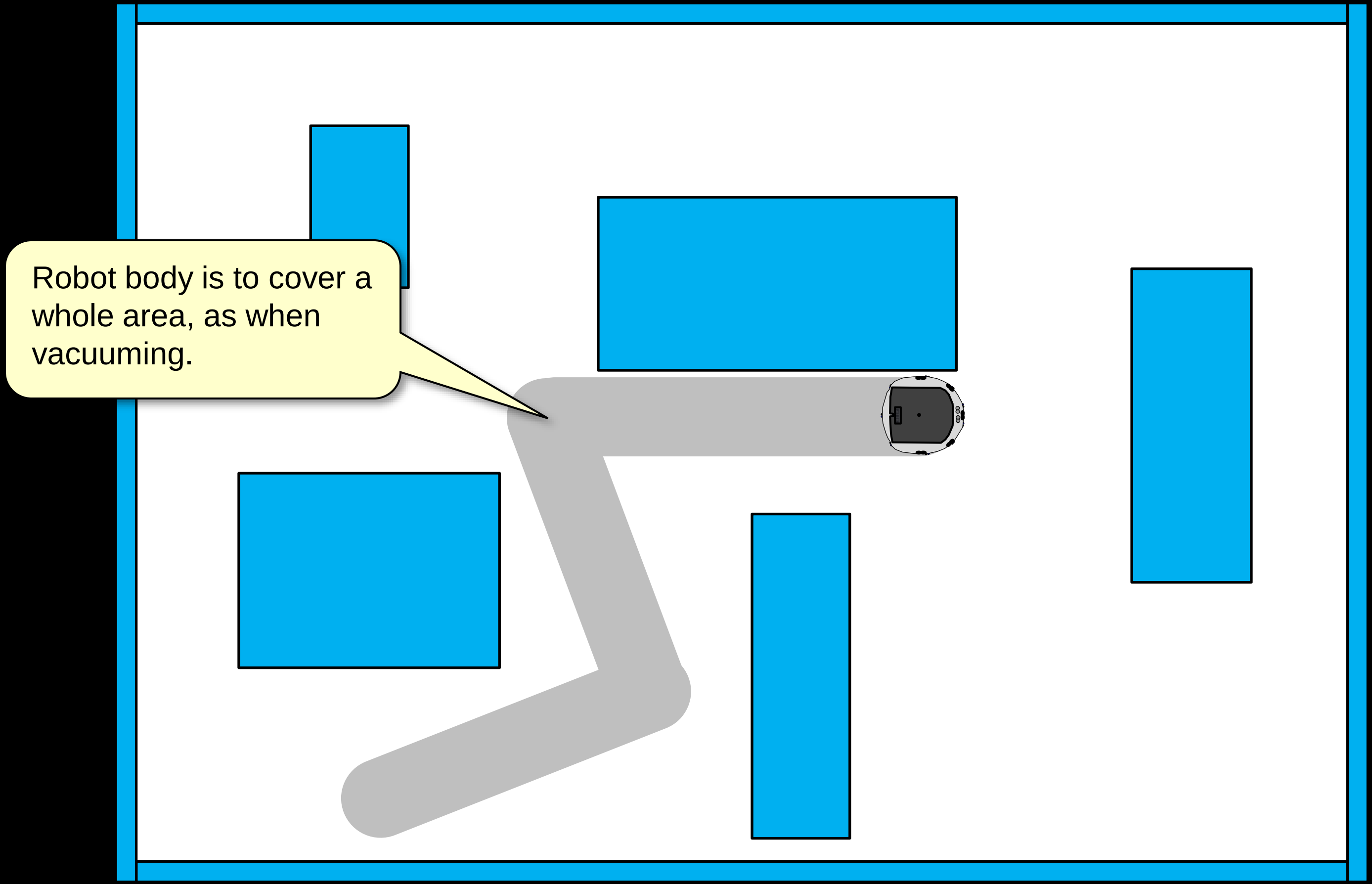
Area Coverage Paths

Area Coverage

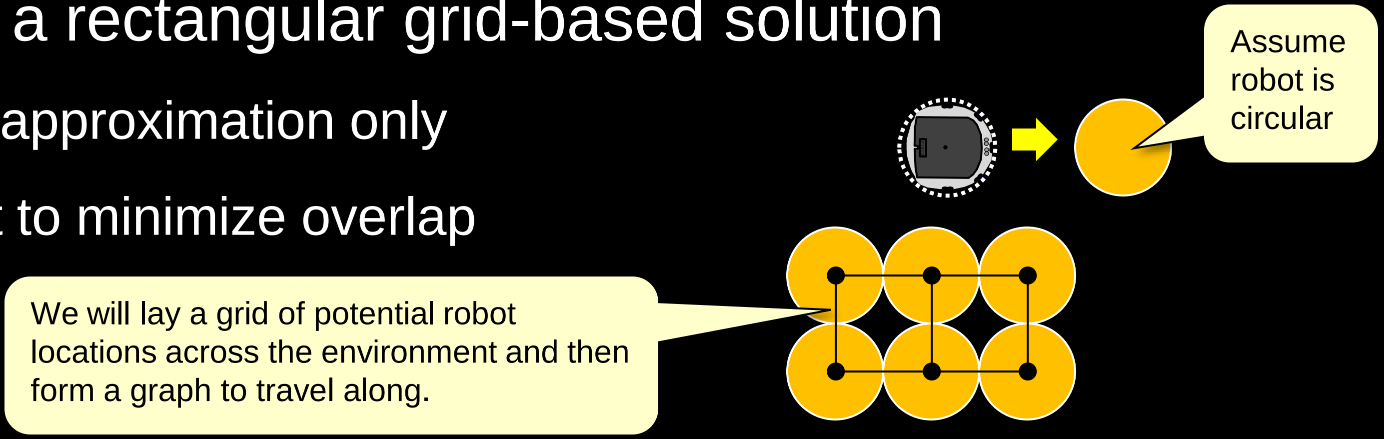
- How do we get a robot to cover the whole area of an environment ?



