Graph Shortest Path

Sai-Keung Wong
National Yang Ming Chiao Tung University
Taiwan (ROC)

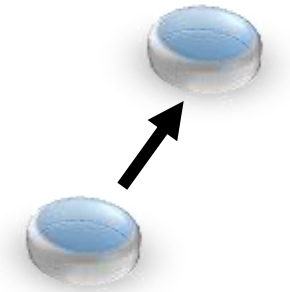
Problem

Given a graph, compute a shortest path between two nodes.
Assume that the distance between two nodes is positive.

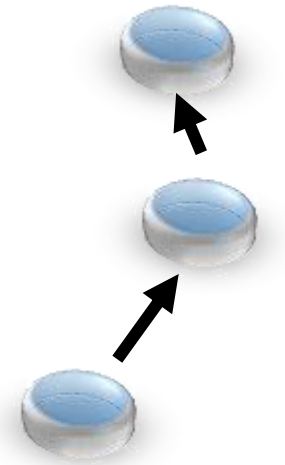
```
void findShortestPath( Node *src, Node *dest ) {
}

void findShortestPath_MainEntry(Node *src, Node *dest) {
   initializeNodeCostOfAllNodes();
   src->path_length = 0;

}
```

```
void findShortestPath_MainEntry(Node *src, Node *dest) {
  initializeNodeCostOfAllNodes();
  // use cost instead of path_length. More general
  src->cost = 0;
  vector<Node*> active;
  active->push_back(src);
  findShortestPath(src, dest, active);
}
```

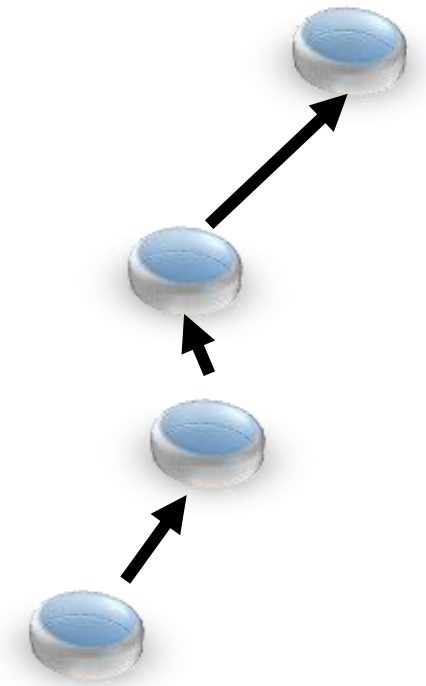
```
void initializeNodeCostOfAllNodes() {
        for (each node) {
           node->cost  = infinity;                    // notation
            node->path_parent = nullptr;
        }
}

void findShortestPath( Node *src, Node *dest) {
}

void findShortestPath_MainEntry(Node *src, Node *dest) {
  initializeNodeCostOfAllNodes();
  src->cost = 0;
  findShortestPath(src, dest);
}
```



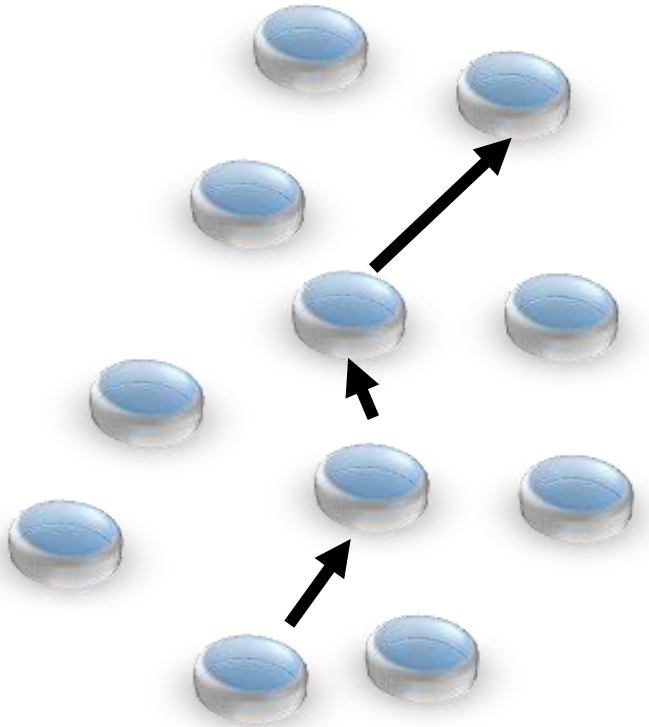
```
void initializeNodeCostOfAllNodes() {
        for (each node) {
           node->cost  = infinity;                    // notation
            node->path_parent = nullptr;
        }
}

void findShortestPath( Node *src, Node *dest) {
}

void findShortestPath_MainEntry(Node *src, Node *dest) {
  initializeNodeCostOfAllNodes();
  src->cost = 0;
  findShortestPath(src, dest);
}
```



