Paths in Graphs: Breadth-First Search

Michael Levin

Department of Computer Science and Engineering
University of California, San Diego

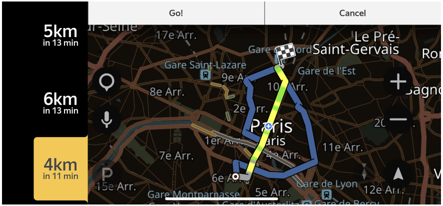
Graph Algorithms
Algorithms and Data Structures

Outline

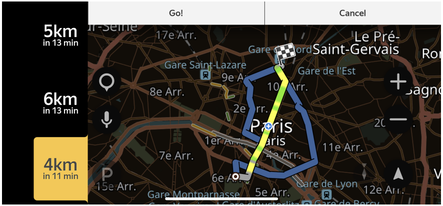
Applications

Applications

Applications



Applications

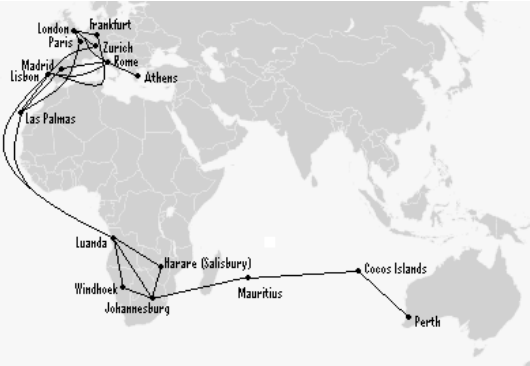


Minimize Transfers

What is the minimum number of transfers to get from London to Perth?

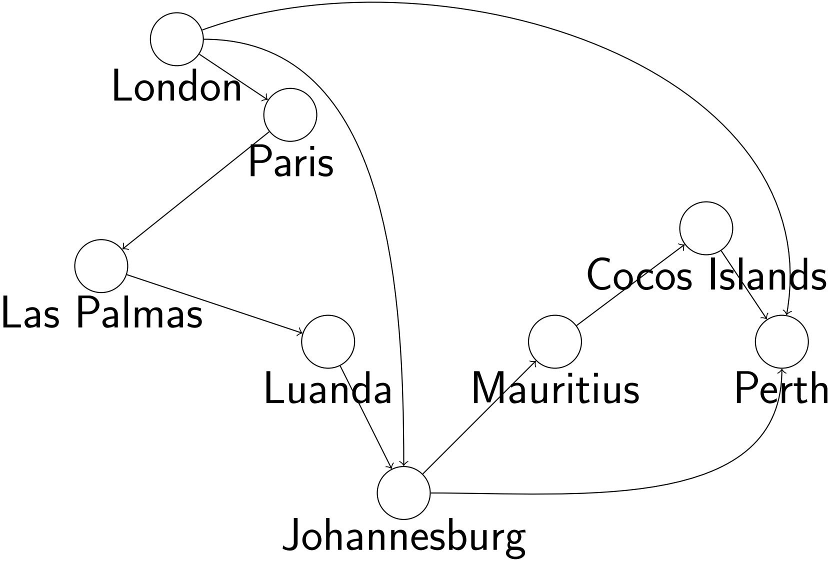
Minimize Transfers

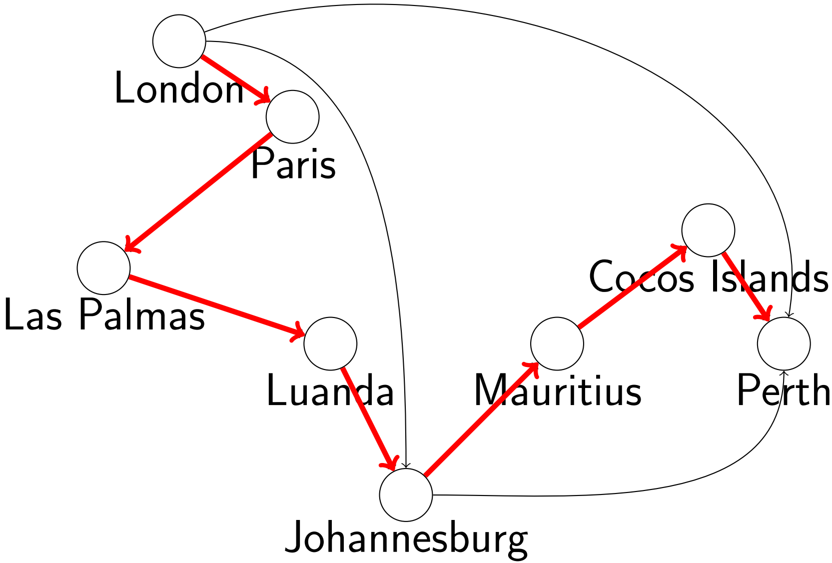
What is the minimum number of transfers to get from London to Perth?

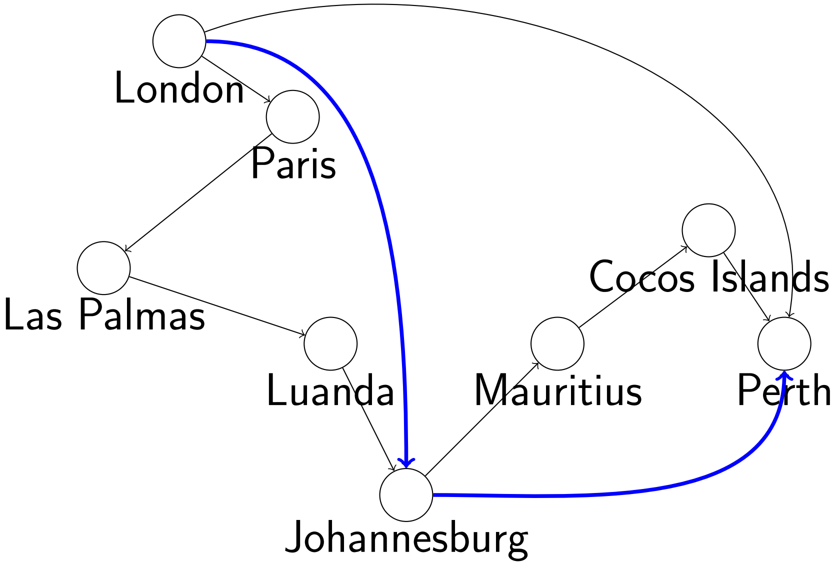


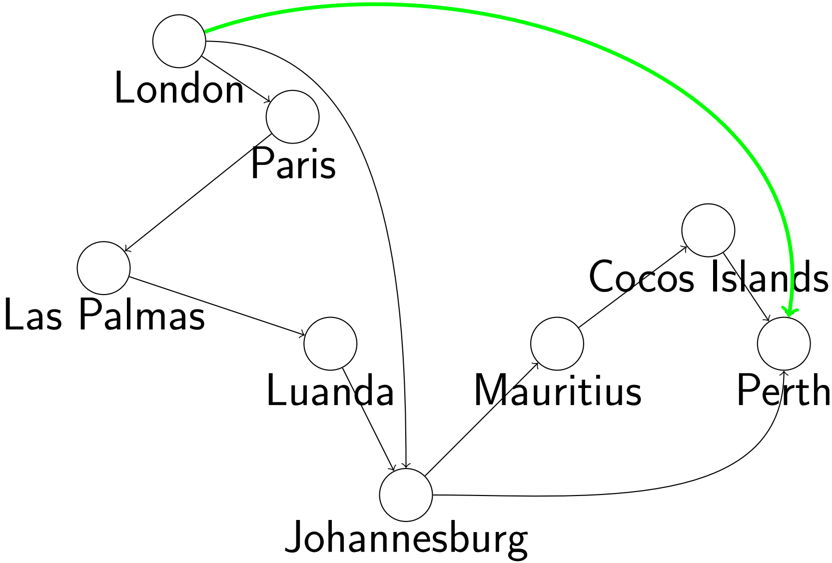
Minimize Transfers

What is the minimum number of transfers to get from London to Perth?


London
Paris
Las Palmas
Luanda
Johannesburg
Mauritius
Cocos Islands
Perth


London
Paris
Las Palmas
Luanda
Johannesburg
Mauritius
Cocos Islands
Perth

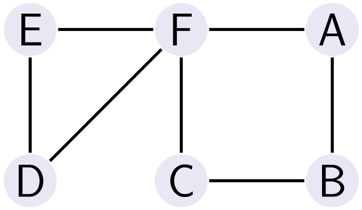

London
Paris
Las Palmas
Luanda
Johannesburg
Mauritius
Cocos Islands
Perth


London
Paris
Las Palmas
Luanda
Johannesburg
Mauritius
Cocos Islands
Perth

Outline

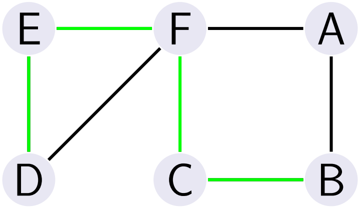
Path Length

Path length $L(P)$ is the number of edges in the path.



Path Length

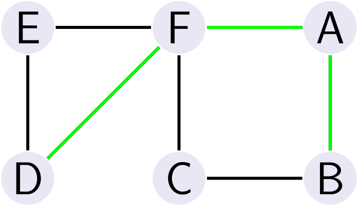
Path length $L(P)$ is the number of edges in the path.



$$L(B - C - F - E - D) = 4$$

Path Length

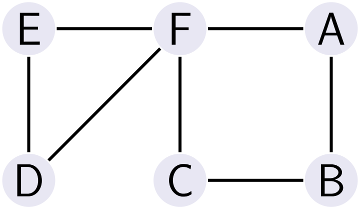
Path length $L(P)$ is the number of edges in the path.



$L(B - C - F - E - D) = 4$
$L(B - A - F - D) = 3$

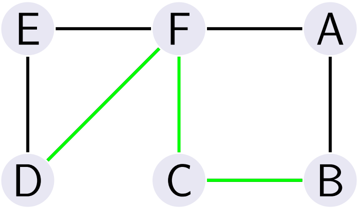
Distance

The distance between two nodes is the length of the shortest path between them.



Distance

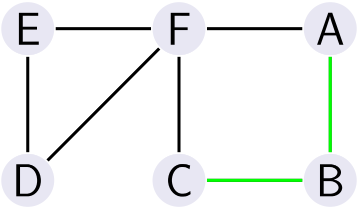
The distance between two nodes is the length of the shortest path between them.



$d(B, D) = 3$

Distance

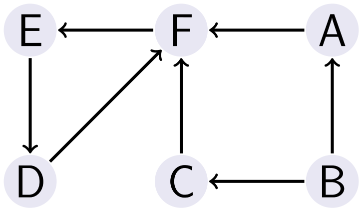
The distance between two nodes is the length of the shortest path between them.



$d(B, D) = 3$
$d(A, C) = 2$

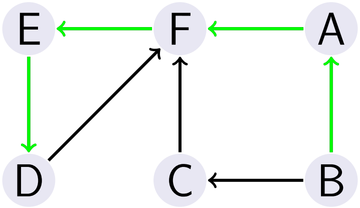
Distance

The distance between two vertices is the length of the shortest path between them.



Distance

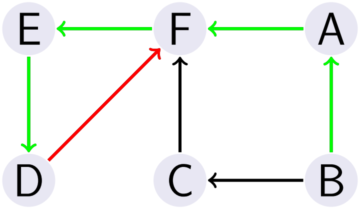
The distance between two vertices is the length of the shortest path between them.



$d(B, D) = 4$

Distance

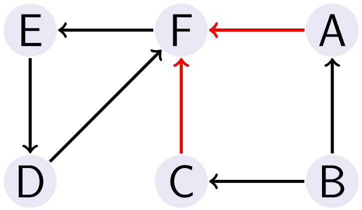
The distance between two vertices is the length of the shortest path between them.



$d(B, D) = 4$

Distance

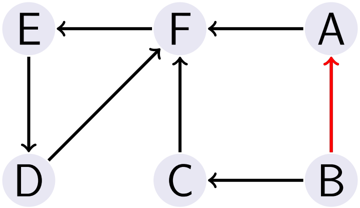
The distance between two vertices is the length of the shortest path between them.



$d(B, D) = 4$
$d(A, C) = \infty$

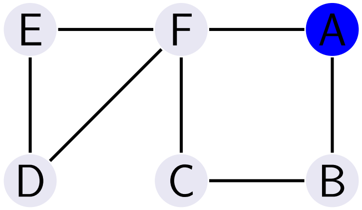
Distance

The distance between two vertices is the length of the shortest path between them.

