these are the leaves of* T*).*
- *Start with the empty tree* T_0*.*
- *Add the edge* (t_1, l_1) *to* T_0 *remove both of from the list and the set.*

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.
2. Given a list L of $n-2$ integers there is unique labeled tree of order n whose list is L.
3. There is a bijection between the n^{n-2} different lists and and labeled trees of order n.

Observation

To reconstruct T from a given list we proceed as follows:

- *Let* $\mathbf{L} = [l_1, l_2, \ldots, l_{n-2}]$ *be a list.*
- *Let* $\{t_1, t_2, \ldots, t_k\}$ *be the labels not in* $\mathbf{L}$ *arranged in decreasing order (these are the leaves of* T*).*
- *Start with the empty tree* T_0*.*
- *Add the edge* (t_1, l_1) *to* T_0 *remove both of from the list and the set.*
- *If there are no more apperances of* l_1 *in the list, add it to the set.*

Prüfer's code has the following properties:

1. With every tree T of order n it asociates a list of lenght $n-2$.
2. Given a list L of $n-2$ integers there is unique labeled tree of order n whose list is L.
3. There is a bijection between the n^{n-2} different lists and and labeled trees of order n.

Observation

To reconstruct T from a given list we proceed as follows:

- *Let $\mathbf{L} = [l_1, l_2, \ldots, l_{n-2}]$ be a list.*
- *Let $\{t_1, t_2, \ldots, t_k\}$ be the labels not in $\mathbf{L}$ arranged in decreasing order (these are the leaves of T).*
- *Start with the empty tree T_0.*
- *Add the edge (t_1, l_1) to T_0 remove both of from the list and the set.*
- *If there are no more apperances of l_1 in the list, add it to the set.*
- *After $n-2$ steps the list will be empty, the set will contain two labels. Add the edge between the labels.*

Examples

1. The list $[k, k, k, \ldots k]$ is the labeled tree with $n-1$ leaves connected to a single vertex of degree $n-1$ labeled k.

Examples

1. The list $[k, k, k, \ldots k]$ is the labeled tree with $n-1$ leaves connected to a single vertex of degree $n-1$ labeled k.
2. If the label k appears m times in the list it corresponds to a labeld tree with a vertex labeled k of degree $m+1$.

Examples

1. The list $[k, k, k, \ldots k]$ is the labeled tree with $n-1$ leaves connected to a single vertex of degree $n-1$ labeled k.
2. If the label k appears m times in the list it corresponds to a labeld tree with a vertex labeled k of degree $m+1$.
3. If the list has $n-2$ distinct labels then it is the code of a path.

Examples

1. The list $[k, k, k, \ldots k]$ is the labeled tree with $n-1$ leaves connected to a single vertex of degree $n-1$ labeled k.
2. If the label k appears m times in the list it corresponds to a labeld tree with a vertex labeled k of degree $m+1$.
3. If the list has $n-2$ distinct labels then it is the code of a path.
4. A challenge: how many labeled trees of order 100 have exactly 5 leaves?

Examples

1. The list $[k, k, k, \ldots k]$ is the labeled tree with $n-1$ leaves connected to a single vertex of degree $n-1$ labeled k.
2. If the label k appears m times in the list it corresponds to a labeld tree with a vertex labeled k of degree $m+1$.
3. If the list has $n-2$ distinct labels then it is the code of a path.
4. A challenge: how many labeled trees of order 100 have exactly 5 leaves?
5. How many different labeled trees of order 100 in which all non-leaves have degree 3 are there?

Efficiency of Algorithms

- We shall try to analyze roughly the execution efficiency of algorithms we study. That is count the number of “steps” an algorithm executes on a given input.

Efficiency of Algorithms

- We shall try to analyze roughly the execution efficiency of algorithms we study. That is count the number of “steps” an algorithm executes on a given input.
- An algorithm is considered **efficient** if the number of steps it executes as a function of the size of the input is polynomial.

Efficiency of Algorithms

- We shall try to analyze roughly the execution efficiency of algorithms we study. That is count the number of “steps” an algorithm executes on a given input.
- An algorithm is considered **efficient** if the number of steps it executes as a function of the size of the input is polynomial.
- For example, searching for an item in a list of n items takes n steps. It is effcient.