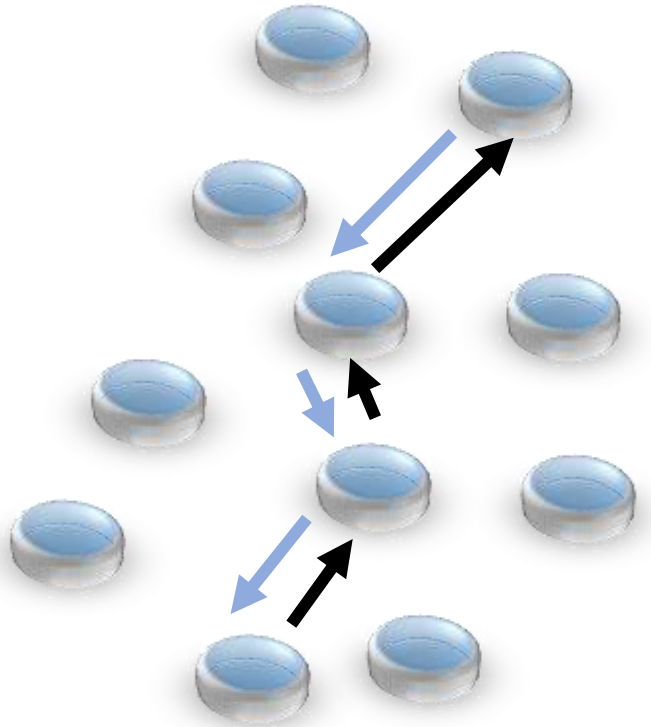
```
void initializeNodeCostOfAllNodes() {
        for (each node) {
            node->cost  = infinity;                    // notation
             node->path_parent = nullptr;
        }
}

void findShortestPath( Node *src, Node *dest) {
}

void findShortestPath_MainEntry(Node *src, Node *dest) {
  initializeNodeCostOfAllNodes();
  src->cost = 0;
  findShortestPath(src, dest);
}
```



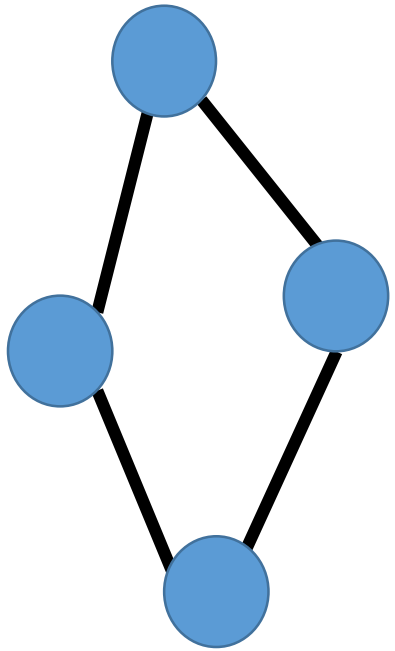
```
void initializeNodeCostOfAllNodes() {
        for (each node) {
            node->cost  = infinity;                    // notation
             node->path_parent = nullptr;
        }
}

void findShortestPath( Node *src, Node *dest) {
}

void findShortestPath_MainEntry(Node *src, Node *dest) {
  initializeNodeCostOfAllNodes();
  src->cost = 0;
  findShortestPath(src, dest);
}
```



```
void initializeNodeCostOfAllNodes() {
        for (each node) {
            node->cost  = infinity;                    // notation
             node->path_parent = nullptr;
        }
}

void findShortestPath( Node *src, Node *dest) {
}

void findShortestPath_MainEntry(Node *src, Node *dest) {
  initializeNodeCostOfAllNodes();
  src->cost = 0;
  findShortestPath(src, dest);
}
```