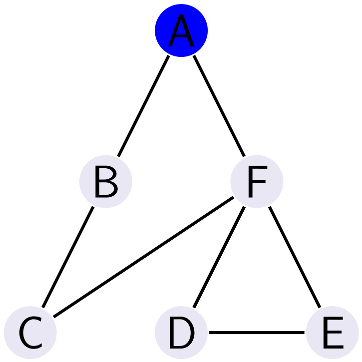


$d(B, D) = 4$
$d(A, C) = \infty$

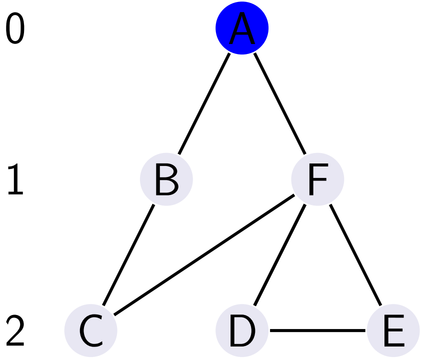
Distance



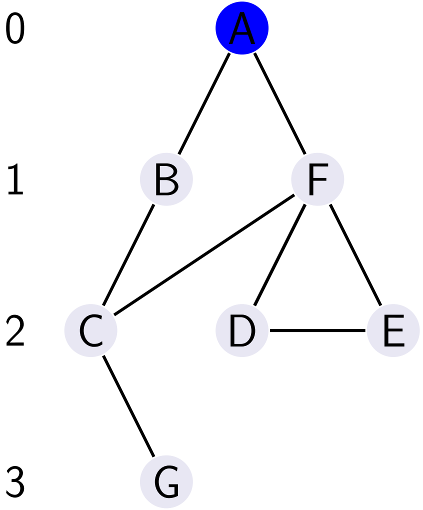
Distance layers



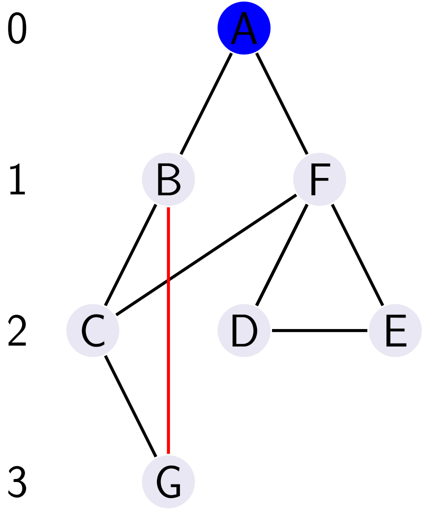
Distance layers



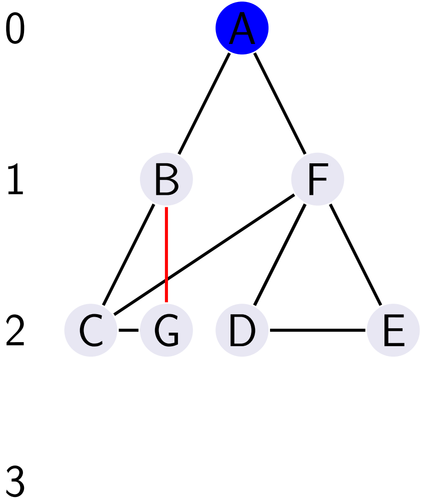
Distance layers



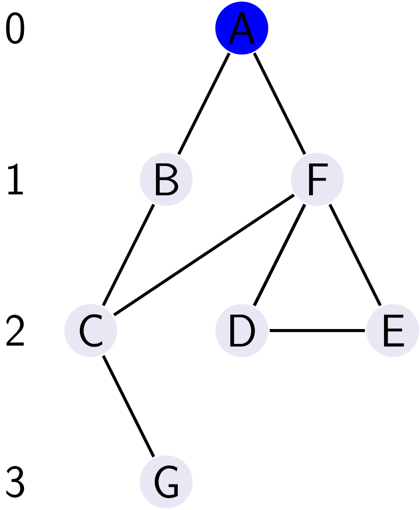
Distance layers



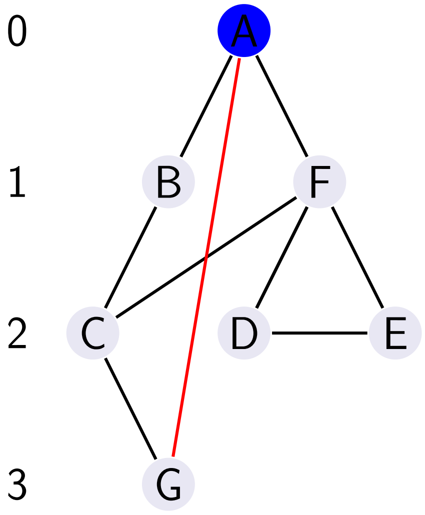
Distance layers



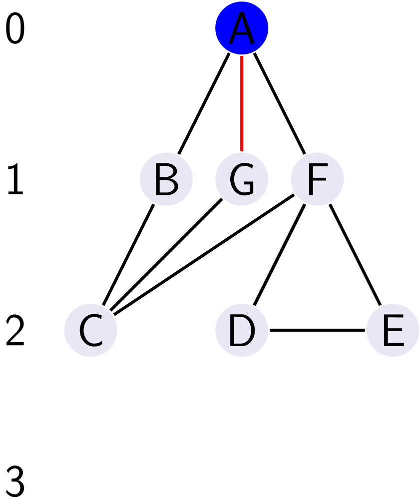
Distance layers



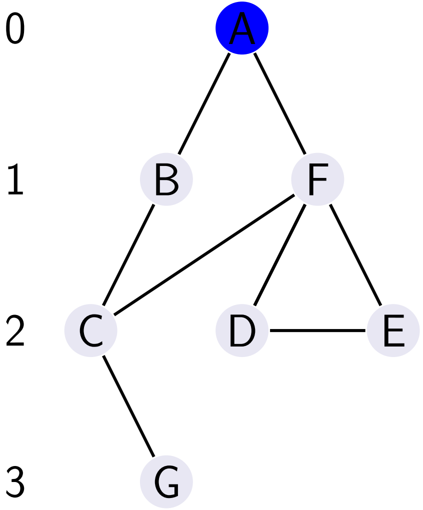
Distance layers



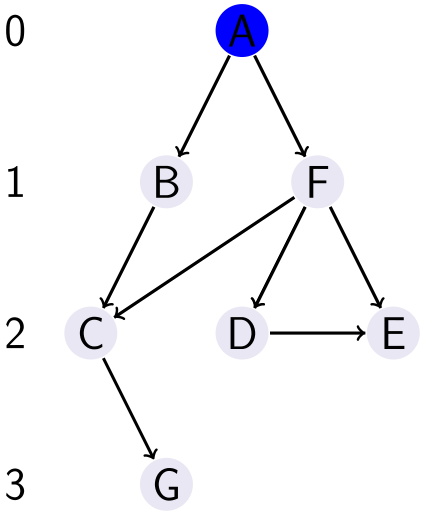
Distance layers



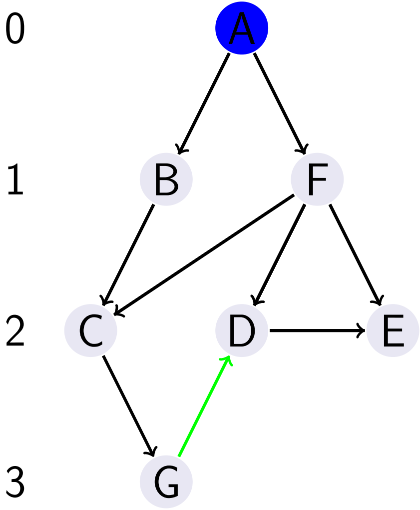
Distance layers



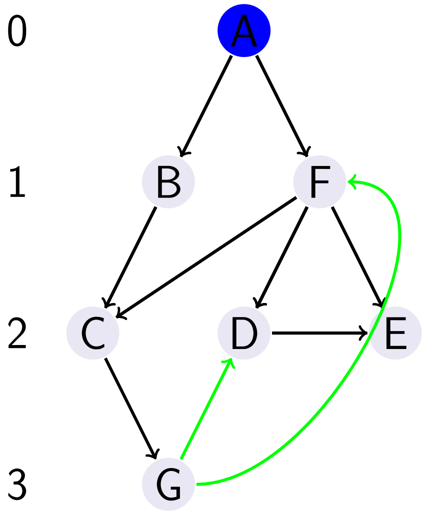
Distance layers



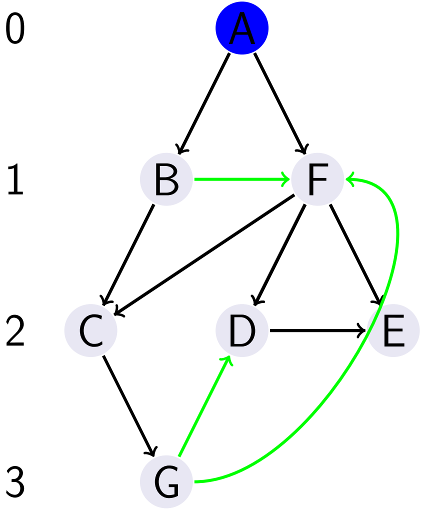
Distance layers



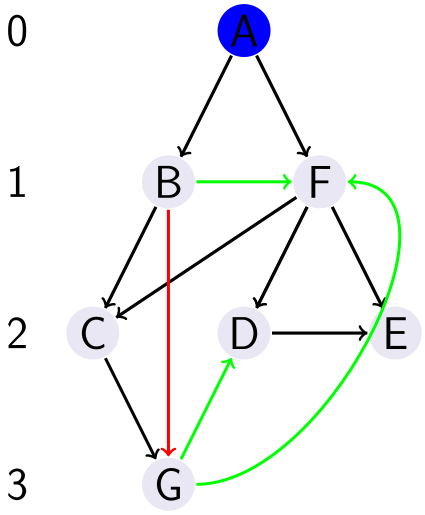
Distance layers



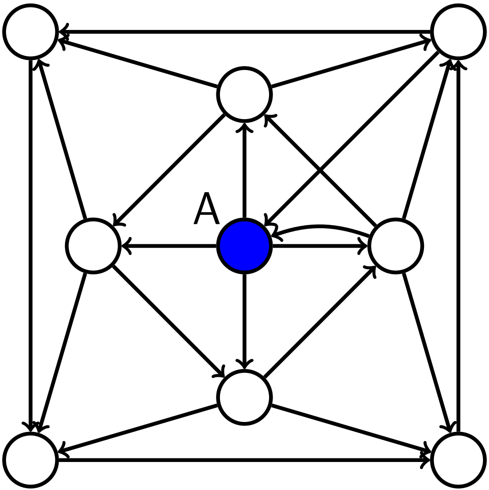
Distance layers

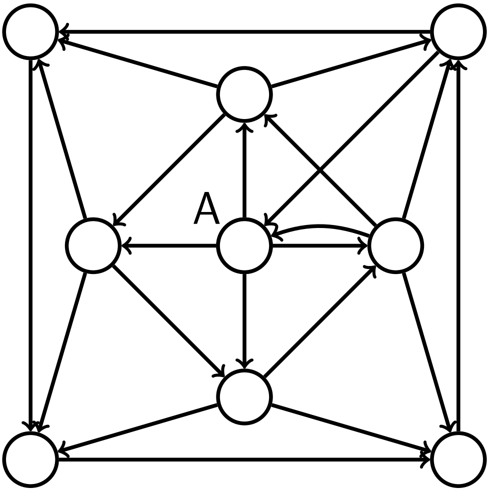


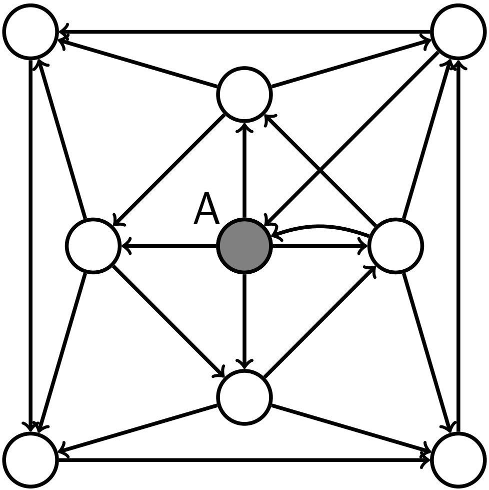