Efficiency of Algorithms

- We shall try to analyze roughly the execution efficiency of algorithms we study. That is count the number of “steps” an algorithm executes on a given input.
- An algorithm is considered **efficient** if the number of steps it executes as a function of the size of the input is polynomial.
- For example, searching for an item in a list of n items takes n steps. It is effcient.
- Sorting a list of n items is also efficient.

Efficiency of Algorithms

- We shall try to analyze roughly the execution efficiency of algorithms we study. That is count the number of “steps” an algorithm executes on a given input.
- An algorithm is considered **efficient** if the number of steps it executes as a function of the size of the input is polynomial.
- For example, searching for an item in a list of n items takes n steps. It is effcient.
- Sorting a list of n items is also efficient.
- Listing all permutations of n objecst is **NOT** efficient. It executes $n!$ steps.

MCST minimum cost spanning tree

The MCST is very important in many applications in discrete optimization.

- We shall present two algorithms to find a minimum cost spanning tree in a weighted graph. The algorithms are based on two common techniques for building a tree:

MCST minimum cost spanning tree

The MCST is very important in many applications in discrete optimization.

- We shall present two algorithms to find a minimum cost spanning tree in a weighted graph. The algorithms are based on two common techniques for building a tree:
- To a given tree T add a new vertex and connect it by a single edge to some vertex in T.

MCST minimum cost spanning tree

The MCST is very important in many applications in discrete optimization.

- We shall present two algorithms to find a minimum cost spanning tree in a weighted graph. The algorithms are based on two common techniques for building a tree:
- To a given tree T add a new vertex and connect it by a single edge to some vertex in T.
- Prim's algorithm (1957) is based on this technique.

MCST minimum cost spanning tree

The MCST is very important in many applications in discrete optimization.

- We shall present two algorithms to find a minimum cost spanning tree in a weighted graph. The algorithms are based on two common techniques for building a tree:
- To a given tree T add a new vertex and connect it by a single edge to some vertex in T.
- Prim's algorithm (1957) is based on this technique.
- Add an edge connecting two vertices belonging to two disjoint trees.

MCST minimum cost spanning tree

The MCST is very important in many applications in discrete optimization.

- We shall present two algorithms to find a minimum cost spanning tree in a weighted graph. The algorithms are based on two common techniques for building a tree:
- To a given tree T add a new vertex and connect it by a single edge to some vertex in T.
- Prim's algorithm (1957) is based on this technique.
- Add an edge connecting two vertices belonging to two disjoint trees.
- Kurskal's algorithm (1956) is based on this technique.

MCST minimum cost spanning tree

The MCST is very important in many applications in discrete optimization.

- We shall present two algorithms to find a minimum cost spanning tree in a weighted graph. The algorithms are based on two common techniques for building a tree:
- To a given tree T add a new vertex and connect it by a single edge to some vertex in T.
- Prim's algorithm (1957) is based on this technique.
- Add an edge connecting two vertices belonging to two disjoint trees.
- Kurskal's algorithm (1956) is based on this technique.
- Both algorithms were discovered earlier by Czech matehmaticians.

MCST minimum cost spanning tree

The MCST is very important in many applications in discrete optimization.

- We shall present two algorithms to find a minimum cost spanning tree in a weighted graph. The algorithms are based on two common techniques for building a tree:
- To a given tree T add a new vertex and connect it by a single edge to some vertex in T.
- Prim's algorithm (1957) is based on this technique.
- Add an edge connecting two vertices belonging to two disjoint trees.
- Kurskal's algorithm (1956) is based on this technique.
- Both algorithms were discovered earlier by Czech matehmaticians.
- You can see a nice demonstration of Prim's and other algorithms at: **http://www.unf.edu/∼wkloster/foundations/PrimApplet/PrimApplet.htm**

MCST

(Kruskal's Algorithm)

MCST

(Kruskal's Algorithm)

- *Organize all edges in a priority queue.*

MCST

(Kruskal's Algorithm)

- *Organize all edges in a priority queue.*
- *Start with all vertices as trees with a single vertex (a forest)*

MCST

(Kruskal's Algorithm)

- *Organize all edges in a priority queue.*
- *Start with all vertices as trees with a single vertex (a forest)*
- *Repeat: select the cheapest edge from the queue.*

MCST

(Kruskal's Algorithm)

