Lecture 15 - Trees Contd

April 26, 2019

Review

Trees – Review

- Tree – A connected graph that contains no simple circuits.

Trees – Review

- Tree – A connected graph that contains no simple circuits.
- An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

Trees – Review

- Tree – A connected graph that contains no simple circuits.
- An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.
- Rooted trees - a particular vertex is root.

Trees – Review

- Tree – A connected graph that contains no simple circuits.
- An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.
- Rooted trees - a particular vertex is root.
- It could be any node which is why designating root node as internal node is not that difficult to see.

Trees – Review

- Tree – A connected graph that contains no simple circuits.
- An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.
- Rooted trees - a particular vertex is root.
- It could be any node which is why designating root node as internal node is not that difficult to see.
- We are only worried about nodes which are leaves and which are not!

Trees – Review

- Tree – A connected graph that contains no simple circuits.
- An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.
- Rooted trees - a particular vertex is root.
- It could be any node which is why designating root node as internal node is not that difficult to see.
- We are only worried about nodes which are leaves and which are not!
- m-ary trees – if every internal vertex has no more than m children,

Trees – Review

- Tree – A connected graph that contains no simple circuits.
- An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.
- Rooted trees - a particular vertex is root.
- It could be any node which is why designating root node as internal node is not that difficult to see.
- We are only worried about nodes which are leaves and which are not!
- m-ary trees – if every internal vertex has no more than m children,full m-ary tree – if every internal vertex has exactly m children.

Trees – Review

- Tree – A connected graph that contains no simple circuits.
- An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.
- Rooted trees - a particular vertex is root.
- It could be any node which is why designating root node as internal node is not that difficult to see.
- We are only worried about nodes which are leaves and which are not!
- m-ary trees – if every internal vertex has no more than m children,full m-ary tree – if every internal vertex has exactly m children.
- A tree with n vertices has $n - 1$ edges.

Trees – Review

- Tree – A connected graph that contains no simple circuits.
- An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.
- Rooted trees - a particular vertex is root.
- It could be any node which is why designating root node as internal node is not that difficult to see.
- We are only worried about nodes which are leaves and which are not!
- m-ary trees – if every internal vertex has no more than m children,full m-ary tree – if every internal vertex has exactly m children.
- A tree with n vertices has $n-1$ edges.
- Height of a tree, h, balanced m-ary – leaves at h or $h-1$

Trees – Review

- There are at most m^h leaves in an m-ary tree of height h.

Trees – Review

- There are at most m^h leaves in an m-ary tree of height h. We showed proof by induction.

- There are at most m^h leaves in an m-ary tree of height h. We showed proof by induction.
- If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$.

Trees – Review

- There are at most m^h leaves in an m-ary tree of height h. We showed proof by induction.
- If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$. This was more or less a corollary of previous theorem.

Trees – Review

- There are at most m^h leaves in an m-ary tree of height h. We showed proof by induction.
- If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$. This was more or less a corollary of previous theorem.
- Applications of trees - BSTs and Decision Trees.

- There are at most m^h leaves in an m-ary tree of height h. We showed proof by induction.
- If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$. This was more or less a corollary of previous theorem.
- Applications of trees - BSTs and Decision Trees.
- BST - the complexity of adding and locating an item in a BST.

Trees – Review

- There are at most m^h leaves in an m-ary tree of height h. We showed proof by induction.
- If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$. This was more or less a corollary of previous theorem.
- Applications of trees - BSTs and Decision Trees.
- BST - the complexity of adding and locating an item in a BST.
- We considered a BST T for a list of n items.

Computational Complexity of Adding and Locating an Item in BST - Quick Review

- We formed a full binary tree U from T.

Computational Complexity of Adding and Locating an Item in BST - Quick Review

- We formed a full binary tree U from T.
- The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf.

Computational Complexity of Adding and Locating an Item in BST - Quick Review

- We formed a full binary tree U from T.
- The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf.
- The internal vertices of U are vertices of T, therefore U has n vertices.

Computational Complexity of Adding and Locating an Item in BST - Quick Review

- We formed a full binary tree U from T.
- The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf.
- The internal vertices of U are vertices of T, therefore U has n vertices.
- If there i internal vertices for a full m-ary tree we have the number of leaves $l = (m-1)i + 1$.

Computational Complexity of Adding and Locating an Item in BST - Quick Review

- We formed a full binary tree U from T.
- The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf.
- The internal vertices of U are vertices of T, therefore U has n vertices.
- If there i internal vertices for a full m-ary tree we have the number of leaves $l = (m-1)i + 1$.
- Here therefore there are $n+1$ leaves for U.