Distance layers

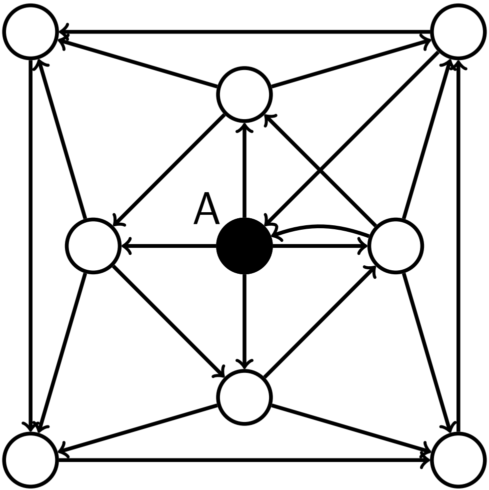


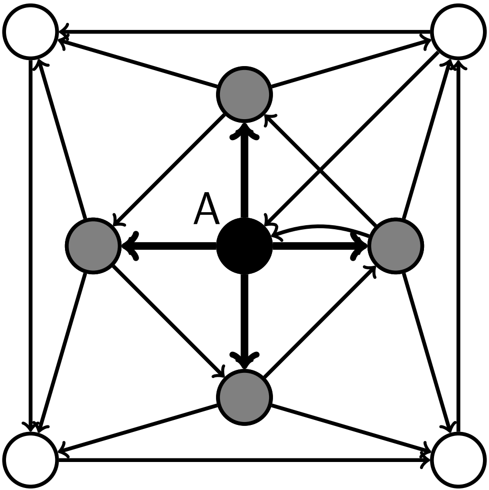
Outline

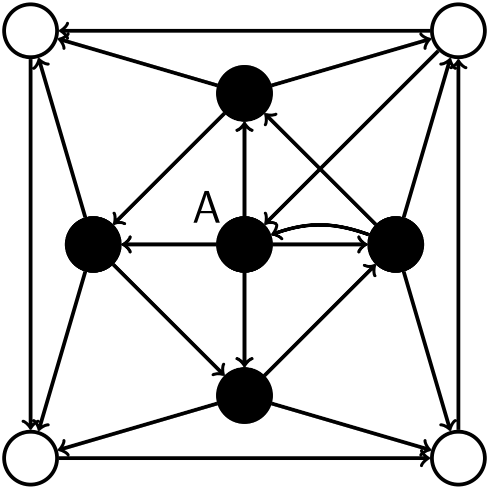

A

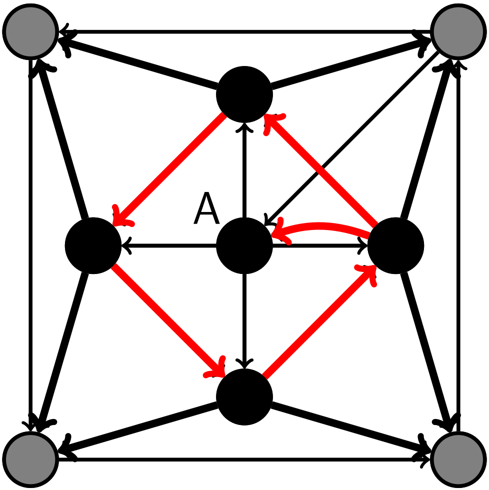

A


A

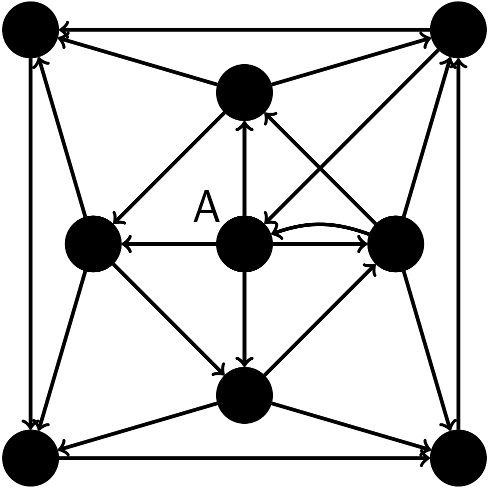

A

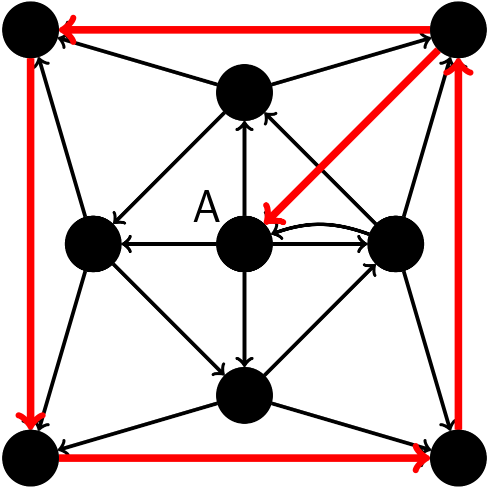

A

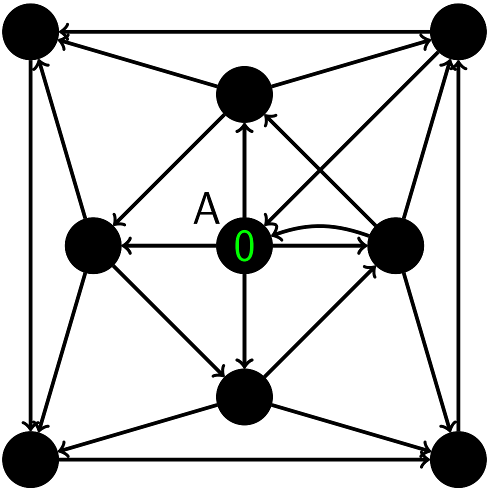

A

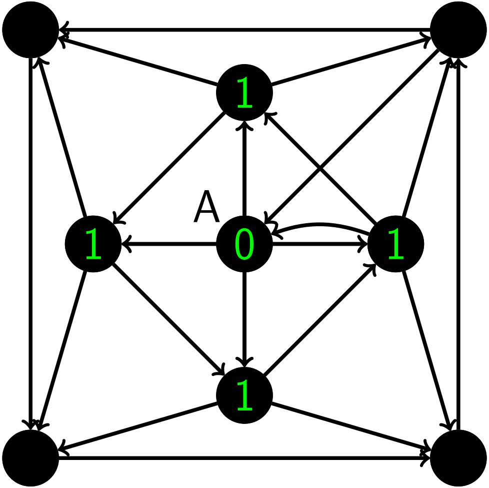


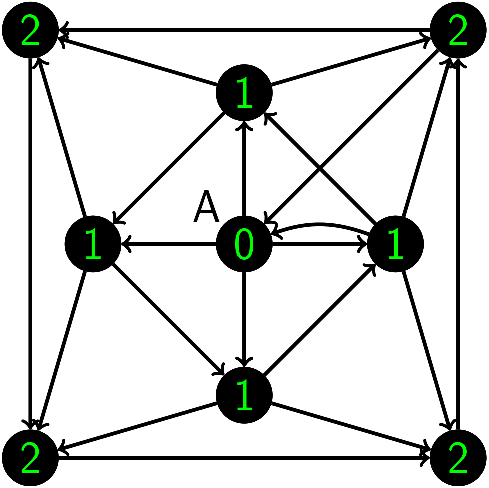
A

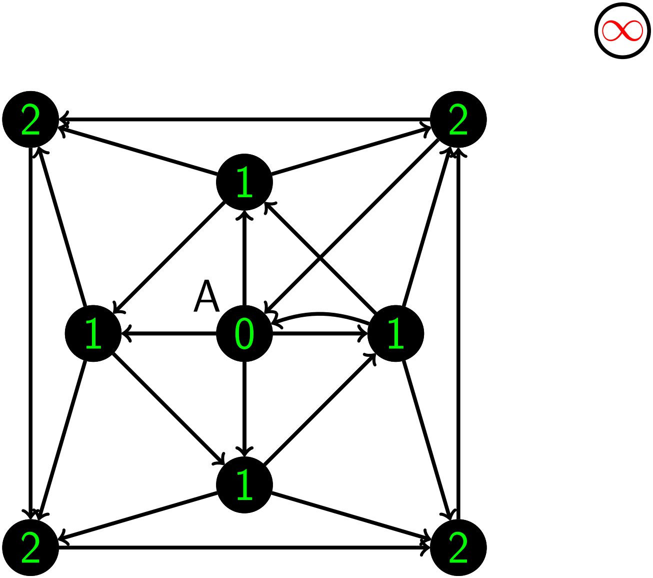

A


A

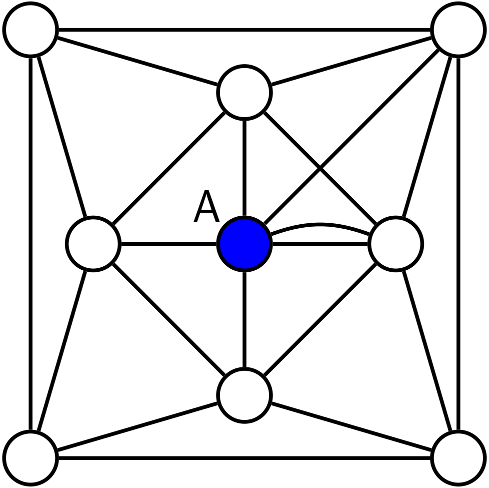

A
0

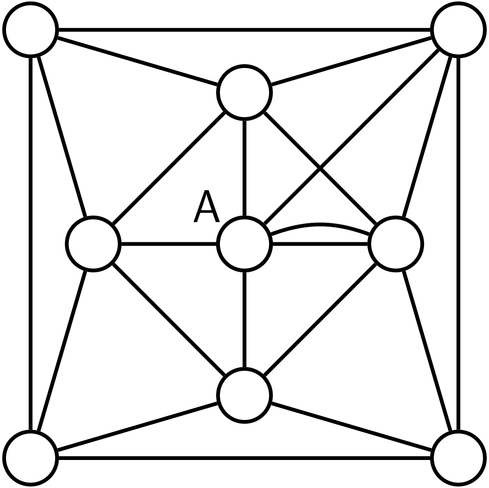

1
A
1
0
1
1

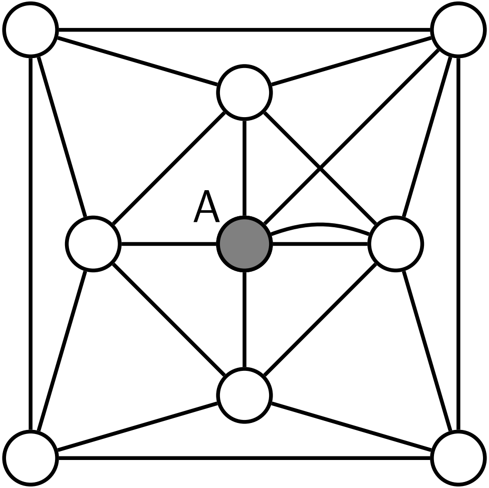

2
2
1
A
1
0
1
1
2
2

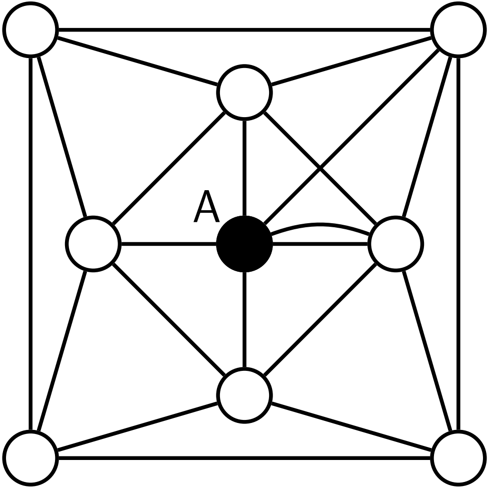


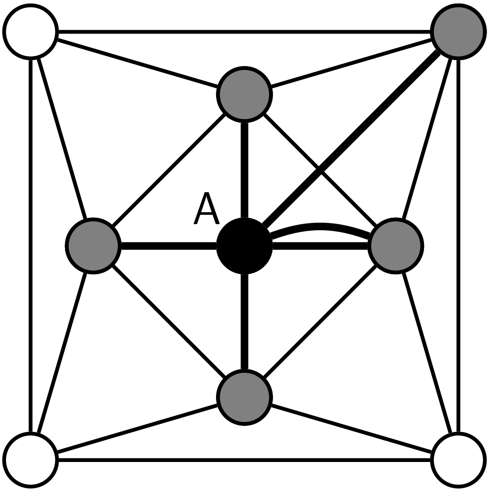
∞
2
2
1
A
1
0
1
1
2
2

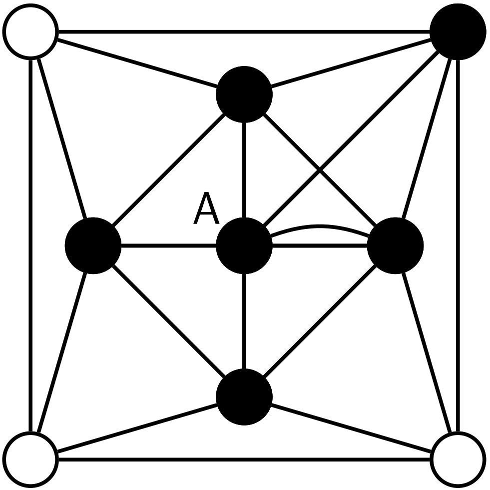

A

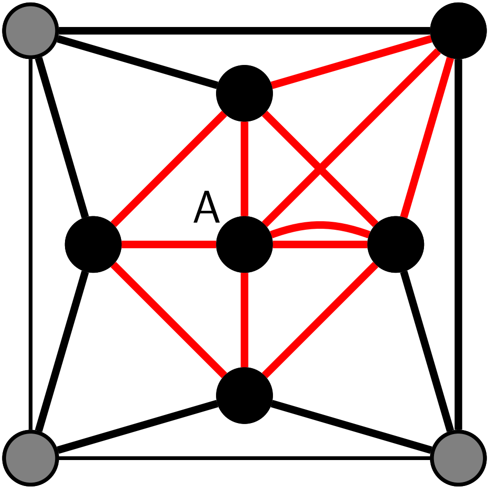

A

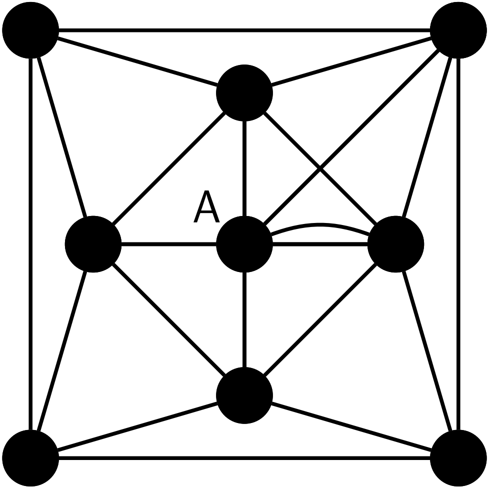

A

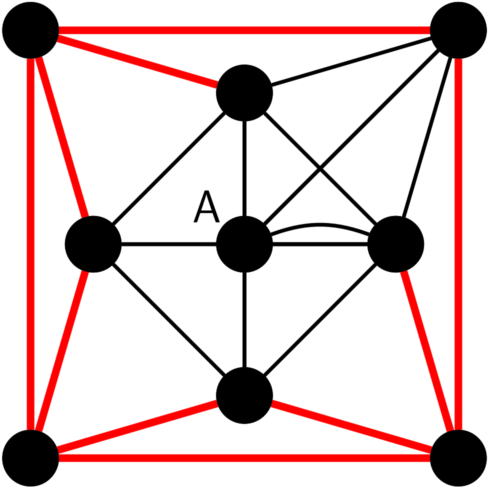

A

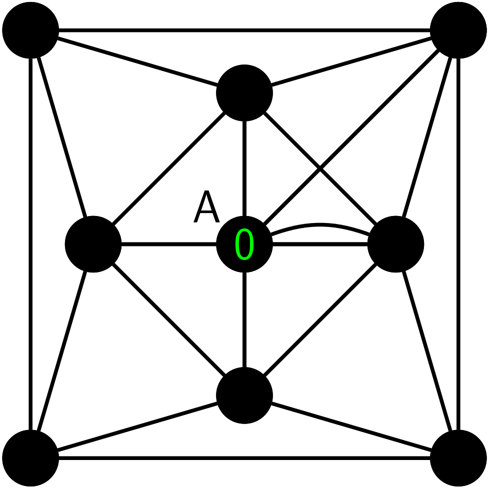

A

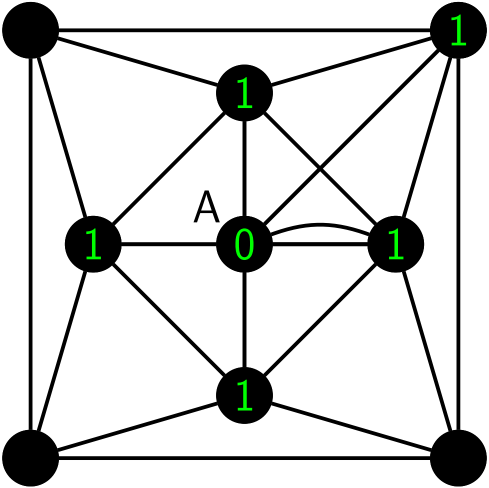