- *Organize all edges in a priority queue.*
- *Start with all vertices as trees with a single vertex (a forest)*
- *Repeat: select the cheapest edge from the queue.*
- *If it connects two vertices belonging to two different trees in your forest, add it (merge the trees into a single tree). Discard the edge.*

MCST

(Kruskal's Algorithm)

- *Organize all edges in a priority queue.*
- *Start with all vertices as trees with a single vertex (a forest)*
- *Repeat: select the cheapest edge from the queue.*
- *If it connects two vertices belonging to two different trees in your forest, add it (merge the trees into a single tree). Discard the edge.*
- *Stop when you add $n - 1$ edges.*

MCST

(Kruskal's Algorithm)

- *Organize all edges in a priority queue.*
- *Start with all vertices as trees with a single vertex (a forest)*
- *Repeat: select the cheapest edge from the queue.*
- *If it connects two vertices belonging to two different trees in your forest, add it (merge the trees into a single tree). Discard the edge.*
- *Stop when you add $n - 1$ edges.*

(Analysis:)

MCST

(Kruskal's Algorithm)

- *Organize all edges in a priority queue.*
- *Start with all vertices as trees with a single vertex (a forest)*
- *Repeat: select the cheapest edge from the queue.*
- *If it connects two vertices belonging to two different trees in your forest, add it (merge the trees into a single tree). Discard the edge.*
- *Stop when you add $n-1$ edges.*

(Analysis:)

- *Organizing all edges in a priority queue takes $c|E|$ steps.*

MCST

(Kruskal's Algorithm)

- *Organize all edges in a priority queue.*
- *Start with all vertices as trees with a single vertex (a forest)*
- *Repeat: select the cheapest edge from the queue.*
- *If it connects two vertices belonging to two different trees in your forest, add it (merge the trees into a single tree). Discard the edge.*
- *Stop when you add $n-1$ edges.*

(Analysis:)

- *Organizing all edges in a priority queue takes $c|E|$ steps.*
- *We next execute $n-1$ steps (selecting and adding edges). Each step takes at most $\log(|E|)$ steps.*

MCST

(Kruskal's Algorithm)

- *Organize all edges in a priority queue.*
- *Start with all vertices as trees with a single vertex (a forest)*
- *Repeat: select the cheapest edge from the queue.*
- *If it connects two vertices belonging to two different trees in your forest, add it (merge the trees into a single tree). Discard the edge.*
- *Stop when you add* $n - 1$ *edges.*

(Analysis:)

- *Organizing all edges in a priority queue takes* $c|E|$ *steps.*
- *We next execute* $n - 1$ *steps (selecting and adding edges). Each step takes at most* $\log(|E|)$ *steps.*
- *Since* $|E| < n^2$ *the total number of steps is polynomial in n, the size of the input graph.*

MCST

(Prim's Algortihm)

MCST

(Prim's Algortihm)

- *Select a vertex for the weighted graph G. This is your initial tree.*

MCST

(Prim's Algortihm)

- *Select a vertex for the weighted graph G. This is your initial tree.*
- *Scan all vertices "visible" from the current tree. Select the vertex that is connected to the current tree by the cheapest edge and add the vertex and the edge to the tree.*

MCST

(Prim's Algortihm)

- *Select a vertex for the weighted graph G. This is your initial tree.*
- *Scan all vertices "visible" from the current tree. Select the vertex that is connected to the current tree by the cheapest edge and add the vertex and the edge to the tree.*
- *Stop when the tree has n vertices, it is a spanning tree.*

MCST

(Prim's Algortihm)

- *Select a vertex for the weighted graph G. This is your initial tree.*
- *Scan all vertices "visible" from the current tree. Select the vertex that is connected to the current tree by the cheapest edge and add the vertex and the edge to the tree.*
- *Stop when the tree has n vertices, it is a spanning tree.*

Comment

We still need to prove the correctness of both algorithms. That is to show that the spanning trees produced by the algorithms do not cost more than any other tree,
Note that the MCST is not unique, there maybe many spanning trees with the minimal cost.
A proof of the correctness of Kruskal's algorithm can be found in the supplements folder.

Dijkstra Shortest Path Algorithm

Question

A package delivering company is serving all of Vietnam. They plan to use commercial airlines, trucks or trains to ship their customers packages. A package shipped from one destination to another may pass through intermediate destinations. The cost of shipping directly between some destinations is known.