Area Coverage

- No simple perfect solution
  - Many algorithms of various complexities
- Main goals are:
  - cover all reachable areas of environment
  - minimize amount of overlapping

- Robots are not perfectly accurate, approximate solution is ok
- We will discus a rectangular grid-based solution
  - will provide an approximation only
  - will not attempt to minimize overlap



Grid Creation

- Overlay a 2D grid (i.e., graph of nodes & edges) of "robot-sized" circles touching adjacently vertically and horizontally:

Vertex of graph is center of circle/robot.

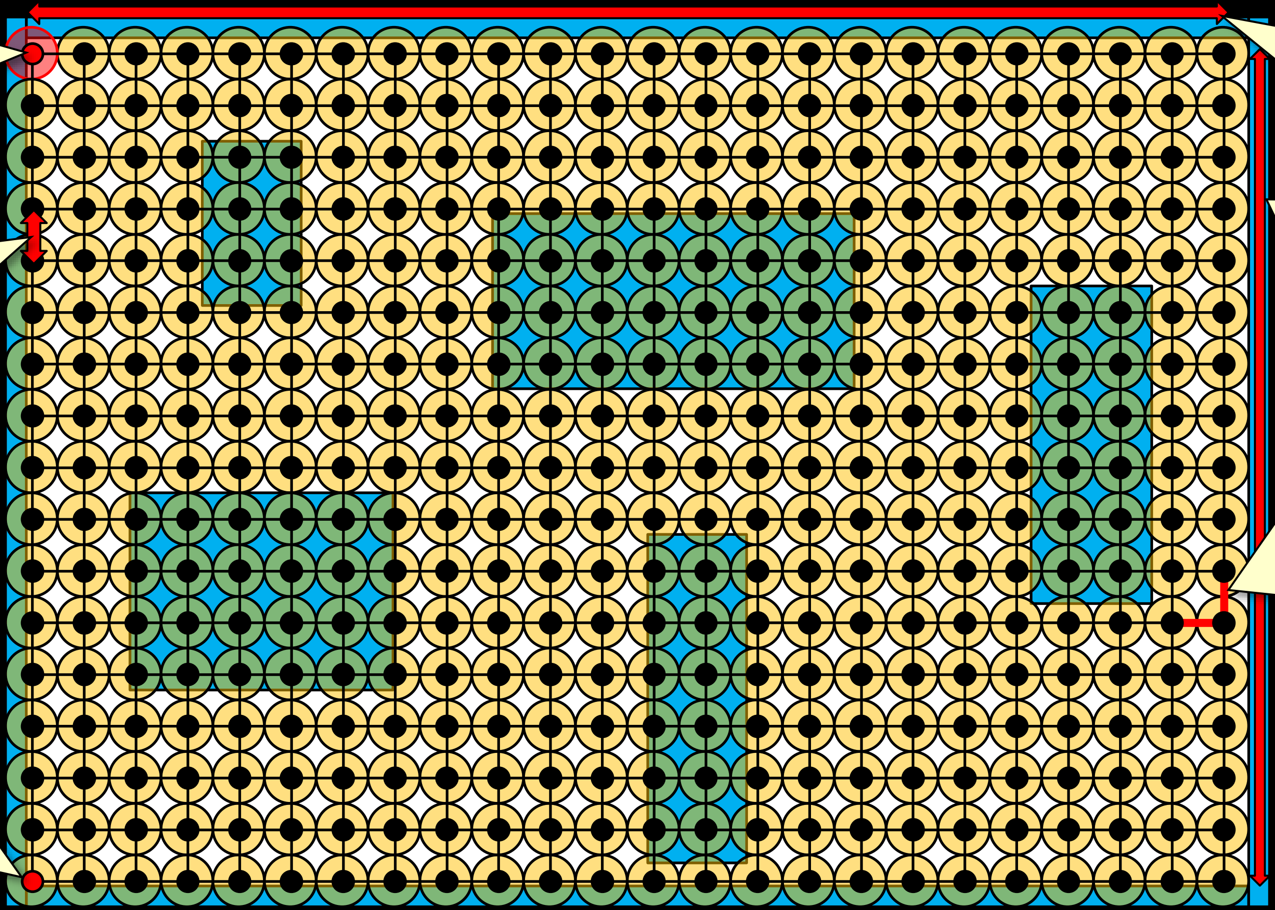
Distance between vertices is Robot's Diameter.

First vertex placed such that

x = Radius

and

y = Radius



NumCols = $\frac{\text{Width}}{\text{Robot Diameter}}$

NumRows = $\frac{\text{Height}}{\text{Robot Diameter}}$

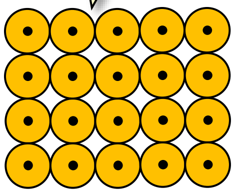
Add a graph edge between vertices that are adjacent vertically as well as horizontally.

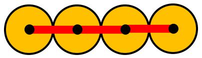
Grid Creation Pseudocode

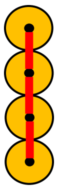
- The code for creating the grid is basic:

```
g = an empty graph

numRows = EnvironmentHeight / ROBOT_DIAMETER
numCols = EnvironmentWidth / ROBOT_DIAMETER

nodes = an empty 2D array that is numRows x numCols in size
FOR each row r DO
   FOR each column c DO
      nodes[r][c] = new node centered at that r and c
      add nodes[r][c] to g


FOR each row r DO
   FOR each column c (except the last one) DO
      add edge in g from add nodes[r][c] to nodes[r][c+1]

FOR each row r (except the last one) DO
   FOR each column c DO
      add edge in g from add nodes[r][c] to nodes[r+1][c]
```

Distance between vertices is robot diameter.