Computational Complexity of Adding and Locating an Item in BST

- The height of U is greater than or equal to $h = \lceil log(n + 1) \rceil$.

Computational Complexity of Adding and Locating an Item in BST

- The height of U is greater than or equal to $h = \lceil log(n+1) \rceil$.
- Therefore, we need to perform at least $\lceil log(n+1) \rceil$ comparisons to add an item.

Computational Complexity of Adding and Locating an Item in BST

- The height of U is greater than or equal to $h = \lceil log(n+1) \rceil$.
- Therefore, we need to perform at least $\lceil log(n+1) \rceil$ comparisons to add an item.
- For a balanced m-ary tree then its height is equal to $\lceil log(n+1) \rceil$ and so no more comparisons are required.

Computational Complexity of Adding and Locating an Item in BST

- The height of U is greater than or equal to $h = \lceil log(n+1) \rceil$.
- Therefore, we need to perform at least $\lceil log(n+1) \rceil$ comparisons to add an item.
- For a balanced m-ary tree then its height is equal to $\lceil log(n+1) \rceil$ and so no more comparisons are required.
- This is why there are many algorithms that try to rebalance BSTs after items are added.

Applications of Trees - Decision Trees

- Using decisions trees we were trying to lower bound the worst-case complexity of comparison based sorting algorithms.

Applications of Trees - Decision Trees

- Using decisions trees we were trying to lower bound the worst-case complexity of comparison based sorting algorithms.
- Given a list of n elements we considered a binary decision tree in which each internal vertex represents a comparison of two elements.

Applications of Trees - Decision Trees

- Using decisions trees we were trying to lower bound the worst-case complexity of comparison based sorting algorithms.
- Given a list of n elements we considered a binary decision tree in which each internal vertex represents a comparison of two elements.
- Each leaf represents one of the $n!$ permutations of n elements.

Applications of Trees - Decision Trees

- Using decisions trees we were trying to lower bound the worst-case complexity of comparison based sorting algorithms.
- Given a list of n elements we considered a binary decision tree in which each internal vertex represents a comparison of two elements.
- Each leaf represents one of the $n!$ permutations of n elements.
- Complexity is based on number of binary comparisons, worst case complexity is based on largest number of binary comparisons needed to sort a list with n elements.

Applications of Trees - Decision Trees

- Using decisions trees we were trying to lower bound the worst-case complexity of comparison based sorting algorithms.
- Given a list of n elements we considered a binary decision tree in which each internal vertex represents a comparison of two elements.
- Each leaf represents one of the $n!$ permutations of n elements.
- Complexity is based on number of binary comparisons, worst case complexity is based on largest number of binary comparisons needed to sort a list with n elements.
- That is the height of the decision tree with $n!$ leaves - at least $\lceil log\ n! \rceil$

Complexity of Comparison based sorting algorithms

- We thus concluded – a sorting algorithm based on binary comparisons requires at least $\lceil log n! \rceil$ comparisons.

Complexity of Comparison based sorting algorithms

- We thus concluded – a sorting algorithm based on binary comparisons requires at least $\lceil logn! \rceil$ comparisons.
- Using the fact $\lceil logn! \rceil$ is $\Theta(nlogn)$, we get

Complexity of Comparison based sorting algorithms

- We thus concluded – a sorting algorithm based on binary comparisons requires at least $\lceil log n! \rceil$ comparisons.
- Using the fact $\lceil log n! \rceil$ is $\Theta(nlogn)$, we get the number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(nlogn)$.

Complexity of Comparison based sorting algorithms

- We thus concluded – a sorting algorithm based on binary comparisons requires at least $\lceil log n! \rceil$ comparisons.
- Using the fact $\lceil log n! \rceil$ is $\Theta(n log n)$, we get the number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(n log n)$.
- We also gave a lower bound for average number of comparisons.

Tree Traversal

- Traversal algorithms – preorder, inorder and postorder traversal.

Tree Traversal

- Traversal algorithms – preorder, inorder and postorder traversal.
- Preorder – Root, left most subtree, the next tree from the left, and so on till the right most subtree.

Tree Traversal

- Traversal algorithms – preorder, inorder and postorder traversal.
- Preorder – Root, left most subtree, the next tree from the left, and so on till the right most subtree.
- Inorder –left most subtree, Root, the next tree from the left, and so on till the right most subtree.