For a given package from city A to the destination in city B the company would like to know the cost of the cheapest path from A to B.

Dijkstra Shortest Path Algorithm

Question

A package delivering company is serving all of Vietnam. They plan to use commercial airlines, trucks or trains to ship their customers packages. A package shipped from one destination to another may pass through intermediate destinations. The cost of shipping directly between some destinations is known.

For a given package from city A to the destination in city B the company would like to know the cost of the cheapest path from A to B.

Answer

This problem can be modeled by a weighted digraph whose vertces are all possible destinations and weighted edges are the non-negative costs of direct shipping from A to B.

Dijkstra Shortest Path Algorithm

Question

A package delivering company is serving all of Vietnam. They plan to use commercial airlines, trucks or trains to ship their customers packages. A package shipped from one destination to another may pass through intermediate destinations. The cost of shipping directly between some destinations is known.

For a given package from city A to the destination in city B the company would like to know the cost of the cheapest path from A to B.

Answer

This problem can be modeled by a weighted digraph whose vertces are all possible destinations and weighted edges are the non-negative costs of direct shipping from A to B.

Dijkstra Shortest Path Algorithm

Question

A package delivering company is serving all of Vietnam. They plan to use commercial airlines, trucks or trains to ship their customers packages. A package shipped from one destination to another may pass through intermediate destinations. The cost of shipping directly between some destinations is known.

For a given package from city A to the destination in city B the company would like to know the cost of the cheapest path from A to B.

Answer

This problem can be modeled by a weighted digraph whose vertces are all possible destinations and weighted edges are the non-negative costs of direct shipping from A to B.

Dijkstra's shortest path algorithm efficiently computes the shortest (cheapest) path.

Dijkstra's Shortest Path Algorithm

For a given vertex $v \in D(V, E)$ Dijkstra's algorithm produces a spanning tree T, rooted in v, such that the unique path in T to any vertex $w \in D(V)$ is a shortest path in D from $v \to w$.
The algorithm proceeds in steps. In each step it adds a vertex to a tree T rooted at v which is a subgraph of D such that the path from v to any vertex in T is the shortest path in D that uses only vertices in T.
At each stage of the algorithm the vertices of D will be divided into three sets:

- 1. Vertices in T are vertices whose shortest distance from v has been determined.

Dijkstra's Shortest Path Algorithm

For a given vertex $v \in D(V, E)$ Dijkstra's algorithm produces a spanning tree T, rooted in v, such that the unique path in T to any vertex $w \in D(V)$ is a shortest path in D from $v \to w$.
The algorithm proceeds in steps. In each step it adds a vertex to a tree T rooted at v which is a subgraph of D such that the path from v to any vertex in T is the shortest path in D that uses only vertices in T.
At each stage of the algorithm the vertices of D will be divided into three sets:

- 1. Vertices in T are vertices whose shortest distance from v has been determined.
- 2. Vertices whose shortest distance to v subject to the condition that the shortest known path uses only vertices from T. We say that these vertices have been **discovered**.

Dijkstra's Shortest Path Algorithm

For a given vertex $v \in D(V, E)$ Dijkstra's algorithm produces a spanning tree T, rooted in v, such that the unique path in T to any vertex $w \in D(V)$ is a shortest path in D from $v \to w$.
The algorithm proceeds in steps. In each step it adds a vertex to a tree T rooted at v which is a subgraph of D such that the path from v to any vertex in T is the shortest path in D that uses only vertices in T.
At each stage of the algorithm the vertices of D will be divided into three sets:

- 1. Vertices in T are vertices whose shortest distance from v has been determined.
- 2. Vertices whose shortest distance to v subject to the condition that the shortest known path uses only vertices from T. We say that these vertices have been **discovered**.
- 3. The remaining vertices.

(Dijkstra's algorithm)

(Dijkstra's algorithm)

- *Initialize.* $T = \{v\}$. *Mark all other vertices by* $(From, Dist, \infty)$.