A

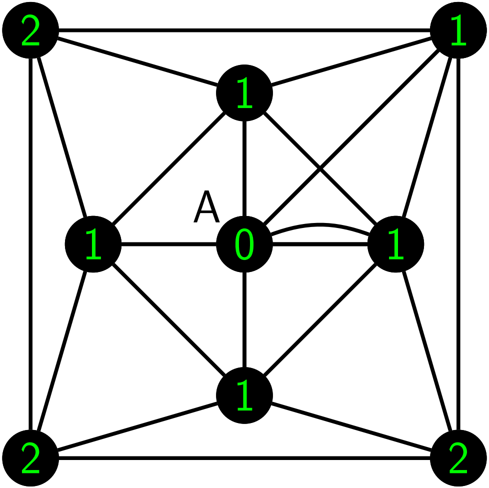

A

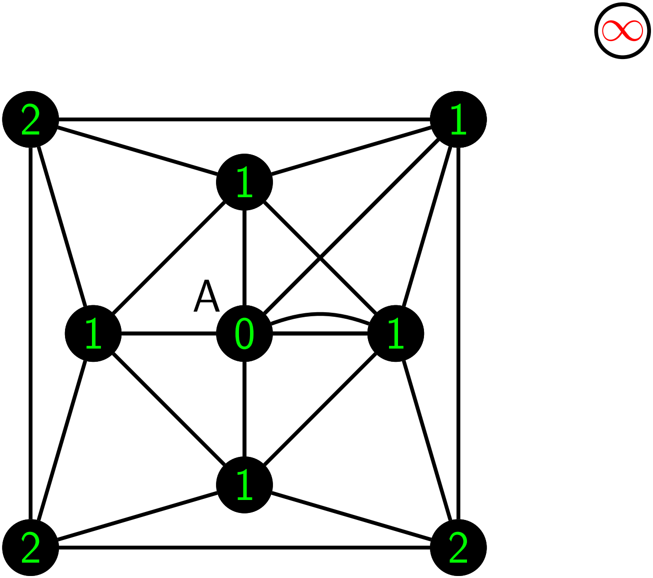

A

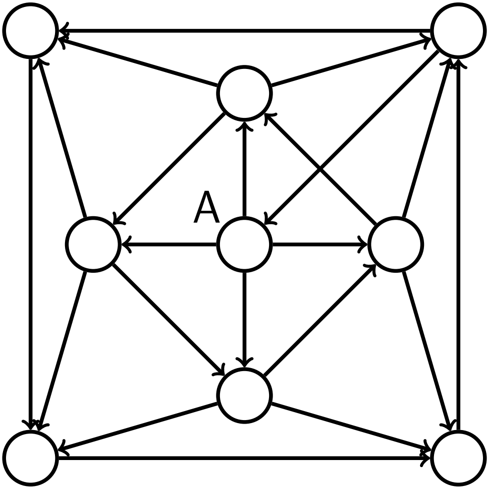

A

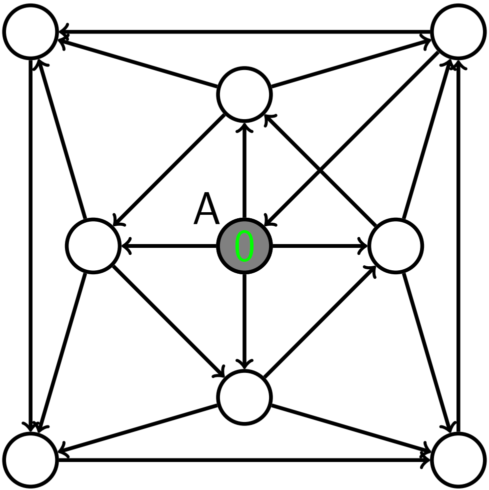

A
0

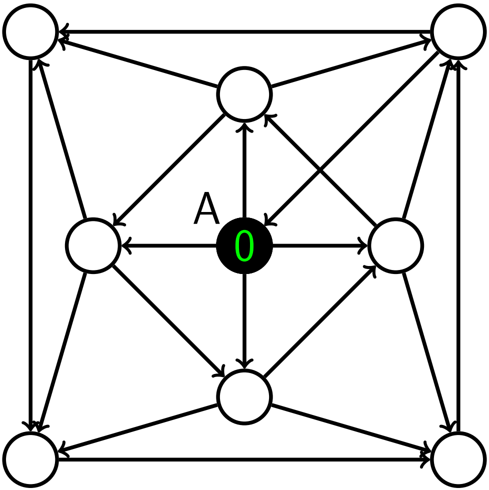

1
1
A
1
0
1
1

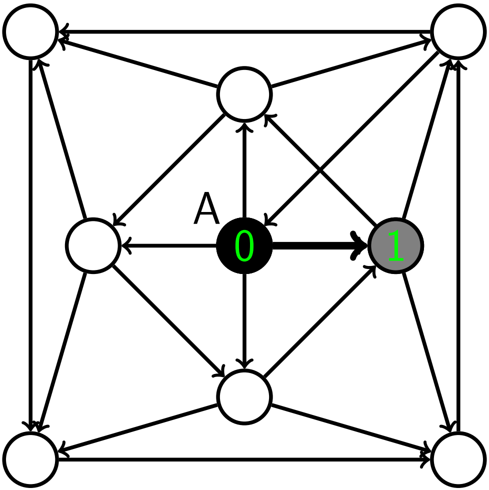

2
1
1
A
1
0
1
1
2
2

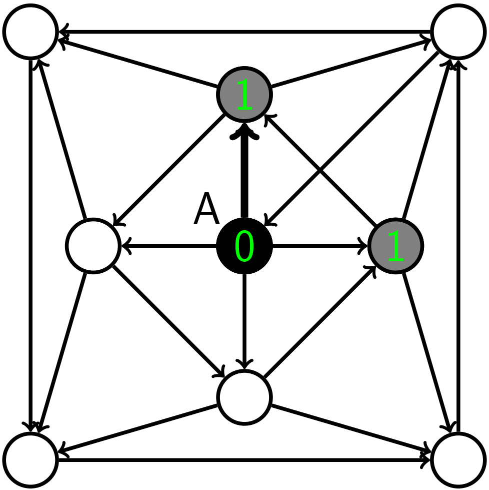

∞
2
1
1
A
1
0
1
1
2
2


A

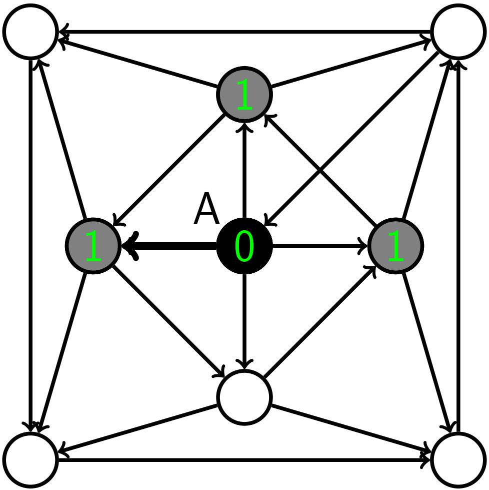

A
0

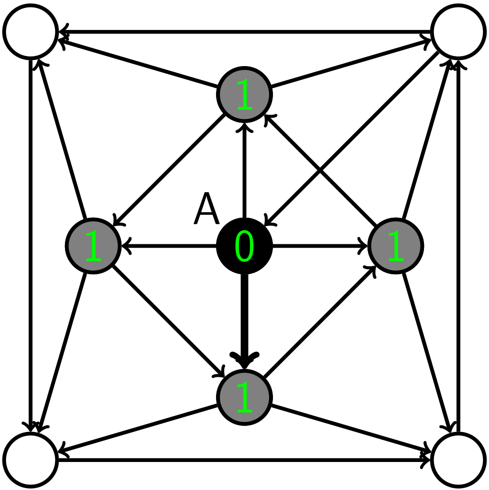

A
0

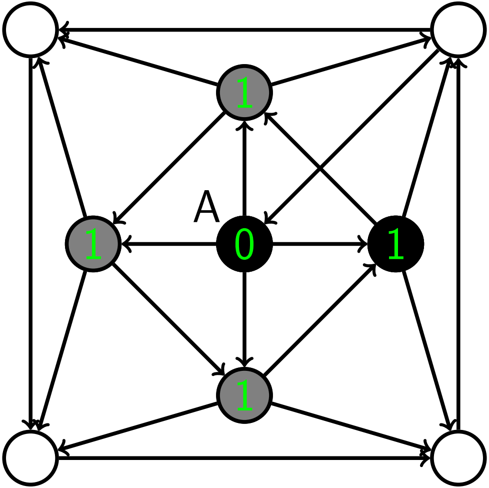

A
0
1



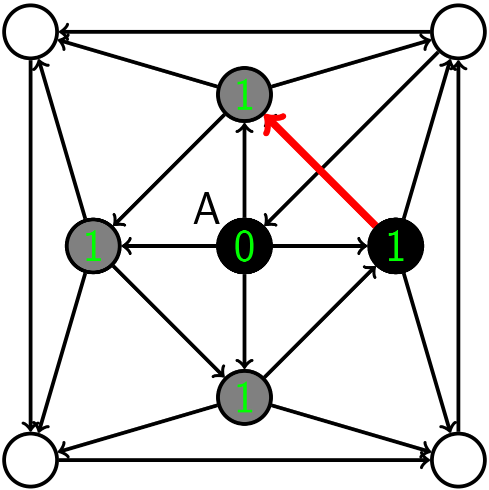
1
A
0
1


1
A
1
0
1

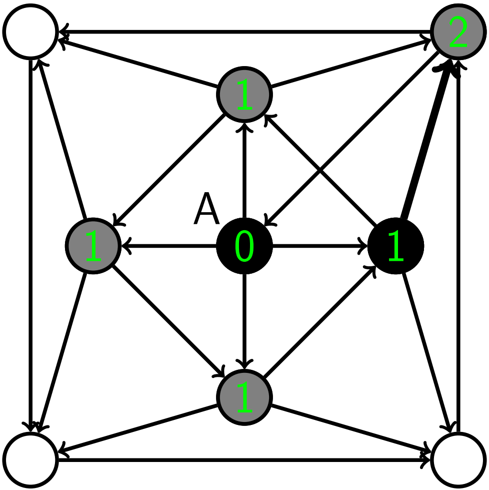

A
0
1
1
1
1



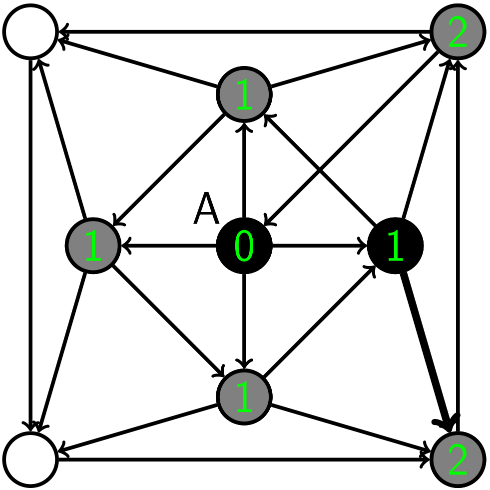
1
A
1
0
1
1



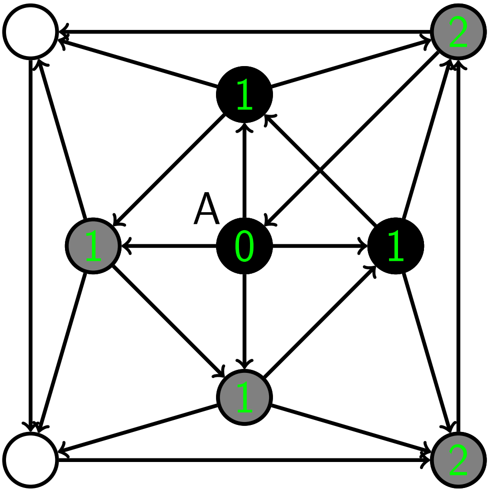
A
0
1
1
1
1

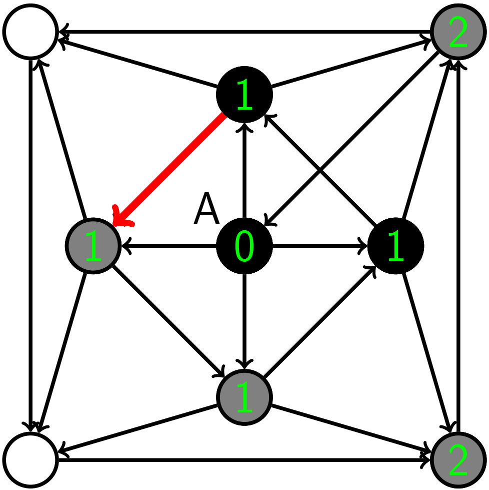


A
0
1
1
1
1
2

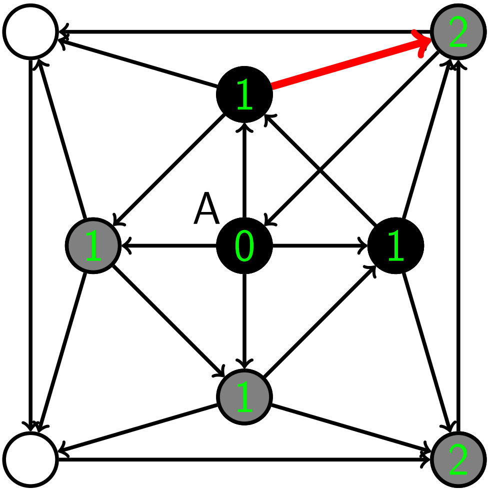


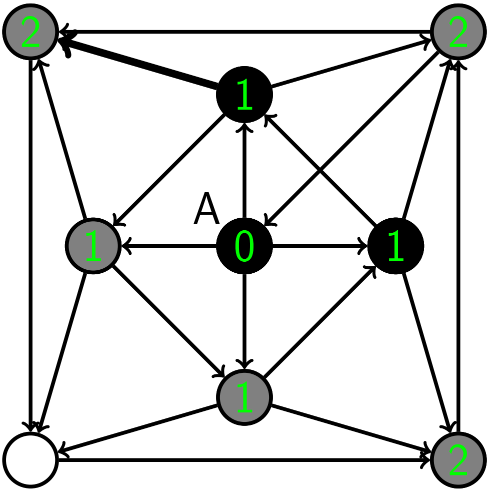
2
1
A
1
0
1
1
2

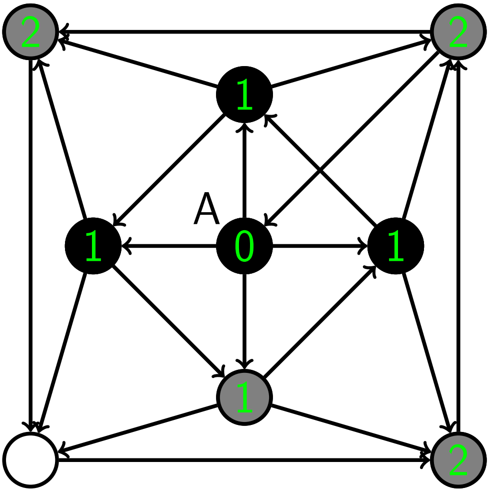

2
1
A
1
0
1
1
2