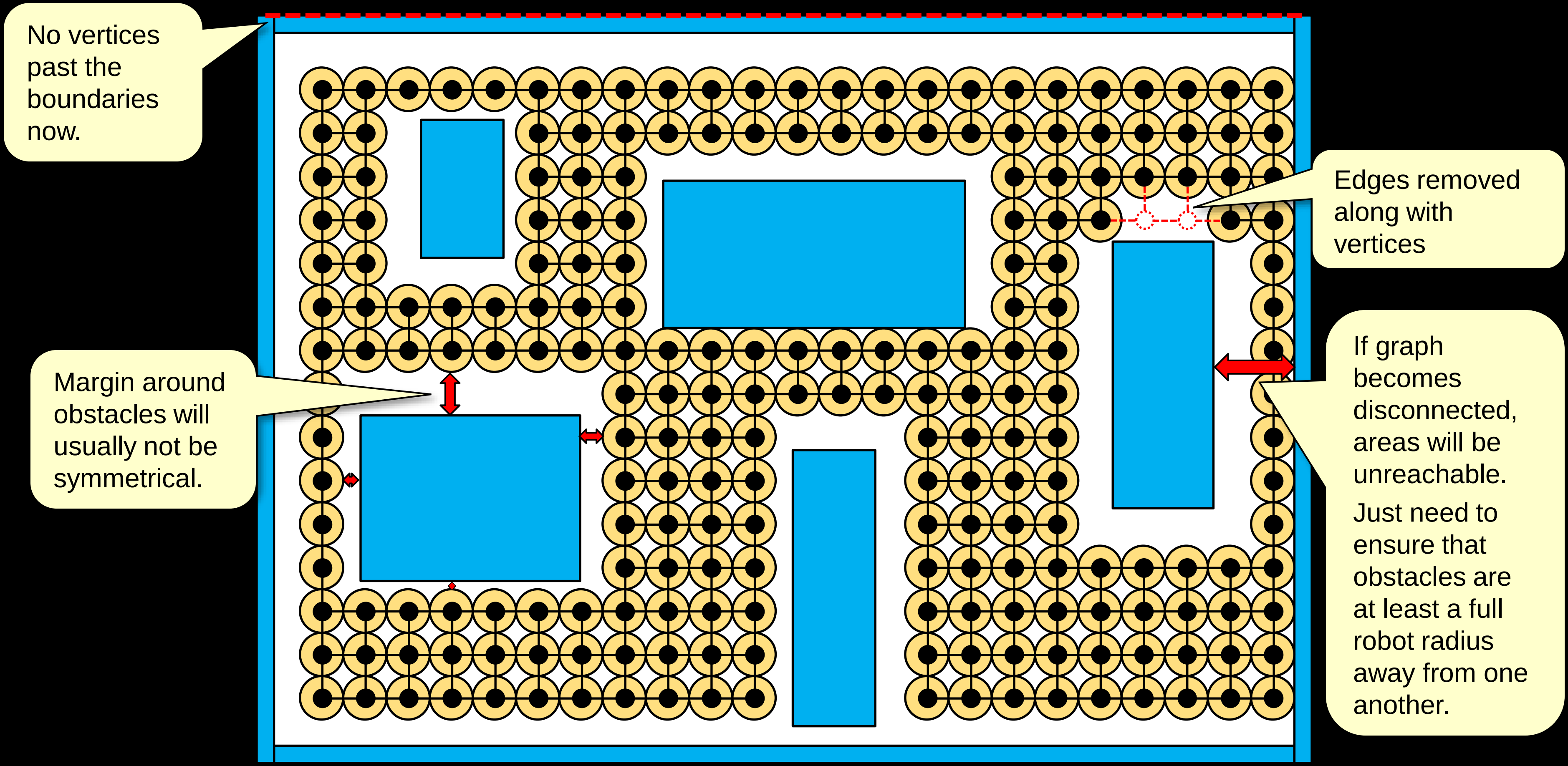






Grid Reduction

- Remove all vertices that represent robot positions that intersect with any obstcales:



Grid Reduction Pseudocode

- The reduced graph simply requires you to check if a node's circle intersects an obstacle:

```
FOR each node n of the graph DO {
   FOR each obstacle obj in the environment DO {
      IF the n's center lies within the obstacle THEN
         mark n as invalid
      FOR each edge e of obj DO {
         IF distance from n's center to e is <= ROBOT_RADIUS THEN
            mark n as invalid
      }
   }
}
FOR each invalid node n DO
   remove n from the graph
```

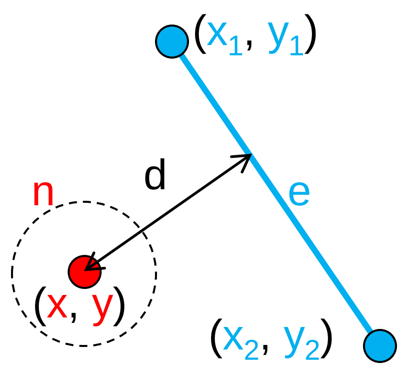
calculate $t = -\dfrac{(x_1 - x)(x_2 - x_1) + (y_1 - y)(y_2 - y_1)}{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

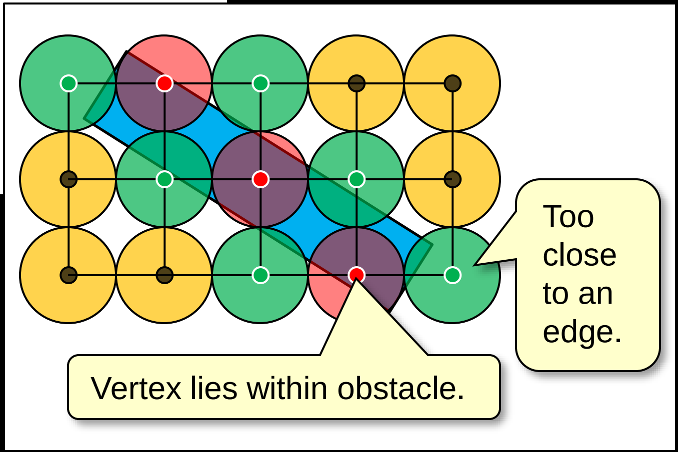
IF $(0 \le t \le 1)$ **THEN**

$$d = \frac{\left|(x_2 - x_1)(y_1 - y) - (y_2 - y_1)(x_1 - x)\right|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