Tree Traversal

- Traversal algorithms – preorder, inorder and postorder traversal.
- Preorder – Root, left most subtree, the next tree from the left, and so on till the right most subtree.
- Inorder –left most subtree, Root, the next tree from the left, and so on till the right most subtree.
- Postorder – left most subtree, the next tree from the left, and so on till the right most subtree, Root.

Tree Traversal

- Traversal algorithms – preorder, inorder and postorder traversal.
- Preorder – Root, left most subtree, the next tree from the left, and so on till the right most subtree.
- Inorder –left most subtree, Root, the next tree from the left, and so on till the right most subtree.
- Postorder – left most subtree, the next tree from the left, and so on till the right most subtree, Root.
- Inorder traversal of BST gives a sorted list.

Tree Traversal

- Traversal algorithms – preorder, inorder and postorder traversal.
- Preorder – Root, left most subtree, the next tree from the left, and so on till the right most subtree.
- Inorder –left most subtree, Root, the next tree from the left, and so on till the right most subtree.
- Postorder – left most subtree, the next tree from the left, and so on till the right most subtree, Root.
- Inorder traversal of BST gives a sorted list.
- Infix notation – inorder traversal of an expression where you include a paranthesis when you come across an operation.

Tree Traversal

- Traversal algorithms – preorder, inorder and postorder traversal.
- Preorder – Root, left most subtree, the next tree from the left, and so on till the right most subtree.
- Inorder –left most subtree, Root, the next tree from the left, and so on till the right most subtree.
- Postorder – left most subtree, the next tree from the left, and so on till the right most subtree, Root.
- Inorder traversal of BST gives a sorted list.
- Infix notation – inorder traversal of an expression where you include a paranthesis when you come across an operation.
- Else inorder traversal of an expression tree is ambiguous.

Postfix and Prefix Notation

- Prefix (Polish notation) and Postfix (Polish reverse notation)

Postfix and Prefix Notation

- Prefix (Polish notation) and Postfix (Polish reverse notation)
- Prefix – traverse the rooted tree in preorder.

Postfix and Prefix Notation

- Prefix (Polish notation) and Postfix (Polish reverse notation)
- Prefix – traverse the rooted tree in preorder.
- Postfix – traverse the rooted tree in postorder.

Postfix and Prefix Notation

- Prefix (Polish notation) and Postfix (Polish reverse notation)
- Prefix – traverse the rooted tree in preorder.
- Postfix – traverse the rooted tree in postorder.
- Both are unambiguous as long as you know the number of operands that the operator will take.

Postfix and Prefix Notation

- Prefix (Polish notation) and Postfix (Polish reverse notation)
- Prefix – traverse the rooted tree in preorder.
- Postfix – traverse the rooted tree in postorder.
- Both are unambiguous as long as you know the number of operands that the operator will take.
- When evaluating a prefix expression you work from right to left and for postfix expression you work from left to right.

Spanning Trees

Spanning Trees

- Given a graph (in real life a network say) we are looking to cover all the vertices of the graph ensuring there is a path between them all having minimum number of edges.

Spanning Trees

- Given a graph (in real life a network say) we are looking to cover all the vertices of the graph ensuring there is a path between them all having minimum number of edges.
- Such a graph is a tree - Let G be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.

Spanning Trees

- Given a graph (in real life a network say) we are looking to cover all the vertices of the graph ensuring there is a path between them all having minimum number of edges.
- Such a graph is a tree - Let G be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.
- A simple graph with a spanning tree must be connected, because there is a path in the spanning tree between any two vertices.

Spanning Trees

- Given a graph (in real life a network say) we are looking to cover all the vertices of the graph ensuring there is a path between them all having minimum number of edges.
- Such a graph is a tree - Let G be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.
- A simple graph with a spanning tree must be connected, because there is a path in the spanning tree between any two vertices.
- Also, every connected simple graph has a spanning tree.

Spanning Trees

Theorem

A simple graph is connected if and only if it has a spanning tree.

Proof:

Spanning Trees

Theorem

A simple graph is connected if and only if it has a spanning tree.

Proof:

- If a graph G has a spanning tree T, then T contains every vertex of G and there is a path between any two vertices – G is connected.

Spanning Trees

Theorem

A simple graph is connected if and only if it has a spanning tree.

Proof:

- If a graph G has a spanning tree T, then T contains every vertex of G and there is a path between any two vertices – G is connected.
- Suppose G is connected. If G is not a tree then G has a simple circuit.

Spanning Trees

Theorem

A simple graph is connected if and only if it has a spanning tree.