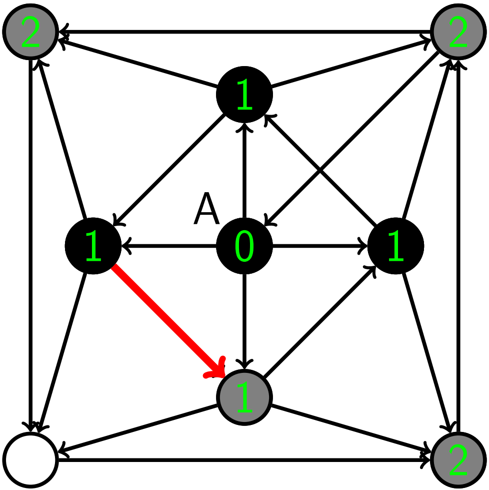


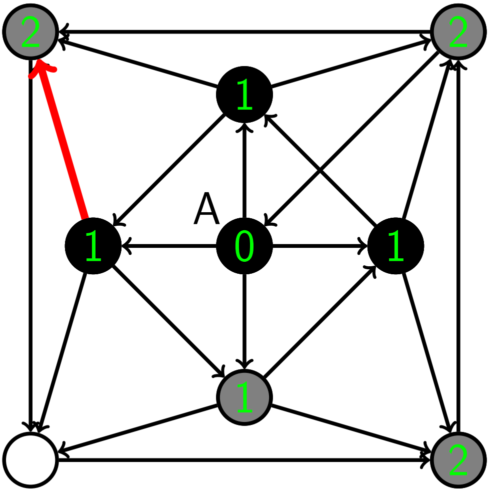
2
1
A
1
0
1
1
2


2
1
A
1
0
1
1
2

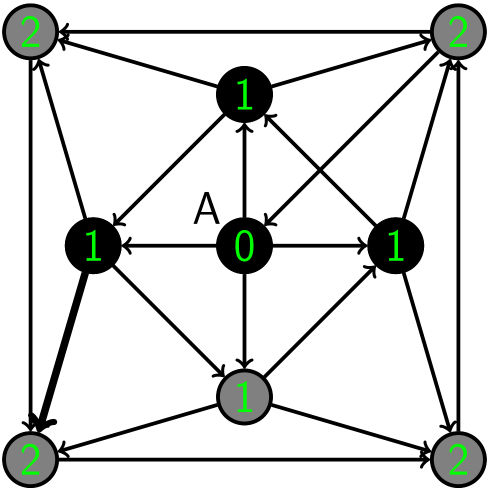

2
2
1
A
1
0
1
1
2

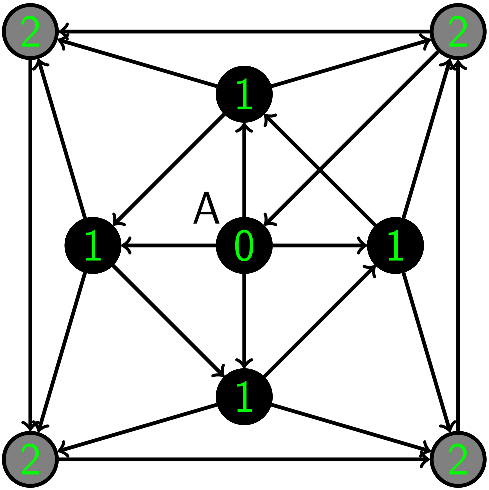

2
2
1
A
1
0
1
1
2

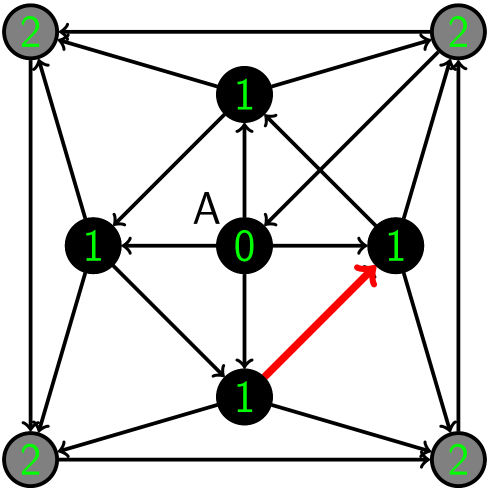

2
2
1
A
1
0
1
1
2

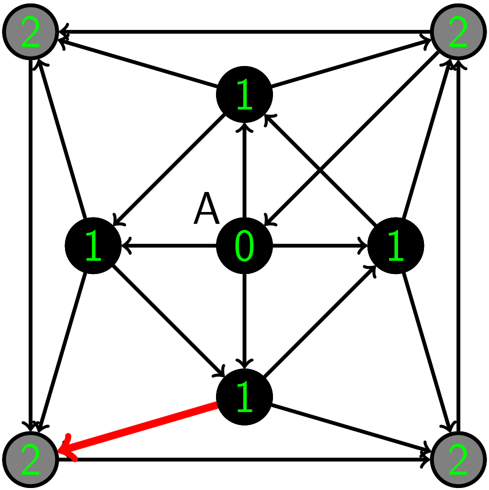


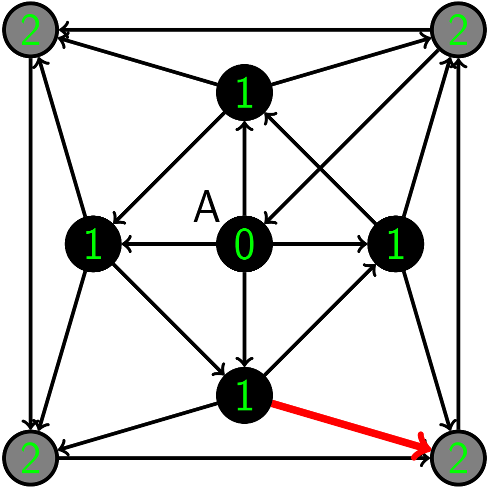
2
2
1
A
1
0
1
1
2


A
0
1
1
1
1
2
2
2
2

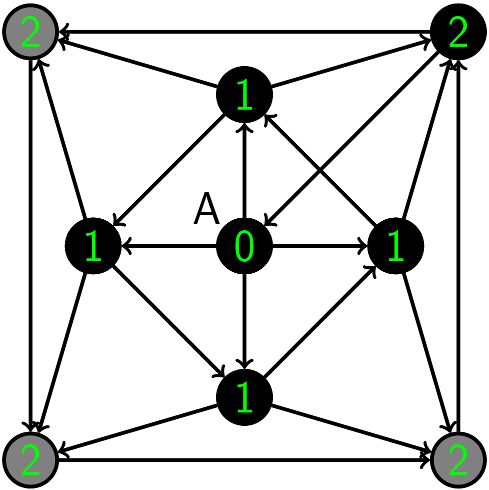

2
2
1
A
1
0
1
1
2
2

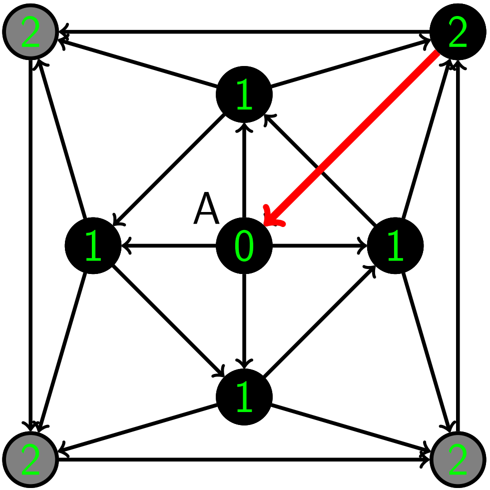

A
0
1
1
1
1
2
2
2
2


2
2
1
A
1
0
1
1
2
2

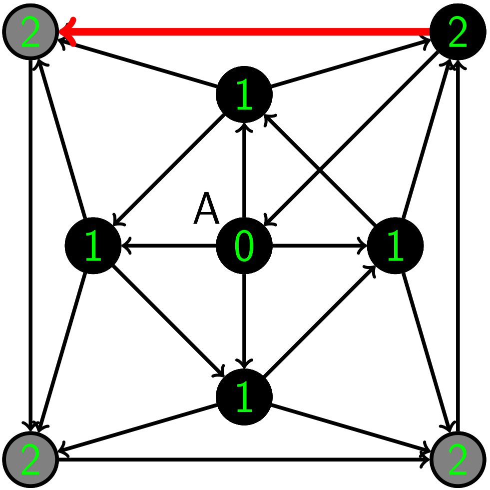


2
2
1
A
1
0
1
1
2
2

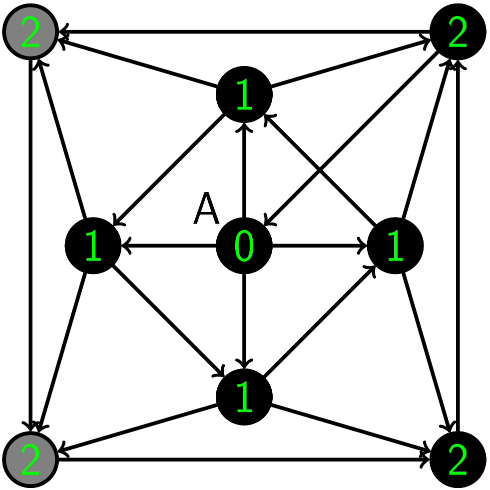

2
2
1
A
1
0
1
1
2
2



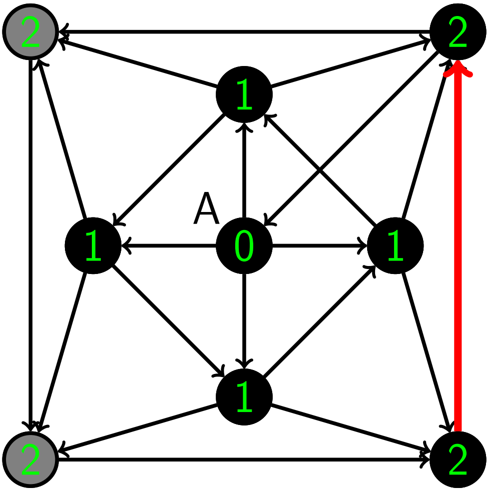
2
2
1
A
1
0
1
1
2
2

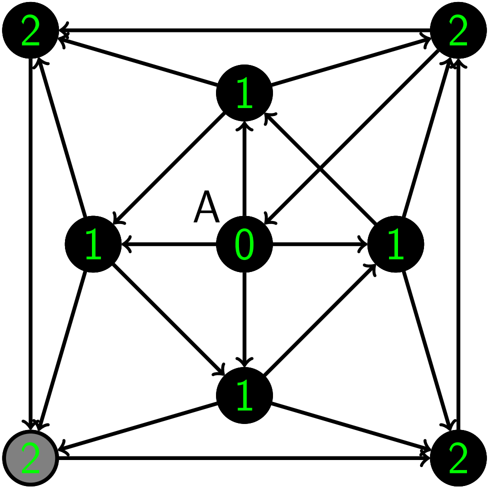


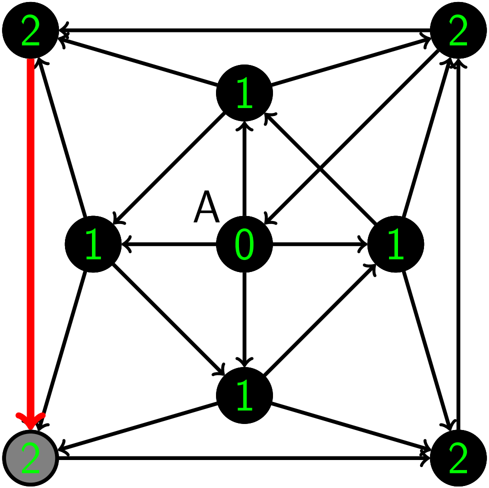
2
2
1
A
1
0
1
1
2
2



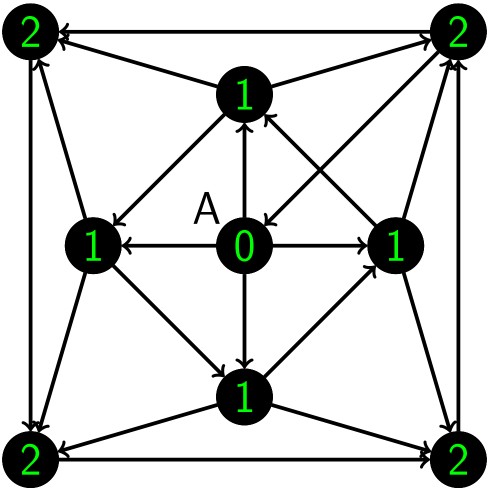
2
2
1
A
1
0
1
1
2
2

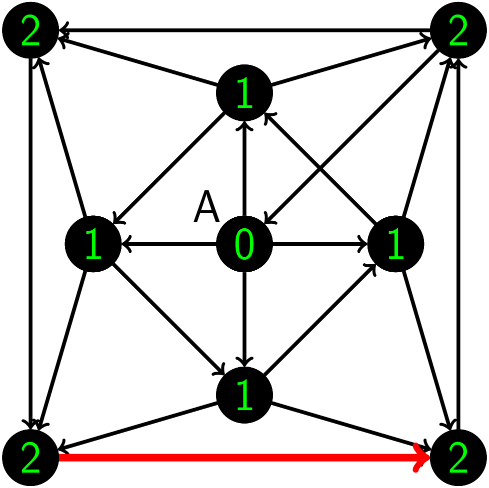

2
2
1
A
1
0
1
1
2
2