(Dijkstra's algorithm)

- *Initialize.* $T = \{v\}$. *Mark all other vertices by* $(From, Dist, \infty)$.
- *Scan all vertices of D connected by an edge to a vertex in* $v \in T$. *For each such vertex w, change the entry* ∞ *to the weight of the edge* (v, w) *and From to v.*

(Dijkstra's algorithm)

- *Initialize.* $T = \{v\}$. *Mark all other vertices by* $(From, Dist, \infty)$.
- *Scan all vertices of D connected by an edge to a vertex in* $v \in T$. *For each such vertex w, change the entry* ∞ *to the weight of the edge* (v, w) *and From to v.*
- **Update**.

(Dijkstra's algorithm)

- *Initialize.* $T = \{v\}$. *Mark all other vertices by* $(From, Dist, \infty)$.
- *Scan all vertices of D connected by an edge to a vertex in* $v \in T$. *For each such vertex w, change the entry* ∞ *to the weight of the edge* (v, w) *and From to v.*
- **Update**.
  - *Among all vertices that have been discovered select u, the vertex with smallest* $Dist(u)$.

(Dijkstra's algorithm)

- *Initialize.* $T = \{v\}$. *Mark all other vertices by* $(From, Dist, \infty)$.
- *Scan all vertices of D connected by an edge to a vertex in* $v \in T$. *For each such vertex w, change the entry* ∞ *to the weight of the edge* (v, w) *and From to v.*
- **Update**.
  - *Among all vertices that have been discovered select u, the vertex with smallest* $Dist(u)$.
  - *Add u to T.*

(Dijkstra's algorithm)

- *Initialize.* $T = \{v\}$. *Mark all other vertices by* $(From, Dist, \infty)$.
- *Scan all vertices of D connected by an edge to a vertex in* $v \in T$. *For each such vertex w, change the entry* ∞ *to the weight of the edge* (v, w) *and From to v.*
- **Update**.
  - *Among all vertices that have been discovered select u, the vertex with smallest* $Dist(u)$.
  - *Add u to T.*
  - *For every vertex x connected by an edge to u , if* $Dist(x) > \omega(u, x) + Dist(u)$ *then* $From(x) = u,\ Dist(x) = \omega(u, x) + Dist(u)$.

(Dijkstra's algorithm)

- *Initialize.* $T = \{v\}$. *Mark all other vertices by* $(From, Dist, \infty)$.
- *Scan all vertices of D connected by an edge to a vertex in* $v \in T$. *For each such vertex w, change the entry* ∞ *to the weight of the edge* (v, w) *and From to v.*
- **Update**.
  - *Among all vertices that have been discovered select u, the vertex with smallest* $Dist(u)$.
  - *Add u to T.*
  - *For every vertex x connected by an edge to u , if* $Dist(x) > \omega(u, x) + Dist(u)$ *then* $From(x) = u,\ Dist(x) = \omega(u, x) + Dist(u)$.
  - *if* $|V(T)| < |V(D)|$ *go back to* **Update** *else finish, you are done.*

Sample execution of Dijkstra's shortest path algorithm

Floyd's all pairs shortest distance algorithm

(Introduction)

Floyd's all pairs shortest path algorithm is a beautiful example of using dynamic programming to produce a very simple and elegant code to produce the all pairs shortest paths. It will give us both the shortest distance and the actual shortest path between any pair of vertices in weighted graph.

. Three matrices of size $|V(G)| \times |V(G)|$ *will be involved:*

Floyd's all pairs shortest distance algorithm

(Introduction)

Floyd's all pairs shortest path algorithm is a beautiful example of using dynamic programming to produce a very simple and elegant code to produce the all pairs shortest paths. It will give us both the shortest distance and the actual shortest path between any pair of vertices in weighted graph.

. *Three matrices of size* $|V(G)| \times |V(G)|$ *will be involved:*

- *Graph will hold the direct distance (cost) between any pair of vertices.*

Floyd's all pairs shortest distance algorithm

(Introduction)

Floyd's all pairs shortest path algorithm is a beautiful example of using dynamic programming to produce a very simple and elegant code to produce the all pairs shortest paths. It will give us both the shortest distance and the actual shortest path between any pair of vertices in weighted graph.