Proof:

- If a graph G has a spanning tree T, then T contains every vertex of G and there is a path between any two vertices – G is connected.
- Suppose G is connected. If G is not a tree then G has a simple circuit.
- Remove an edge from the circuits.

Spanning Trees

Theorem

A simple graph is connected if and only if it has a spanning tree.

Proof:

- If a graph G has a spanning tree T, then T contains every vertex of G and there is a path between any two vertices – G is connected.
- Suppose G is connected. If G is not a tree then G has a simple circuit.
- Remove an edge from the circuits.
- The subgraph is still connected.

Theorem

A simple graph is connected if and only if it has a spanning tree.

Proof:

- If a graph G has a spanning tree T, then T contains every vertex of G and there is a path between any two vertices – G is connected.
- Suppose G is connected. If G is not a tree then G has a simple circuit.
- Remove an edge from the circuits.
- The subgraph is still connected.
- If its not a tree, it has a simple circuit - again remove an edge till no simple circuits remain.

Spanning Trees

Theorem

A simple graph is connected if and only if it has a spanning tree.

Proof:

- If a graph G has a spanning tree T, then T contains every vertex of G and there is a path between any two vertices – G is connected.
- Suppose G is connected. If G is not a tree then G has a simple circuit.
- Remove an edge from the circuits.
- The subgraph is still connected.
- If its not a tree, it has a simple circuit - again remove an edge till no simple circuits remain.
- The graph is finite so this process will terminate and a tree is produced that contains all the vertices of G.

How to construct/find spanning trees?

- We have from previous theorem that you have to remove edges from simple circuits to build a spanning tree.

How to construct/find spanning trees?

- We have from previous theorem that you have to remove edges from simple circuits to build a spanning tree.
- That would mean identifying simple circuits - very inefficient!

How to construct/find spanning trees?

- We have from previous theorem that you have to remove edges from simple circuits to build a spanning tree.
- That would mean identifying simple circuits - very inefficient!
- The idea is instead of removing edges why not build by successively adding edges – Depth-first search and Breadth-first search.

How to construct/find spanning trees?

- We have from previous theorem that you have to remove edges from simple circuits to build a spanning tree.
- That would mean identifying simple circuits - very inefficient!
- The idea is instead of removing edges why not build by successively adding edges – Depth-first search and Breadth-first search.
- Depth-first search – Form a rooted tree by arbitrarily choosing a vertex as root.

How to construct/find spanning trees?

- We have from previous theorem that you have to remove edges from simple circuits to build a spanning tree.
- That would mean identifying simple circuits - very inefficient!
- The idea is instead of removing edges why not build by successively adding edges – Depth-first search and Breadth-first search.
- Depth-first search – Form a rooted tree by arbitrarily choosing a vertex as root.
- Form a path starting at this vertex by adding vertices and edges – each new edge is incident with the last vertex in the path and a vertex not already in the path.

Depth-First search

- Continue adding vertices and edges as long as possible.
- If all vertices are done, the tree consisting of this path is a spanning tree.

Depth-First search

- Continue adding vertices and edges as long as possible.
- If all vertices are done, the tree consisting of this path is a spanning tree.
- However, if the path does not go through all vertices, more vertices and edges must be added.

Depth-First search

- Continue adding vertices and edges as long as possible.
- If all vertices are done, the tree consisting of this path is a spanning tree.
- However, if the path does not go through all vertices, more vertices and edges must be added.
- Move back to the next to last vertex in the path, and, if possible, form a new path from here without covering vertices that were not already visited.

Depth-First search

- Continue adding vertices and edges as long as possible.
- If all vertices are done, the tree consisting of this path is a spanning tree.
- However, if the path does not go through all vertices, more vertices and edges must be added.
- Move back to the next to last vertex in the path, and, if possible, form a new path from here without covering vertices that were not already visited.
- If this cannot be done, move back another vertex in the path, that is, two vertices back in the path, and try again.

Depth-First search

- Continue adding vertices and edges as long as possible.
- If all vertices are done, the tree consisting of this path is a spanning tree.
- However, if the path does not go through all vertices, more vertices and edges must be added.
- Move back to the next to last vertex in the path, and, if possible, form a new path from here without covering vertices that were not already visited.
- If this cannot be done, move back another vertex in the path, that is, two vertices back in the path, and try again.
- Repeat until no more edges can be added. Note that its a finite graph!