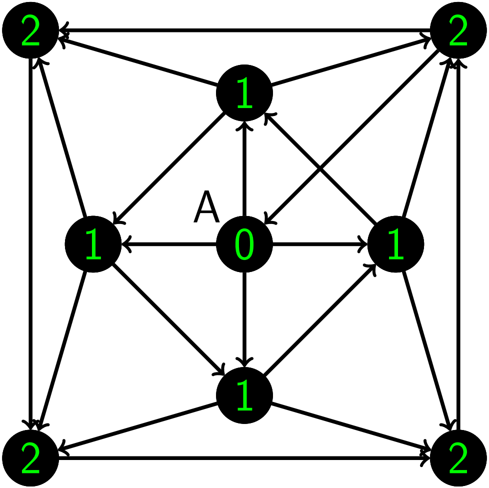

2
2
1
A
1
0
1
1
2
2

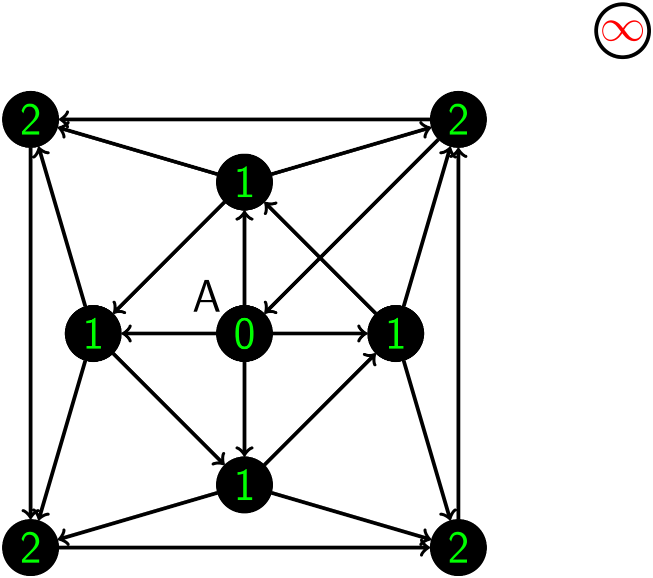


2
2
1
A
1
0
1
1
2
2


2
2
1
A
1
0
1
1
2
2


2
2
1
A
1
0
1
1
2
2


2
2
1
A
1
0
1
1
2
2



∞
2
2
1
A
1
0
1
1
2
2

Outline

Breadth-first search

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Breadth-first search

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Breadth-first search

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Breadth-first search

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Breadth-first search

$\texttt{BFS}(G, A)$

$\texttt{for all } u \in V\texttt{:}$
  $\texttt{dist}[u] \leftarrow \infty$
$\texttt{dist}[A] \leftarrow 0$
$Q \leftarrow \{A\}$ $\{\texttt{queue containing just } A\}$
$\texttt{while } Q \texttt{ is not empty:}$
  $u \leftarrow \texttt{Dequeue}(Q)$
  $\texttt{for all } (u, v) \in E\texttt{:}$
    $\texttt{if dist}[v] = \infty\texttt{:}$
      $\texttt{Enqueue}(Q, v)$
      $\texttt{dist}[v] \leftarrow \texttt{dist}[u] + 1$