OTHERWISE **d** is the smallest of these two:

$$\sqrt{(x_1 - x)^2 + (y_1 - y)^2}$$
$$\sqrt{(x_2 - x)^2 + (y_2 - y)^2}$$



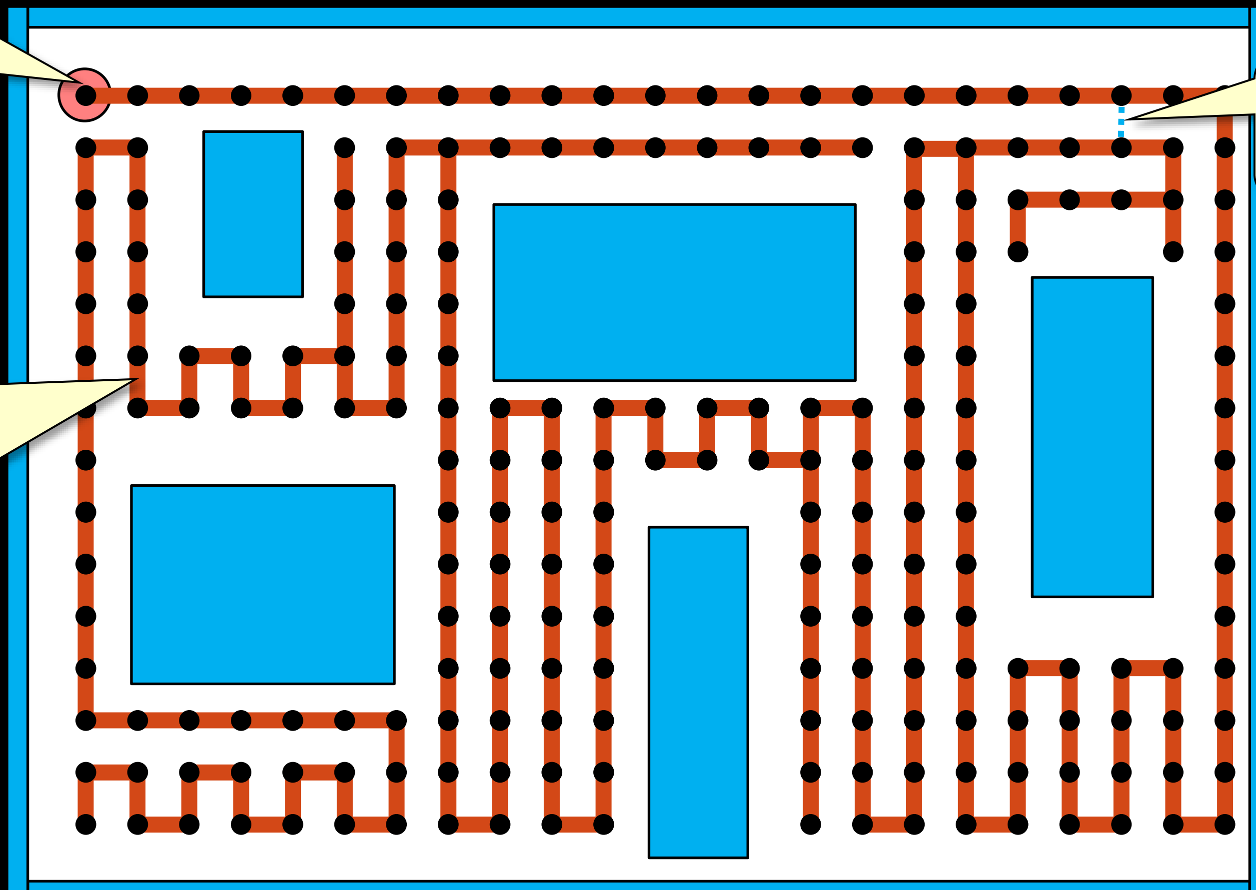


Spanning Tree

- Compute a *spanning tree* in the graph (shown in red):
  – represents a path that covers all the vertices.

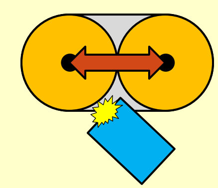
In this example, this is root of the spanning tree.

There are many possible spanning trees. This one follows an up, right, down, left ordering traversal.



Many graph edges have been removed

This algorithm assumes that no object lies between adjacent vertex circles so that robot can travel between them without collision:



Spanning Tree Pseudocode

- Spanning tree can be any traversal of the graph that reaches all nodes. It will depend on which edges are traversed first at each vertex.

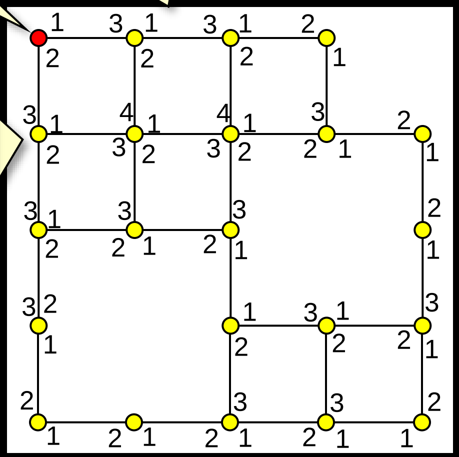
By marking a node as “visited”, we can avoid processing that node again and this is essential to stop the recursion.

```
computeSpanningTree(G) {
    FOR each node n of graph G DO
        mark n as "not visited"
    startNode = any node in G
    dummyEdge = an edge from startNode to itself
    computeSpanningTreeFrom(startNode, dummyEdge)

    edges = all edges that are not marked as part of tree
    FOR each edge e in edges DO
        remove e from G
}


computeSpanningTreeFrom(aNode, incomingEdge) {
    IF aNode was already visited THEN
        RETURN

    mark aNode as "visited"
    mark incomingEdge as part of the spanning tree

    FOR each edge e connected to aNode DO
        otherNode = node of edge e that is not aNode
        computeSpanningTreeFrom(otherNode, e)
}
```

The ordering of edges in this example is right, down, left, up.