Depth-First search

- Continue adding vertices and edges as long as possible.
- If all vertices are done, the tree consisting of this path is a spanning tree.
- However, if the path does not go through all vertices, more vertices and edges must be added.
- Move back to the next to last vertex in the path, and, if possible, form a new path from here without covering vertices that were not already visited.
- If this cannot be done, move back another vertex in the path, that is, two vertices back in the path, and try again.
- Repeat until no more edges can be added. Note that its a finite graph!
- Depth-first search is also called backtracking because the algorithm returns to vertices previously visited to add path.

Algorithm for Depth-First search

ALGORITHM 1 Depth-First Search.

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex v_1
visit(v_1)

procedure *visit*(v: vertex of G)
for each vertex w adjacent to v and not yet in T
  add vertex w and edge $\{v, w\}$ to T
  visit(w)

Algorithm for Depth-First search

- Things to prove :

Algorithm for Depth-First search

- Things to prove :
  - No circuits (but edges are between vertices not in the tree already).
  - The tree remains connected as it is built.
  - Because graph is connected, every vertex is visited and is added to the tree.

Algorithm for Depth-First search

- Things to prove :
  - No circuits (but edges are between vertices not in the tree already).
  - The tree remains connected as it is built.
  - Because graph is connected, every vertex is visited and is added to the tree.
- Complexity Analysis –

Algorithm for Depth-First search

- Things to prove :
    - No circuits (but edges are between vertices not in the tree already).
    - The tree remains connected as it is built.
    - Because graph is connected, every vertex is visited and is added to the tree.
- Complexity Analysis –
    - $visit(v)$ is called only when the v is first encountered.

Algorithm for Depth-First search

- Things to prove :
  - No circuits (but edges are between vertices not in the tree already).
  - The tree remains connected as it is built.
  - Because graph is connected, every vertex is visited and is added to the tree.
- Complexity Analysis –
  - $visit(v)$ is called only when the v is first encountered.
  - Finding adjacent vertices of v is just a lookup of adjacency matrix.

Algorithm for Depth-First search

- Things to prove :
  - No circuits (but edges are between vertices not in the tree already).
  - The tree remains connected as it is built.
  - Because graph is connected, every vertex is visited and is added to the tree.
- Complexity Analysis –
  - $visit(v)$ is called only when the v is first encountered.
  - Finding adjacent vertices of v is just a lookup of adjacency matrix.
  - Each edge is examined at most twice – to add this edge and one of its endpoints to the tree.

Algorithm for Depth-First search

- Things to prove :
  - No circuits (but edges are between vertices not in the tree already).
  - The tree remains connected as it is built.
  - Because graph is connected, every vertex is visited and is added to the tree.
- Complexity Analysis –
  - $visit(v)$ is called only when the v is first encountered.
  - Finding adjacent vertices of v is just a lookup of adjacency matrix.
  - Each edge is examined at most twice – to add this edge and one of its endpoints to the tree.
  - So DFS constructs a spanning tree in $O(e)$ steps.

Algorithm for Depth-First search

- Things to prove :
  - No circuits (but edges are between vertices not in the tree already).
  - The tree remains connected as it is built.
  - Because graph is connected, every vertex is visited and is added to the tree.
- Complexity Analysis –
  - $visit(v)$ is called only when the v is first encountered.
  - Finding adjacent vertices of v is just a lookup of adjacency matrix.
  - Each edge is examined at most twice – to add this edge and one of its endpoints to the tree.
  - So DFS constructs a spanning tree in $O(e)$ steps.
  - For a simple graph $e < n(n-1)/2$ so $O(n^2)$ steps.

Breadth-First search

- Arbitrarily choose a root and add all edges incident to this vertex.

Breadth-First search

- Arbitrarily choose a root and add all edges incident to this vertex.
- That is, they are finally the vertices at level 1 in the spanning tree.

Breadth-First search

- Arbitrarily choose a root and add all edges incident to this vertex.
- That is, they are finally the vertices at level 1 in the spanning tree.
- Order them in some way and visit each vertex at level 1 in order and add each edge incident to this vertex to the tree,

Breadth-First search

- Arbitrarily choose a root and add all edges incident to this vertex.
- That is, they are finally the vertices at level 1 in the spanning tree.
- Order them in some way and visit each vertex at level 1 in order and add each edge incident to this vertex to the tree, provided it doesnt create a circuit.

Breadth-First search

- Arbitrarily choose a root and add all edges incident to this vertex.
- That is, they are finally the vertices at level 1 in the spanning tree.
- Order them in some way and visit each vertex at level 1 in order and add each edge incident to this vertex to the tree, provided it doesnt create a circuit.
- Then order these vertices -they are level 2. Continue till it can't be done!