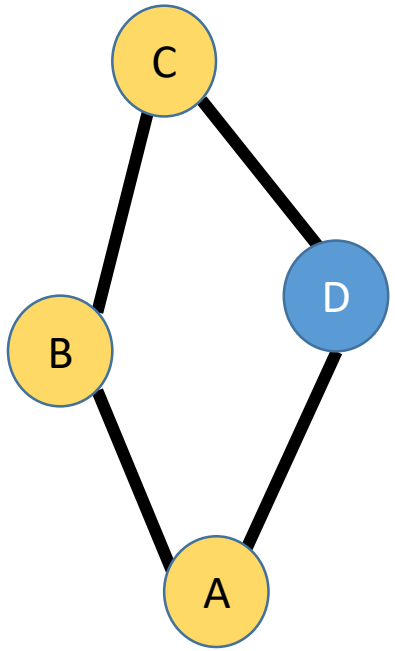


```
void findShortestPath( Node *src, Node *dest) {
  for (each adjacent node of src) {
     cost = length(src, node)
     if (node->cost > cost + src->cost) {
          node->cost = cost + src->cost;
          node->path_parent = src;
          …….
     }
  }
}
```



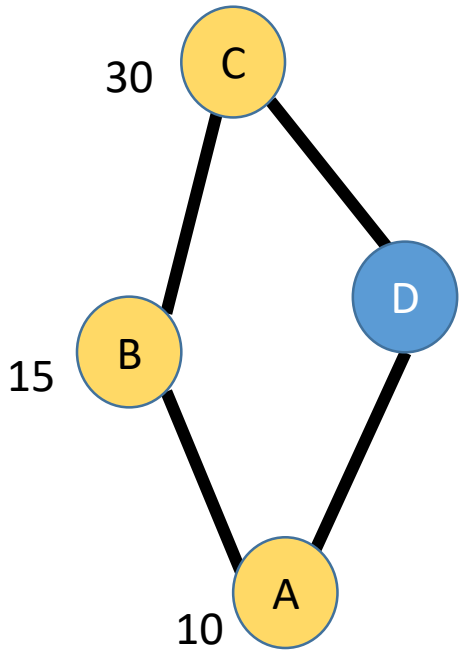
The cost is positive if we travel from one node to an adjacent node.

```
void findShortestPath( Node *src, Node *dest) {
  for (each adjacent node of src) {
     cost = length(src, node)
     if (node->cost > cost + src->cost) {
          node->cost = cost + src->cost;
          node->path_parent = src;
          ……..
     }
  }
}
```



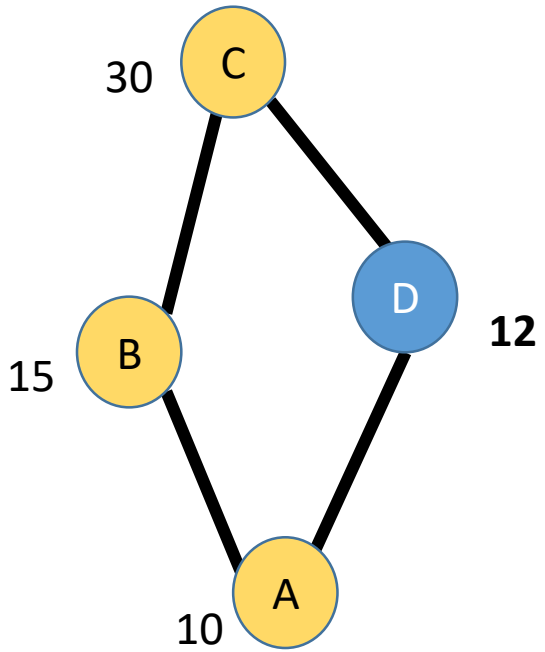
The cost is positive if we travel from one node to an adjacent node.

```
void findShortestPath( Node *src, Node *dest) {
  for (each adjacent node of src) {
     cost = length(src, node)
     if (node->cost > cost + src->cost) {
          node->cost = cost + src->cost;
          node->path_parent = src;
          …….
     }
  }
}
```