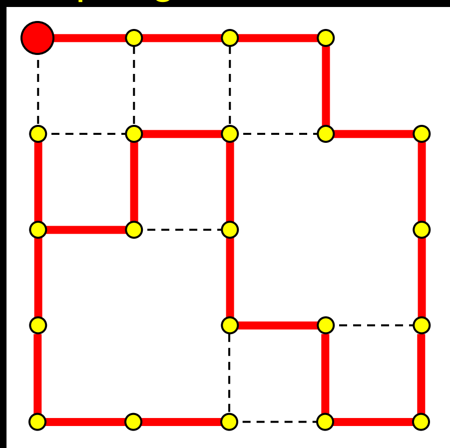
startNode

Each vertex has its edges in some order. Neighbours are visited in that order.

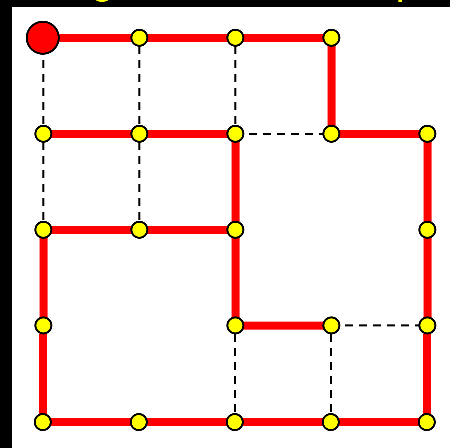


Various Spanning Trees

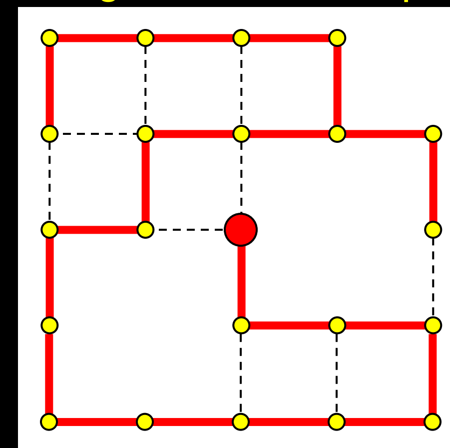
Up/Right/Down/Left



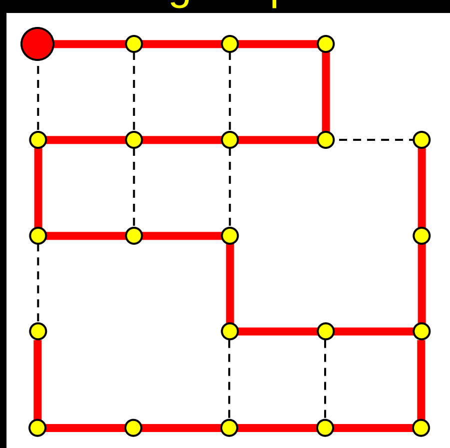
Right/Down/Left/Up



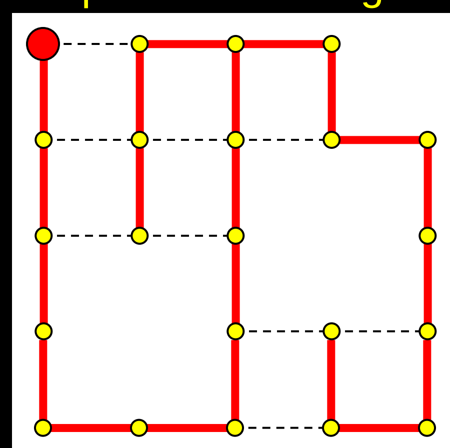
Right/Down/Left/Up



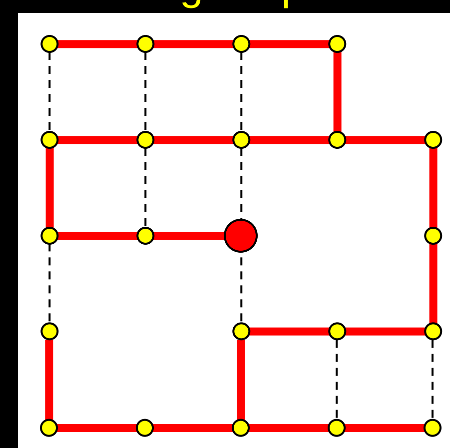
Left/Right/Up/Down



Up/Down/Left/Right

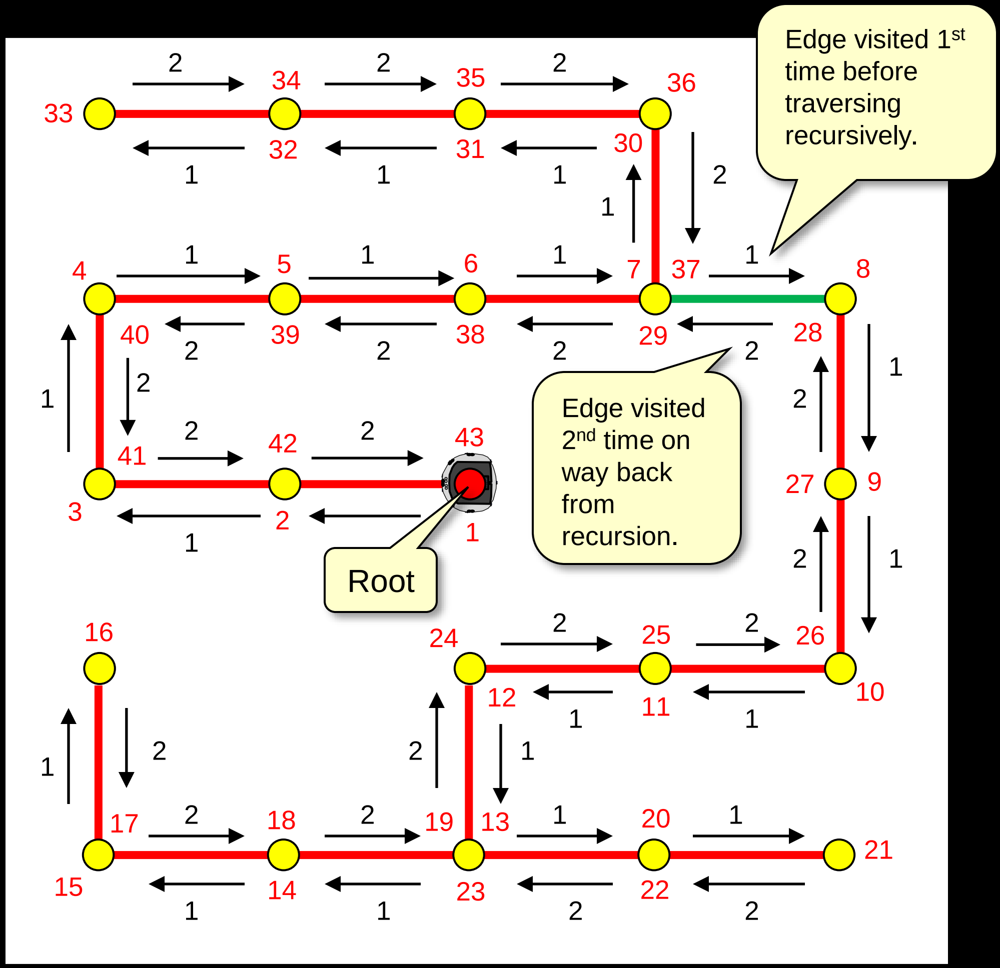


Left/Right/Up/Down



Spanning Tree Traversal

- Need to compute a path in the spanning tree, recursively.
- Each edge needs to be travelled on twice, which means that each vertex needs to be added to the path as many times as it has edges.
- Path in this example will have 43 points on it.
- Robot travels from point to point.



Spanning Tree Traversal

- To travel along spanning tree, we need to compute a path:

We build up a **path** in the tree which will represent the ordering of nodes to visit by the robot.

Instead of "visited" boolean, keep the previous node that is the node's parent in the spanning tree.

We will assume that the robot can travel from its start location to the start node without collision.

```
computeSpanningTree(G) {
    ... Compute spanning as before

    FOR each node n of graph G DO
        set previous of n to NULL
    set path to be an empty list
    dummyEdge = an edge from startNode to itself
    computeCoveragePathFrom(startNode, dummyEdge)
}
```

Each time we arrive at a **aNode** that has no previous value set, it is a new node in the path, so add it to the **path**.

path is a global variable

Set **aNode**'s previous to be the other end of the edge that we came in on.

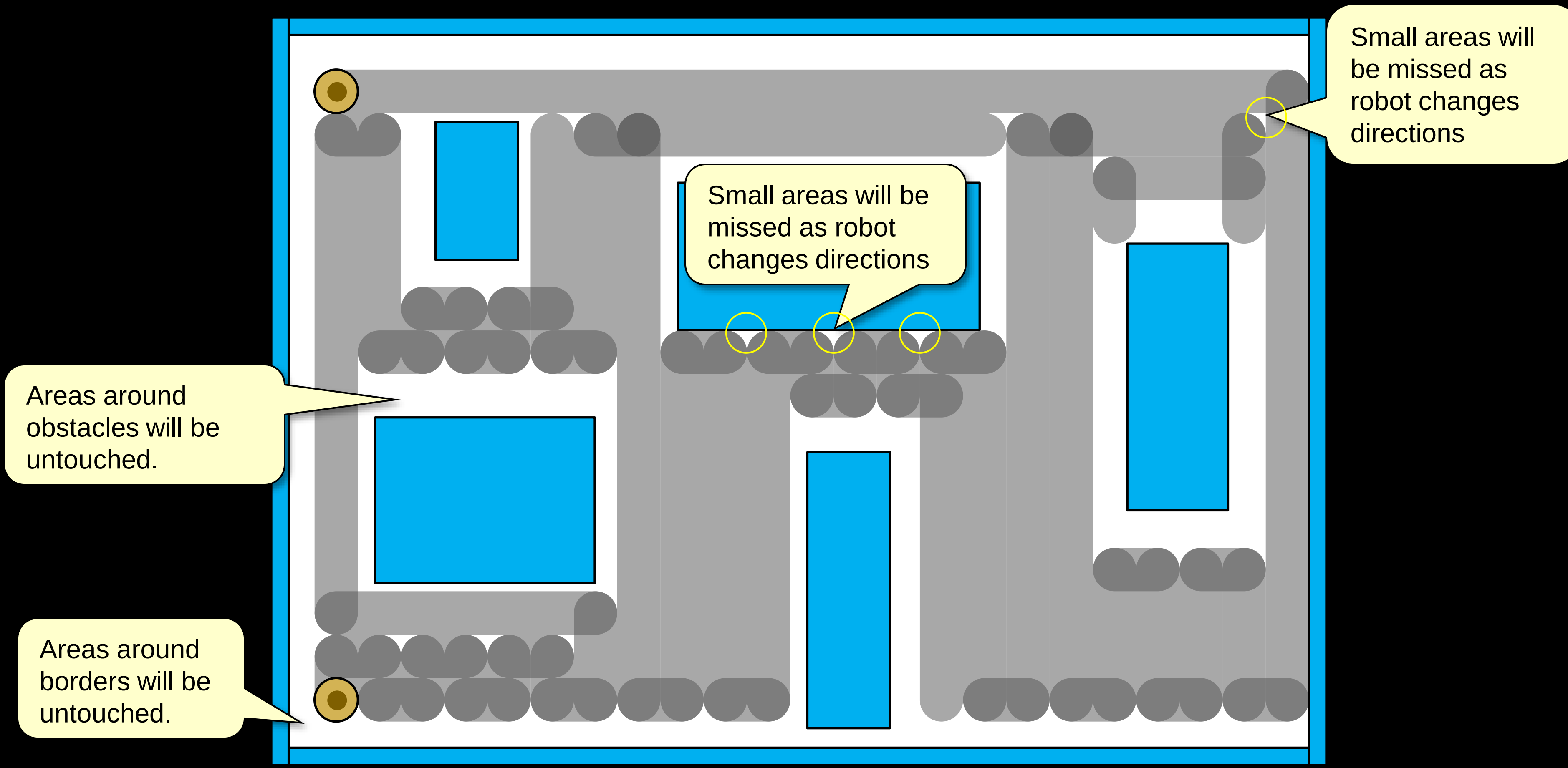
Make sure that the path goes back to the previous node.