Algorithm for Breadth-First search

ALGORITHM 2 Breadth-First Search.

procedure *BFS* (G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of vertex v_1
L := empty list
put v_1 in the list L of unprocessed vertices
while L is not empty
  remove the first vertex, v, from L
  for each neighbor w of v
    if w is not in L and not in T **then**
      add w to the end of the list L
      add w and edge $\{v, w\}$ to T

Complexity Analysis for Breadth-First search

- Again with adjacency lists or matrices no computation is needed to determine the adjacent vertices to a given vertex.

Complexity Analysis for Breadth-First search

- Again with adjacency lists or matrices no computation is needed to determine the adjacent vertices to a given vertex.
- Here also each edge is checked at most twice to determine whether the edge is to be added or not and if its end point is already in tree or not.

Complexity Analysis for Breadth-First search

- Again with adjacency lists or matrices no computation is needed to determine the adjacent vertices to a given vertex.
- Here also each edge is checked at most twice to determine whether the edge is to be added or not and if its end point is already in tree or not.
- So again here as well BFS is $O(e)$ or $O(n^2)$.

Backtracking Applications

- For problems that need an exhaustive search of all possible outcomes.

Backtracking Applications

- For problems that need an exhaustive search of all possible outcomes.
- A decision tree – each internal vertex a decision and leaf a solution.

Backtracking Applications

- For problems that need an exhaustive search of all possible outcomes.
- A decision tree – each internal vertex a decision and leaf a solution.
- First we go down a sequence a decisions and see if we get to a solution or else backtrack to work towards a solution with another series of decisions.

Backtracking Applications

- For problems that need an exhaustive search of all possible outcomes.
- A decision tree – each internal vertex a decision and leaf a solution.
- First we go down a sequence a decisions and see if we get to a solution or else backtrack to work towards a solution with another series of decisions.
- Example – graph colouring– How can backtracking be used to decide whether a graph can be colored using n colors?

Backtracking Applications

- Pick some vertex a and give color 1.

Backtracking Applications

- Pick some vertex a and give color 1.
- Then pick another vertex b - if b is adjacent to a then give it color 2 else give it the same color 1.

Backtracking Applications

- Pick some vertex a and give color 1.
- Then pick another vertex b - if b is adjacent to a then give it color 2 else give it the same color 1.
- Then go to another vertex c.

Backtracking Applications

- Pick some vertex a and give color 1.
- Then pick another vertex b - if b is adjacent to a then give it color 2 else give it the same color 1.
- Then go to another vertex c.
- Only if neither color 1 nor color 2 can be used should we bring in color 3.

Backtracking Applications

- Pick some vertex a and give color 1.
- Then pick another vertex b - if b is adjacent to a then give it color 2 else give it the same color 1.
- Then go to another vertex c.
- Only if neither color 1 nor color 2 can be used should we bring in color 3.
- Continue this as long as it is possible to assign one of the n colors to each additional vertex always going for the first allowable color.

Backtracking Applications

- If we cannot be colored by any of the n colors - backtrack to the last assignment made and change the coloring of the last vertex colored using the next allowable color in the list.

Backtracking Applications

- If we cannot be colored by any of the n colors - backtrack to the last assignment made and change the coloring of the last vertex colored using the next allowable color in the list.
- If it is not possible to change this coloring then backtrack further to previous assignments one step at a time.

Backtracking Applications

- If we cannot be colored by any of the n colors - backtrack to the last assignment made and change the coloring of the last vertex colored using the next allowable color in the list.
- If it is not possible to change this coloring then backtrack further to previous assignments one step at a time.
- If a coloring using n colors exists, backtracking will produce it, however inefficient.

Backtracking Applications

- If we cannot be colored by any of the n colors - backtrack to the last assignment made and change the coloring of the last vertex colored using the next allowable color in the list.
- If it is not possible to change this coloring then backtrack further to previous assignments one step at a time.
- If a coloring using n colors exists, backtracking will produce it, however inefficient.
- Example with 3 coloring –

Minimum Spanning Trees

Minimum Spanning Trees

- The edges have weights and we need to construct a spanning tree where the sum of weights of the edges are minimized.

Minimum Spanning Trees

- The edges have weights and we need to construct a spanning tree where the sum of weights of the edges are minimized.
- A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

Minimum Spanning Trees

- The edges have weights and we need to construct a spanning tree where the sum of weights of the edges are minimized.
- A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.
- We discuss two algorithms to construct a MST – Prim's algorithm (originally Jarnik's) and Kruskal algorithm.

Prim's Algorithm

- Choose the edge with smallest weight and put it into the spanning tree.