. *Three matrices of size* $|V(G)| \times |V(G)|$ *will be involved:*

i *Graph will hold the direct distance (cost) between any pair of vertices.*

ii *ShortestDistance after step i will hold the current known distance subject to the condition that only vertices with index* $\leq$ *i can be used in the path.*

Floyd's all pairs shortest distance algorithm

(Introduction)

Floyd's all pairs shortest path algorithm is a beautiful example of using dynamic programming to produce a very simple and elegant code to produce the all pairs shortest paths. It will give us both the shortest distance and the actual shortest path between any pair of vertices in weighted graph.

. Three matrices of size $|V(G)| \times |V(G)|$ *will be involved:*

i *Graph will hold the direct distance (cost) between any pair of vertices.*

ii *ShortestDistance after step i will hold the current known distance subject to the condition that only vertices with index* $\leq$ *i can be used in the path.*

iii *The ShortestPath matrix will enable us to actually list the cities on the shortest path.*

(The code)

```
for ( n = 0; n < vertices; n++)
for ( m = 0; m < vertices; m++){
                ShortestDistance[m][n] = Graph[n][m];
                ShortestPath[m][n] = -1;                }

for (int x = 0; x < vertices; x++)
for (int y = 0; y < vertices; y++)
for (int z = 0; z < vertices; z++)
if (ShortestDistance[y][z] > ShortestDistance[y][x] +
ShortestDistance[x][z]){
ShortestDistance[y][z] = ShortestDistance[y][x] +
ShortestDistance[x][z]; ShortestPath[y][z] = x;
        }
```

Understanding Floyd's algorithm

- The blue code records the cost of going directly from city m to city n.

Understanding Floyd's algorithm

- The blue code records the cost of going directly from city m to city n.
- The entry -1 in the ShortestPath matrix in location (m, n) tells us that the current shortest path between m and n is to go directly from $n \rightarrow m$.

Understanding Floyd's algorithm

- The blue code records the cost of going directly from city m to city n.
- The entry -1 in the ShortestPath matrix in location (m, n) tells us that the current shortest path between m and n is to go directly from $n \to m$.
- In the purple code: when $x = 0$ for every pair the code checks whether it is batter to go from $x \to y$ via 0. If so, it is recorded in both matrices.

Understanding Floyd's algorithm

- The blue code records the cost of going directly from city m to city n.
- The entry -1 in the ShortestPath matrix in location (m, n) tells us that the current shortest path between m and n is to go directly from $n \to m$.
- In the purple code: when $x = 0$ for every pair the code checks whether it is batter to go from $x \to y$ via 0. If so, it is recorded in both matrices.
- When $x = k$ the matrices record the cost of the shortest path from $x \to y$ subject to the restriction that only cities with index $\leq k$ can be used.

Understanding Floyd's algorithm

- The blue code records the cost of going directly from city m to city n.
- The entry -1 in the ShortestPath matrix in location (m, n) tells us that the current shortest path between m and n is to go directly from $n \to m$.
- In the purple code: when $x = 0$ for every pair the code checks whether it is batter to go from $x \to y$ via 0. If so, it is recorded in both matrices.
- When $x = k$ the matrices record the cost of the shortest path from $x \to y$ subject to the restriction that only cities with index $\leq k$ can be used.

Understanding Floyd's algorithm

- The blue code records the cost of going directly from city m to city n.
- The entry -1 in the ShortestPath matrix in location (m, n) tells us that the current shortest path between m and n is to go directly from $n \to m$.
- In the purple code: when $x = 0$ for every pair the code checks whether it is batter to go from $x \to y$ via 0. If so, it is recorded in both matrices.
- When $x = k$ the matrices record the cost of the shortest path from $x \to y$ subject to the restriction that only cities with index $\leq k$ can be used.

This is a beautiful example of dynamic programming. The dynamic aspect of this code is that at every loop of the code we increase the number of cities we may use until we incorporate all cities.

(Analysis)

The purple code excutes n^3 steps and the blue code executes n^2 steps, definitely polynomial.

With a little higher execution time than Dijkstra's algortihm we get more information and a much simpler code.

PERT-CPM

Many complex porjects involve many steps. Some steps cannot be started before other steps have been completed. For instance, when building a road you cannot apply the final coat before many other steps have been completed. When you build a house, you cannot install electricty before the foundations and walls have been completed.

PERT-CPM

Many complex porjects involve many steps. Some steps cannot be started before other steps have been completed. For instance, when building a road you cannot apply the final coat before many other steps have been completed. When you build a house, you cannot install electricty before the foundations and walls have been completed.