Ensures that we are following spanning tree on way back from recursion.

```
computeCoveragePathFrom(aNode, incomingEdge) {
    IF previous of aNode is not NULL THEN
        RETURN

    add aNode's location to the path

    set previous of aNode to node at other end of incomingEdge

    FOR each neighbour neigh of aNode DO {
        edge = the edge that connects aNode and neigh
        computeCoveragePathFrom(neigh, edge)
    }
    add location of aNode's previous to the path
}
```

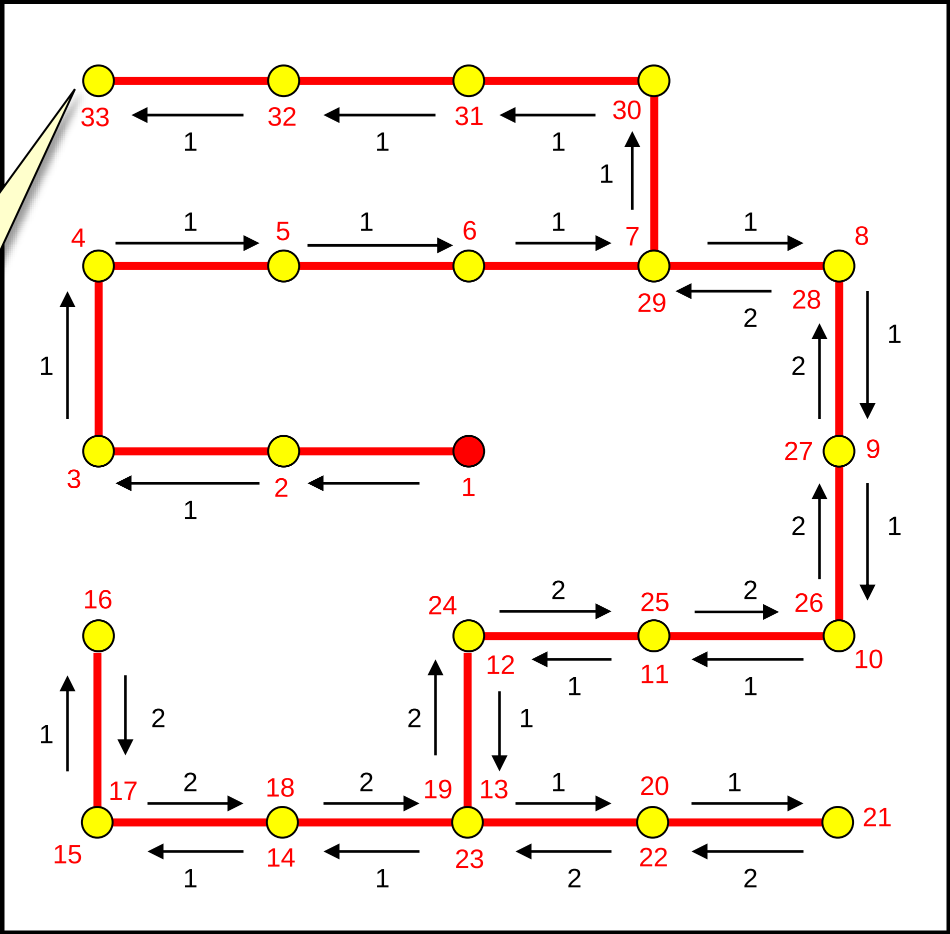
Spanning Tree Coverage

- Traveling along the spanning tree will cover most of the environment (except around borders of obstacles):



More Efficient Travelling

- Currently, robot travels back to root, but this is not necessary.
- Better if we stop when travelled on each edge once.
- Result is that path is shorter with less points.



Robot stops here now. It does not need to go back to the root node.

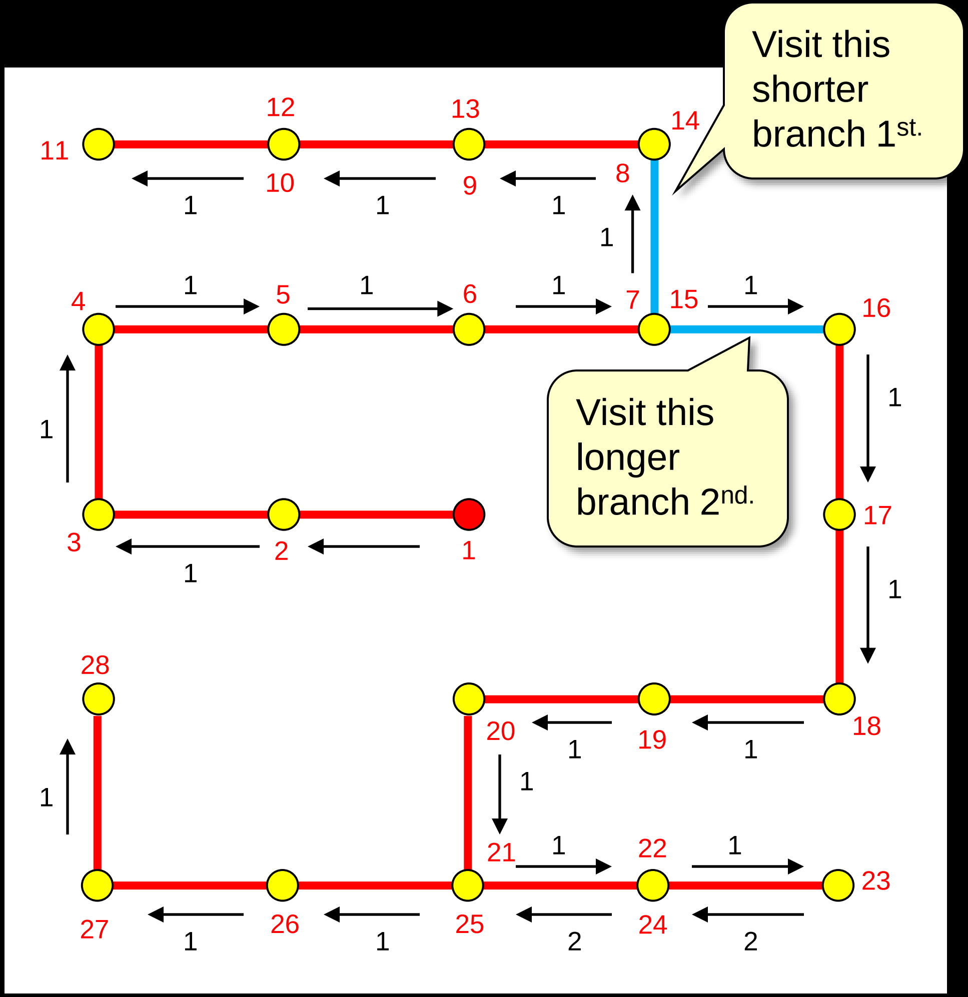
Path has only 33 points … which is 23% shorter!