Prim's Algorithm

- Choose the edge with smallest weight and put it into the spanning tree.
- Keep adding edges of minimum weight that are incident to a vertex already in the tree

Prim's Algorithm

- Choose the edge with smallest weight and put it into the spanning tree.
- Keep adding edges of minimum weight that are incident to a vertex already in the tree but never forming a circuit.

Prim's Algorithm

- Choose the edge with smallest weight and put it into the spanning tree.
- Keep adding edges of minimum weight that are incident to a vertex already in the tree but never forming a circuit.
- Stop when $n - 1$ edges have been added.

Prim's Algorithm

- Choose the edge with smallest weight and put it into the spanning tree.
- Keep adding edges of minimum weight that are incident to a vertex already in the tree but never forming a circuit.
- Stop when $n - 1$ edges have been added.
- Proof that Prim's algorithm is correct – described in detail in Rosen.
- Yes what happens when we have edges with same weight – we need to order in a way that it is deterministic.

Prim's Algorithm

- Choose the edge with smallest weight and put it into the spanning tree.
- Keep adding edges of minimum weight that are incident to a vertex already in the tree but never forming a circuit.
- Stop when $n-1$ edges have been added.
- Proof that Prim's algorithm is correct – described in detail in Rosen.
- Yes what happens when we have edges with same weight – we need to order in a way that it is deterministic.
- Exercise – There could be more than one MST for a connected weighted simple graph.

Kruskal's Algorithm

- Formulated by Joseph Kruskal.

Kruskal's Algorithm

- Formulated by Joseph Kruskal.
- Choose an edge with minimum weight.

Kruskal's Algorithm

- Formulated by Joseph Kruskal.
- Choose an edge with minimum weight.
- Keep adding edges of minimum weight that do not form a circuit with those edges that are already chosen.

Kruskal's Algorithm

- Formulated by Joseph Kruskal.
- Choose an edge with minimum weight.
- Keep adding edges of minimum weight that do not form a circuit with those edges that are already chosen.
- Stop when $n - 1$ edges have been added.

Kruskal's Algorithm

- Formulated by Joseph Kruskal.
- Choose an edge with minimum weight.
- Keep adding edges of minimum weight that do not form a circuit with those edges that are already chosen.
- Stop when $n - 1$ edges have been added.
- The key difference here – in Prim's we choose edges that are min weight and that are incident to a vertex already in the tree and not forming a circuit.

Kruskal's Algorithm

- Formulated by Joseph Kruskal.
- Choose an edge with minimum weight.
- Keep adding edges of minimum weight that do not form a circuit with those edges that are already chosen.
- Stop when $n - 1$ edges have been added.
- The key difference here – in Prim's we choose edges that are min weight and that are incident to a vertex already in the tree and not forming a circuit.
- In Kruskal's we add min edges that are not necessarily incident to a vertex already in the tree i.e. we may create a forest.

Kruskal's Algorithm

- Formulated by Joseph Kruskal.
- Choose an edge with minimum weight.
- Keep adding edges of minimum weight that do not form a circuit with those edges that are already chosen.
- Stop when $n - 1$ edges have been added.
- The key difference here – in Prim's we choose edges that are min weight and that are incident to a vertex already in the tree and not forming a circuit.
- In Kruskal's we add min edges that are not necessarily incident to a vertex already in the tree i.e. we may create a forest.
- Complexity of Prim's algorithm – $O(mlog\ n)$ where m is the no of edges and n no of vertices.

Kruskal's Algorithm

- Formulated by Joseph Kruskal.
- Choose an edge with minimum weight.
- Keep adding edges of minimum weight that do not form a circuit with those edges that are already chosen.
- Stop when $n-1$ edges have been added.
- The key difference here – in Prim's we choose edges that are min weight and that are incident to a vertex already in the tree and not forming a circuit.
- In Kruskal's we add min edges that are not necessarily incident to a vertex already in the tree i.e. we may create a forest.
- Complexity of Prim's algorithm – $O(mlog\ n)$ where m is the no of edges and n no of vertices.
- Complexity of Kruskal's algorithm - $O(mlog m)$ – preferable for sparse graphs where $m << n(n-1)/2$.

Kruskal's Algorithm - Proof that it is correct

- T is a spanning tree –
    - T is a forest - no cycles are created.

Kruskal's Algorithm - Proof that it is correct

- T is a spanning tree –
  - T is a forest - no cycles are created.
  - T is spanning - if not there is a vertex v that is not incident with the edges in T. But then the incident edge of v must have been considered at some step. The first edge in the order would have been added since it would not have created a circuit.