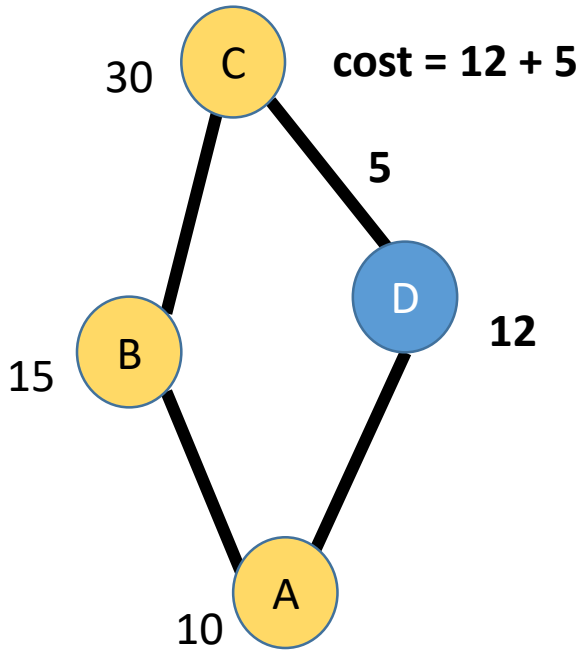
The cost is positive if we travel from one node to an adjacent node.

```
void findShortestPath( Node *src, Node *dest) {
  for (each adjacent node of src) {
     cost = length(src, node)
     if (node->cost > cost + src->cost) {
          node->cost = cost + src->cost;
          node->path_parent = src;
          …….
     }
  }
}
```



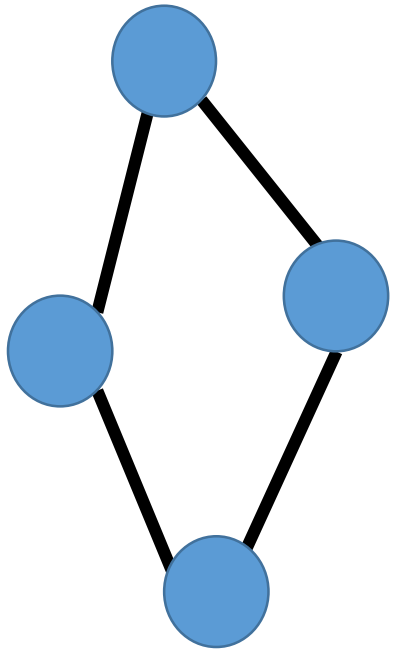
The cost is positive if we travel from one node to an adjacent node.

```
void findShortestPath( Node *src, Node *dest) {
  for (each adjacent node of src) {
     cost = length(src, node)
     if (node->cost > cost + src->cost) {
          node->cost = cost + src->cost;
          node->path_parent = src;
          …….
     }
  }
}
```