Breadth-first search

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Breadth-first search

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Breadth-first search

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Breadth-first search

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Breadth-first search

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Breadth-first search

$\mathtt{BFS}(G, A)$

```
for all u ∈ V:
  dist[u] ← ∞
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

Running time

Lemma

The running time of breadth-first search is $O(|E| + |V|)$.

Proof

Running time

Lemma

The running time of breadth-first search is $O(|E| + |V|)$.

Proof

- Each vertex is enqueued at most once

Running time

Lemma

The running time of breadth-first search is $O(|E| + |V|)$.

Proof

- Each vertex is enqueued at most once
- Each edge is examined either once (for directed graphs) or twice (for undirected graphs) $\square$

Outline

Reachability

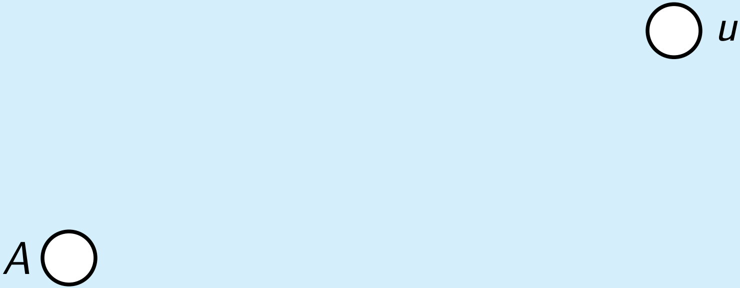
Definition

Node u is reachable from node A if there is a path from A to u

Lemma

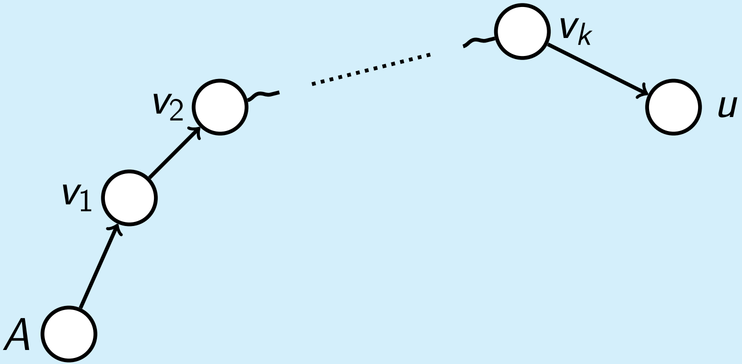
Reachable nodes are discovered at some point, so they get a finite distance estimate from the source. Unreachable nodes are not discovered at any point, and the distance to them stays infinite.

Proof



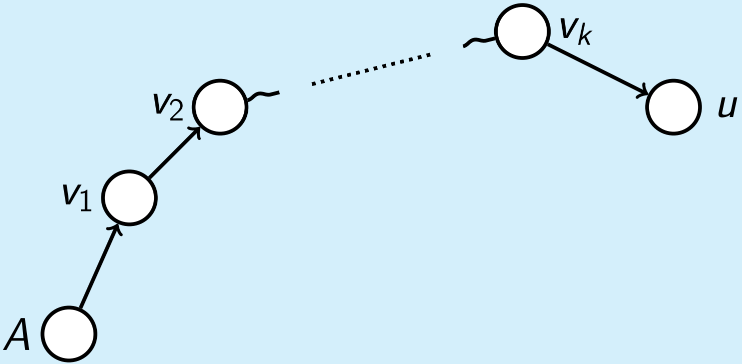
- u — reachable undiscovered closest to A

Proof



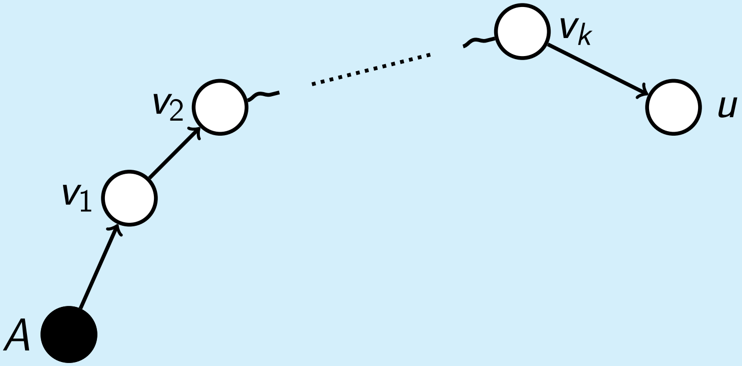
- u — reachable undiscovered closest to A
- $A - v_1 - \cdots - v_k - u$ — shortest path

Proof



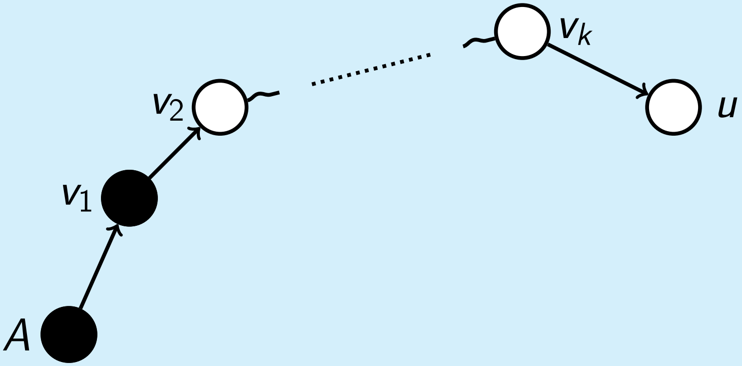
- u — reachable undiscovered closest to A
- $A - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



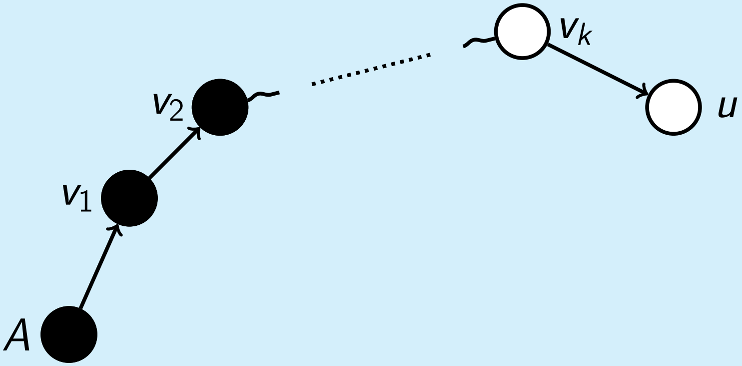
- u — reachable undiscovered closest to A
- $A - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



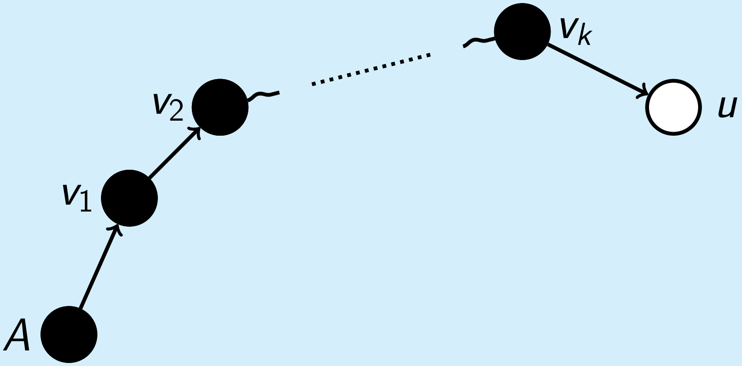
- u — reachable undiscovered closest to A
- $A - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



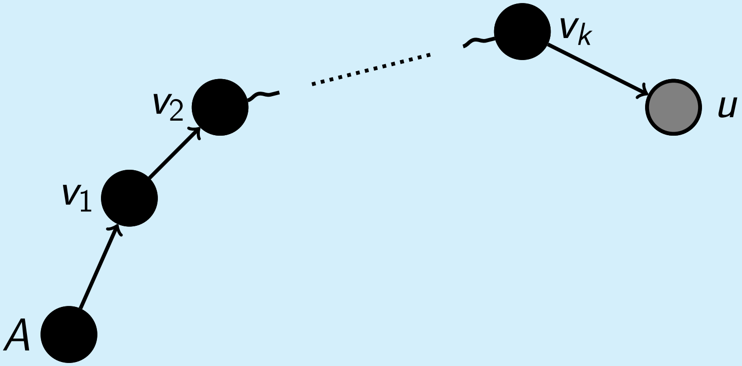
- u — reachable undiscovered closest to A
- $A - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



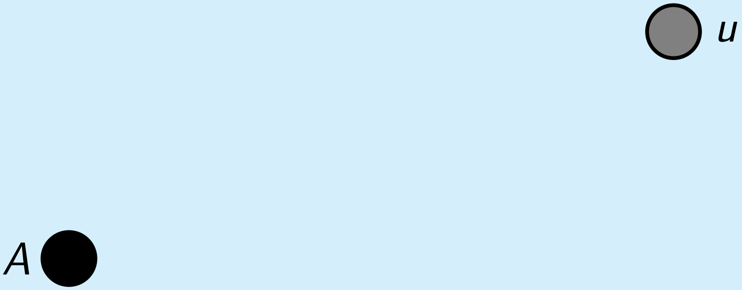
- u — reachable undiscovered closest to A
- $A - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



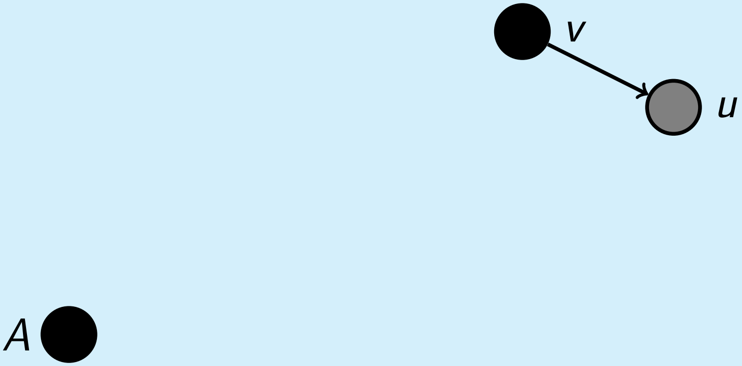
- u — reachable undiscovered closest to A
- $A - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



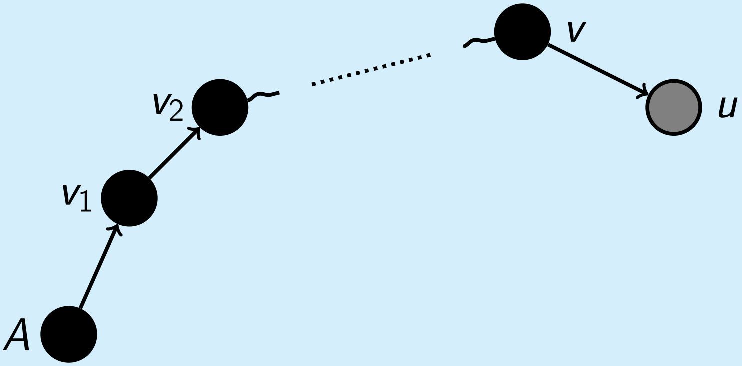
- u — first unreachable discovered

Proof



- u — first unreachable discovered
- u was discovered while processing v

Proof



- u — first unreachable discovered
- u was discovered while processing v
- u is reachable through v $\square$

Order Lemma

Lemma

By the time node u at distance d from A is dequeued, all the nodes at distance at most d have already been discovered (enqueued).

Order Lemma Proof

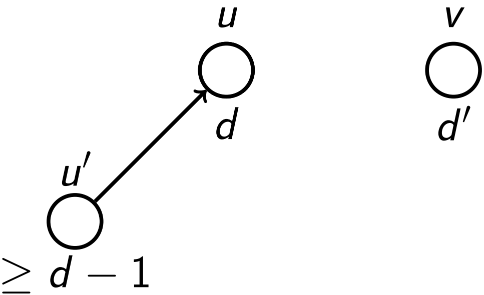
Consider the first time the order was broken

Order Lemma Proof

Consider the first time the order was broken

$d' \leq d$

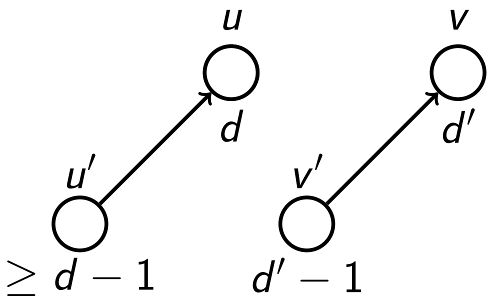
Order Lemma Proof



Consider the first time the order was broken

$d' \leq d$

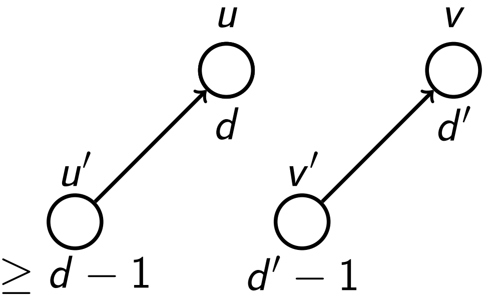
Order Lemma Proof



Consider the first time the order was broken

$d' \leq d$

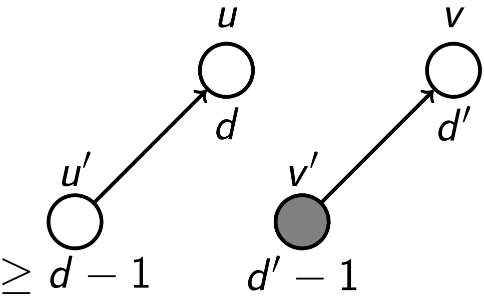
Order Lemma Proof



Consider the first time the order was broken

$d' \leq d \quad \Rightarrow d' - 1 \leq d - 1$, so v' was discovered before u' was dequeued

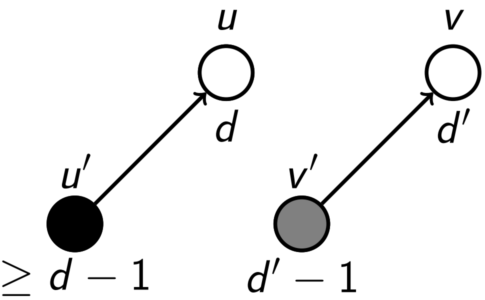
Order Lemma Proof



Consider the first time the order was broken

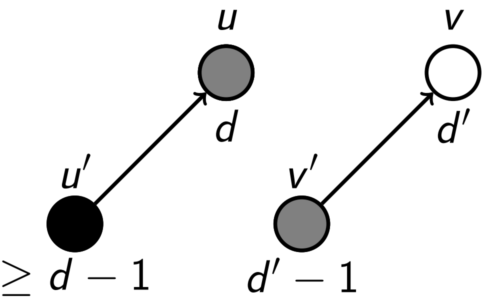
$d' \leq d \quad \Rightarrow d' - 1 \leq d - 1$, so v' was discovered before u' was dequeued

Order Lemma Proof



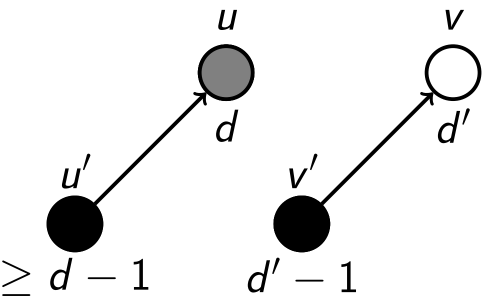
Consider the first time the order was broken
$d' \leq d \quad \Rightarrow d'-1 \leq d-1$, so v' was discovered before u' was dequeued

Order Lemma Proof



Consider the first time the order was broken
$d' \leq d \quad \Rightarrow d' - 1 \leq d - 1$, so v' was discovered before u' was dequeued

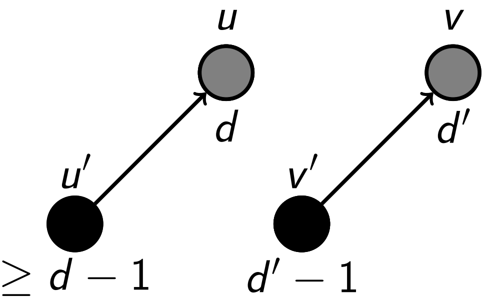