PERT-CPM stands for Project Evaluation Review Technique - Critical Path Method.

To evaluate the minimum time required to complete a project we build a directed graph as follows:

1. We start with two vertices labeld *S and F* (start and finish).

To evaluate the minimum time required to complete a project we build a directed graph as follows:

1. We start with two vertices labeld *S and F* (start and finish).
2. The vertices of the graph will be the basic steps required during the project.

To evaluate the minimum time required to complete a project we build a directed graph as follows:

1. We start with two vertices labeld *S and F* (start and finish).
2. The vertices of the graph will be the basic steps required during the project.
3. There is a directed edge from $s_i \rightarrow s_j$ if step s_j cannot be started before step s_i has been completed.

To evaluate the minimum time required to complete a project we build a directed graph as follows:

1. We start with two vertices labeld *S and F* (start and finish).
2. The vertices of the graph will be the basic steps required during the project.
3. There is a directed edge from $s_i \rightarrow s_j$ if step s_j cannot be started before step s_i has been completed.
4. $\omega(s_i, s_j)$, the weight of the edge is the time required to complete the step s_i.

To evaluate the minimum time required to complete a project we build a directed graph as follows:

1. We start with two vertices labeld *S and F* (start and finish).
2. The vertices of the graph will be the basic steps required during the project.
3. There is a directed edge from $s_i \rightarrow s_j$ if step s_j cannot be started before step s_i has been completed.
4. $\omega(s_i, s_j)$, the weight of the edge is the time required to complete the step s_i.

To evaluate the minimum time required to complete a project we build a directed graph as follows:

1. We start with two vertices labeld *S and F* (start and finish).
2. The vertices of the graph will be the basic steps required during the project.
3. There is a directed edge from $s_i \rightarrow s_j$ if step s_j cannot be started before step s_i has been completed.
4. $\omega(s_i, s_j)$, the weight of the edge is the time required to complete the step s_i.

The minimum time required to finish the project is the length of the longest path from *S to F* (critical path).

Building a xe máy

1. Frame: Build and prepare the frame.

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:
8. Install the gas tank:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:
8. Install the gas tank:
9. Install the front wheel:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:
8. Install the gas tank:
9. Install the front wheel:
10. Install the handle bar:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:
8. Install the gas tank:
9. Install the front wheel:
10. Install the handle bar:
11. Install Ignition system:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:
8. Install the gas tank:
9. Install the front wheel:
10. Install the handle bar:
11. Install Ignition system:
12. Install Battery:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:
8. Install the gas tank:
9. Install the front wheel:
10. Install the handle bar:
11. Install Ignition system:
12. Install Battery:
13. Install head lights:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:
8. Install the gas tank:
9. Install the front wheel:
10. Install the handle bar:
11. Install Ignition system:
12. Install Battery:
13. Install head lights:
14. Install back lights:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:
8. Install the gas tank:
9. Install the front wheel:
10. Install the handle bar:
11. Install Ignition system:
12. Install Battery:
13. Install head lights:
14. Install back lights:
15. Paint:

Building a xe máy

1. Frame: Build and prepare the frame.
2. Back Wheel: Build the back wheel.
3. Front wheel: Build the front wheel.
4. Mount Back wheel:
5. Mount Engine:
6. Mount the gear:
7. Connect to back wheel:
8. Install the gas tank:
9. Install the front wheel:
10. Install the handle bar:
11. Install Ignition system:
12. Install Battery:
13. Install head lights:
14. Install back lights:
15. Paint:
16. Final assemly:

The following table shows which jobs must be preceded by other jobs:

Job	Preceded by	Jobs
Paint		Final assembly
Final Assembly		$1, 2, 3, \ldots, 12$
Back Wheel		1,2
Mount engine		1
Conect to back wheel		$5, 6$
Mount Gear		5

etc.

The following table shows which jobs must be preceded by other jobs:

Job	Preceded by	Jobs
Paint		Final assembly
Final Assembly		$1, 2, 3, \ldots, 12$
Back Wheel		1,2
Mount engine		1
Conect to back wheel		$5, 6$
Mount Gear		5

etc.

Once the table is complete and we know how much time is required in each step we can build the weighted digraph and find the crtical path.

A simpler detailed example, xe dap.pdf can be found in the supplements.