Kruskal's Algorithm - Proof that it is correct

- T is a spanning tree –
  - T is a forest - no cycles are created.
  - T is spanning - if not there is a vertex v that is not incident with the edges in T. But then the incident edge of v must have been considered at some step. The first edge in the order would have been added since it would not have created a circuit.
  - T is connected - if not, then there are two or more connected components in T. Since the graph G is connected, there are edges that connect these components that are in G and not in T.

Kruskal's Algorithm - Proof that it is correct

- T is a spanning tree –
  - T is a forest - no cycles are created.
  - T is spanning - if not there is a vertex v that is not incident with the edges in T. But then the incident edge of v must have been considered at some step. The first edge in the order would have been added since it would not have created a circuit.
  - T is connected - if not, then there are two or more connected components in T. Since the graph G is connected, there are edges that connect these components that are in G and not in T. But first of these edges would have been added in T since they wont form a cycle.

Kruskal's Algorithm - Proof that it is correct

- T is a spanning tree –
  - T is a forest - no cycles are created.
  - T is spanning - if not there is a vertex v that is not incident with the edges in T. But then the incident edge of v must have been considered at some step. The first edge in the order would have been added since it would not have created a circuit.
  - T is connected - if not, then there are two or more connected components in T. Since the graph G is connected, there are edges that connect these components that are in G and not in T. But first of these edges would have been added in T since they wont form a cycle.
- Now to show its of minimum weight.

Kruskal's Algorithm - Proof that it is correct

- Let T^* be the tree of minimum spanning tree.

Kruskal's Algorithm - Proof that it is correct

- Let T^* be the tree of minimum spanning tree. If $T = T^*$ then its done. Let $T \neq T^*$.

Kruskal's Algorithm - Proof that it is correct

- Let T^* be the tree of minimum spanning tree. If $T = T^*$ then its done. Let $T \neq T^*$.
- There exists an edge $e \in T^*$ of min weight not in T and $T \cup e$ contains a circuit C.

Kruskal's Algorithm - Proof that it is correct

- Let T^* be the tree of minimum spanning tree. If $T = T^*$ then its done. Let $T \neq T^*$.
- There exists an edge $e \in T^*$ of min weight not in T and $T \cup e$ contains a circuit C.
- Some observations:
  - Every edge in C is of weight less than or equal to $wt(e)$ by construction.
  - There is some edge $e^{'}$ in C not in T^* since T^* has no cycles.

Kruskal's Algorithm - Proof that it is correct

- Consider $T_2 = T \setminus \{e'\} \cup \{e\}$.
  - T_2 is a spanning tree.
  - T_2 has more edges in common with T^* than T did.
  - $wt(T_2) \geq wt(T)$ - since we exchanged it for a more or equal expensive edge.
- We can now work with T_2 to find a spanning tree T_3 with more edges in common with T^*.

Kruskal's Algorithm - Proof that it is correct

- Consider $T_2 = T \setminus \{e'\} \cup \{e\}$.
  - T_2 is a spanning tree.
  - T_2 has more edges in common with T^* than T did.
  - $wt(T_2) \geq wt(T)$ - since we exchanged it for a more or equal expensive edge.
- We can now work with T_2 to find a spanning tree T_3 with more edges in common with T^*.
- We continue this process till we reach T^*.

Kruskal's Algorithm - Proof that it is correct

- Consider $T_2 = T \setminus \{e^{'}\} \cup \{e\}$.
  - T_2 is a spanning tree.
  - T_2 has more edges in common with T^* than T did.
  - $wt(T_2) \geq wt(T)$ - since we exchanged it for a more or equal expensive edge.
- We can now work with T_2 to find a spanning tree T_3 with more edges in common with T^*.
- We continue this process till we reach T^*.

$$wt(T) \leq wt(T_2) \leq wt(T_3) \leq \cdots \leq wt(T^*).$$

Kruskal's Algorithm - Proof that it is correct

- Consider $T_2 = T \setminus \{e'\} \cup \{e\}$.
  - T_2 is a spanning tree.
  - T_2 has more edges in common with T^* than T did.
  - $wt(T_2) \geq wt(T)$ - since we exchanged it for a more or equal expensive edge.
- We can now work with T_2 to find a spanning tree T_3 with more edges in common with T^*.
- We continue this process till we reach T^*.

$$wt(T) \leq wt(T_2) \leq wt(T_3) \leq \cdots \leq wt(T^*).$$

- Since T^* is of minimum weight this implies that the inequalities are really equalities and T is a MST.