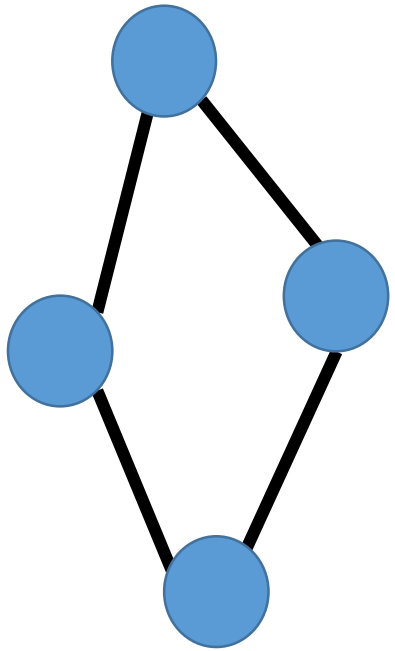


The cost is positive if we travel from one node to an adjacent node.

```
void findShortestPath( Node *dest, vector<*Node> &active ) {
  while (!active.empty()) {
    Node *src = active.back();
    active.pop_back();
    for (each adjacent node of src) {
      cost = length(src, node)
      if (node->cost > cost + src->cost) {
        node->cost = cost + src->cost;
        node->path_parent = src;
       active.push_back( node );
      } // if
    } // for
  } // while
} // func
```



```
void findShortestPath( Node *dest, vector<*Node> &active ) {
  while (!active.empty()) {
    Node *src = active.back();
    active.pop_back();
    for (each adjacent node of src) {
      cost = length(src, node)
      if (node->cost <= cost + src->cost) continue;
      node->cost = cost + src->cost;
      node->path_parent = src;
      active.push_back( node );

    } // for
  } // while
} // func
```



```
void findShortestPath( Node *dest, vector<*Node> &active ) {
  while (!active.empty()) {
    Node *src = active.back();
    active.pop_back();
    for (each adjacent node of src) {
      cost = length(src, node)
      if (node->cost <= cost + src->cost) continue;
      node->cost = cost + src->cost;
      node->path_parent = src;
      active.push_back( node );

    } // for
  } // while
} // func
```

```
void findShortestPath( Node *dest, vector<*Node> &active ) {
  while (!active.empty()) {
    Node *src = active.back();
    active.pop_back();
    for (each adjacent node of src) {
      cost = length(src, node)
      if (node->cost > cost + src->cost) {
        node->cost = cost + src->cost;
        node->path_parent = src;
          active.push_back( node );
      } // if
    } // for
  } // while
} // func
```

```
void findShortestPath( Node *dest, vector<*Node> &active ) {
  while (!active.empty()) {
    Node *src = active.back();
    active.pop_back();
    for (each adjacent node of src) {
      cost = length(src, node)
      if (node->cost <= cost + src->cost) continue;
      node->cost = cost + src->cost;
      node->path_parent = src;
      active.push_back( node );

    } // for
  } // while
} // func
```

```
void findShortestPath( Node *dest, vector<*Node> &active ) {
  while (!active.empty()) {
    Node *src = active.back();
    active.pop_back();
    for (each adjacent node of src) {
      cost = length(src, node)
      if (node->cost > cost + src->cost) {
        node->cost = cost + src->cost;
        node->path_parent = src;
          active.push_back( node );
      } // if
    } // for
  } // while
} // func
```

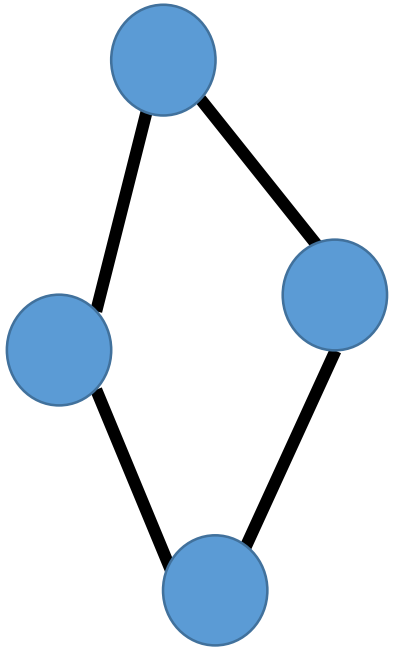
Main algorithm

```
void findShortestPath_MainEntry(Node *src, Node *dest) {
  initializeNodeCostOfAllNodes();
  src->cost = 0;
  vector<Node*> active;
  active->push_back(src);
  // src is placed in active. Don't need it explicitly in the function call.
  findShortestPath(dest, active);
}
```

Iterative approach

```
void findShortestPath( Node *dest, vector<*Node> &active ) {
  while (!active.empty()) {
    Node *src = active.back();
    active.pop_back();
    for (each adjacent node of src) {
      cost = length(src, node)
      if (node->cost <= cost + src->cost) continue;
      node->cost = cost + src->cost;
      node->path_parent = src;
      active.push_back( node );
    }
  }
}
```

```
Node {
  Node *path_parent;
  cost;
}
```



Iterative approach

```
void findShortestPath( Node *dest, vector<*Node> &active ) {
  while (!active.empty()) {
    Node *src = active.back();
    active.pop_back();
    for (each adjacent node of src) {
      cost = length(src, node)
      if (node->cost <= cost + src->cost) continue;
      node->cost = cost + src->cost;
      node->path_parent = src;
      active.push_back( node );
    }
  }
}
```

```
Node {
  Node *path_parent;
  cost;
}
```

Recursive approach

```
void findShortestPath( Node *src ) {
    if (!src) return;
    for (each adjacent node of src) {
        cost = length(src, node)
        if (node->cost <= cost + src->cost) continue;
        node->cost = cost + src->cost;
        node->path_parent = src;
        findShortestPath( node );
    }
}
```

```
Node {
  Node *path_parent;
  Real cost;
}
```