More Efficient Travelling

- Can shorten even more if we visit the smaller branches before the longer ones.

- Results in less travel since we don't have to travel a second time on the longer branches

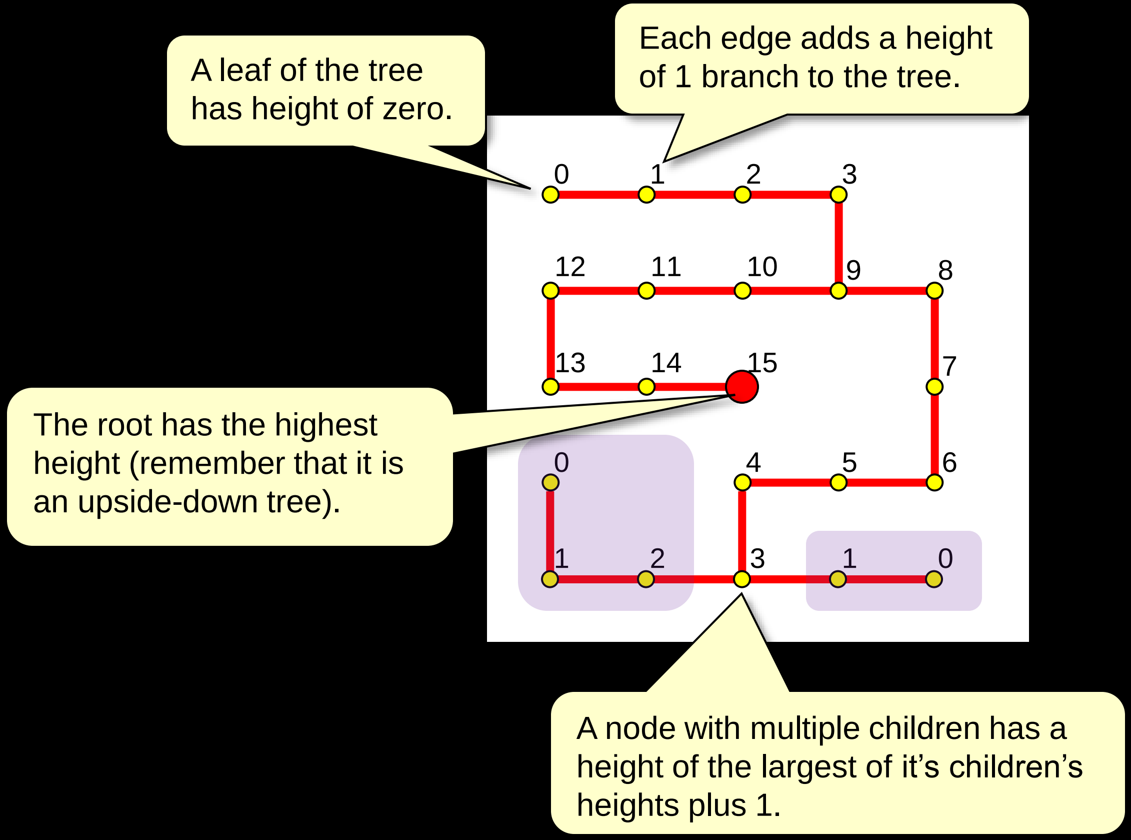
- Requires us to visit the shorter branches of the tree first



Path has only 28 points now … which is 35% shorter than original!

Tree Height

- If we want to visit certain branches of the spanning tree in some specific order (e.g., smallest first), then we need to compute the height of the tree at various nodes.



Computing Tree Height

- Start by assigning heights of 0 to all nodes.
- Then simply traverse all nodes recursively, setting their height to be the height of their maximum child's height

A leaf of the tree has height of zero.

All other nodes have at least a height of 1.

Add the height of the maximum child to **aNode**'s current height in the tree already.

```
computeNodeHeightsFrom(aNode) {
    mark aNode as visited

    IF aNode has just 1 edge THEN
        set aNode's height to 0
    ELSE
        set aNode's height to 1

    max = 0
    FOR each edge e of aNode DO {
        otherNode = node of edge e that is not aNode

        IF otherNode was not yet visited THEN  {
            computeNodeHeightsFrom(otherNode)
            IF height of otherNode > max THEN
                max = height of otherNode
        }
    }
    set height of aNode to its current height + max
}
```

Get all children's heights recursively and keep the maximum.

Traversal By Tree Height

- To traverse according to the branch sizes, we need to sort the neighbouring nodes by height and visit in that order:

Same code as before, but now we get the neighbours and sort them by increasing order of height so that we can visit the smaller branches first.

```
computeCoveragePathFrom(aNode, incomingEdge) {
    IF previous of aNode is not NULL THEN
       return

    add aNode's location to the path

    set previous of aNode to node at other end of incomingEdge

    neighbours = a list of aNode's neighbours
    sort neighbours by increasing order of their precomputed height

    FOR each neighbour neigh in neighbours DO {
       edge = the edge that connects aNode and neigh
       computeCoveragePathFrom(neigh, edge)
    }
    add location of aNode's previous to the path
}
```

Start the Lab ...