Order Lemma Proof



Consider the first time the order was broken

$d' \leq d \quad \Rightarrow d' - 1 \leq d - 1$, so v' was discovered before u' was dequeued

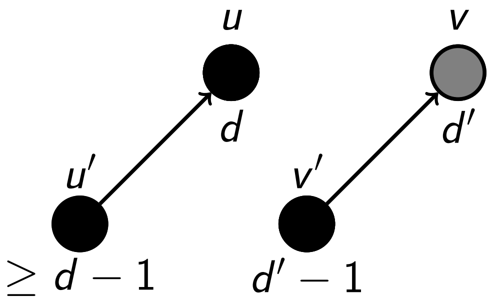
Order Lemma Proof



Consider the first time the order was broken

$d' \leq d \quad \Rightarrow d'-1 \leq d-1$, so v' was discovered before u' was dequeued

Order Lemma Proof



Consider the first time the order was broken
$d' \leq d \quad \Rightarrow d' - 1 \leq d - 1$, so v' was discovered before u' was dequeued

Queue property

Queue:	d	d	d	$\ldots$	d	d	$d+1$	$d+1$	$\ldots$	$d+1$

Lemma

At any moment, if the first node in the queue is at distance d from A, then all the nodes in the queue are either at distance d from A or at distance $d+1$ from A. All the nodes in the queue at distance d go before (if any) all the nodes at distance $d+1$.

Queue property

Proof

- All nodes at distance d were enqueued before first such node is dequeued, so they go before nodes at distance $d + 1$

Queue property

Proof

- All nodes at distance d were enqueued before first such node is dequeued, so they go before nodes at distance $d + 1$
- Nodes at distance $d - 1$ were enqueued before nodes at d, so they are not in the queue anymore

Queue property

Proof

- All nodes at distance d were enqueued before first such node is dequeued, so they go before nodes at distance $d + 1$
- Nodes at distance $d - 1$ were enqueued before nodes at d, so they are not in the queue anymore
- Nodes at distance $> d + 1$ will be discovered when all d are gone $\square$

Outline

Correct distances

Lemma

When node u is discovered (enqueued), `dist[u]` is assigned exactly $d(A, u)$.

Correct distances

Proof

- Use mathematical induction

Correct distances

Proof

- Use mathematical induction
- Base: when A is discovered, `dist[A]` is assigned $0 = d(A, A)$

Correct distances

Proof

- Use mathematical induction
- Base: when A is discovered, `dist[A]` is assigned $0 = d(A, A)$
- Inductive step: suppose proved for all nodes at distance $\leq k$ from $A \rightarrow$ prove for nodes at distance $k + 1$

Correct distances

Proof

- Take a node v at distance $k + 1$ from A

Correct distances

Proof

- Take a node v at distance $k + 1$ from A
- v was discovered while processing u

Correct distances

Proof

- Take a node v at distance $k+1$ from A
- v was discovered while processing u
- $d(A,v) \leq d(A,u)+1 \Rightarrow d(A,u) \geq k$

Correct distances

Proof

- Take a node v at distance $k+1$ from A
- v was discovered while processing u
- $d(A,v) \leq d(A,u)+1 \Rightarrow d(A,u) \geq k$
- v is discovered after u is dequeued, so $d(A,u) < d(A,v) = k+1$

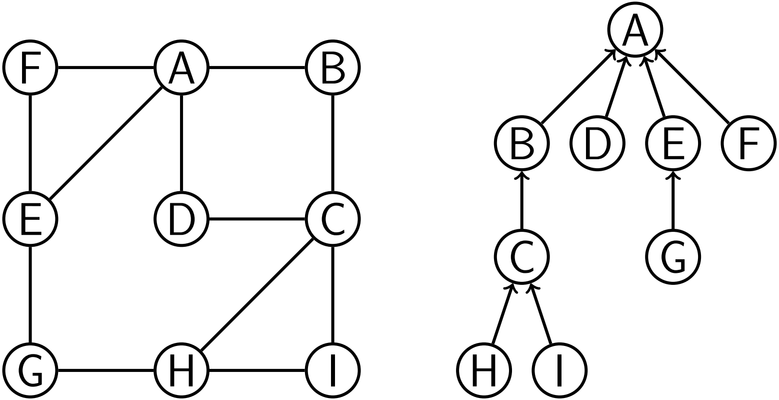
Correct distances

Proof

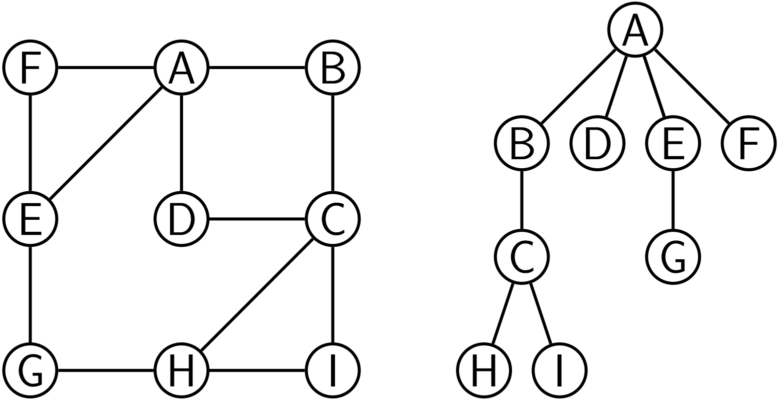
- Take a node v at distance $k+1$ from A
- v was discovered while processing u
- $d(A,v) \leq d(A,u)+1 \Rightarrow d(A,u) \geq k$
- v is discovered after u is dequeued, so $d(A,u) < d(A,v) = k+1$
- So $d(A,u) = k$, and $\text{dist}[v] \leftarrow \text{dist}[u] + 1 = k+1$ $\square$

Outline

Shortest-path tree



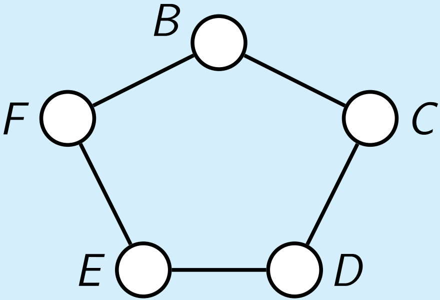
Shortest-path tree



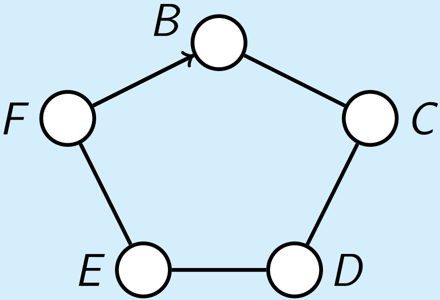
Lemma

Shortest-path tree is indeed a tree, i.e. it doesn't contain cycles (it is a connected component by construction).

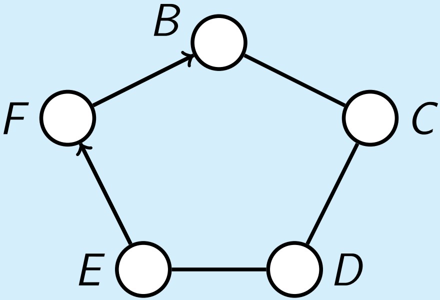
Proof



Proof

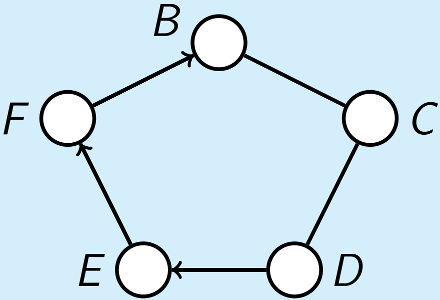


Proof



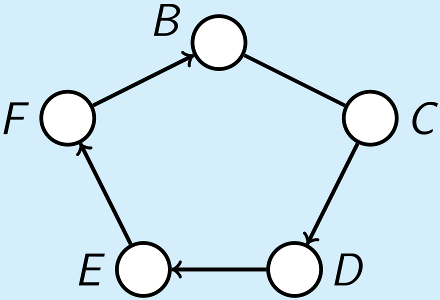
- Only one outgoing edge from each node

Proof



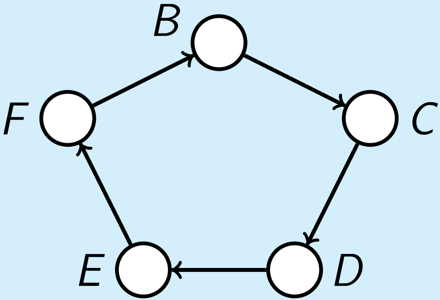
- Only one outgoing edge from each node

Proof



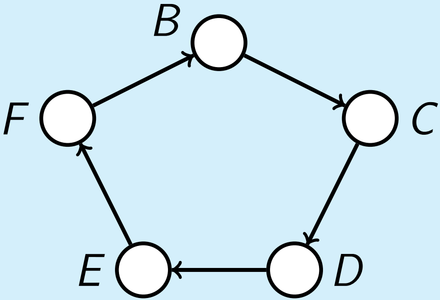
- Only one outgoing edge from each node

Proof



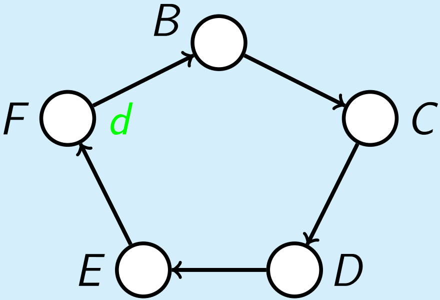
- Only one outgoing edge from each node

Proof



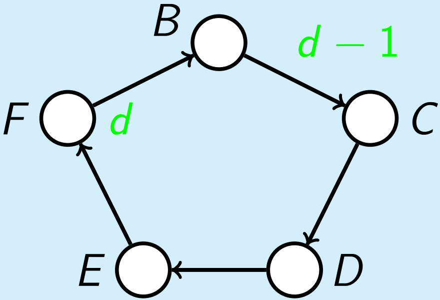
- Only one outgoing edge from each node
- Distance decreases after going by edge

Proof



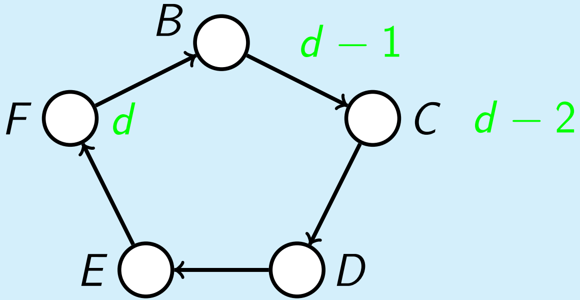
- Only one outgoing edge from each node
- Distance decreases after going by edge

Proof



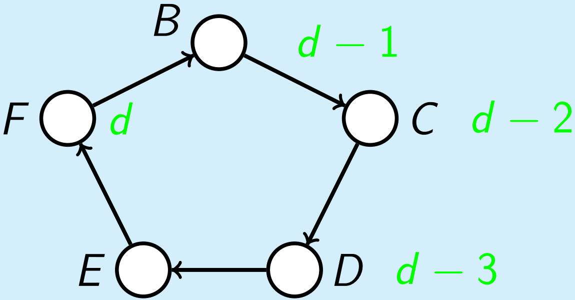
- Only one outgoing edge from each node
- Distance decreases after going by edge

Proof



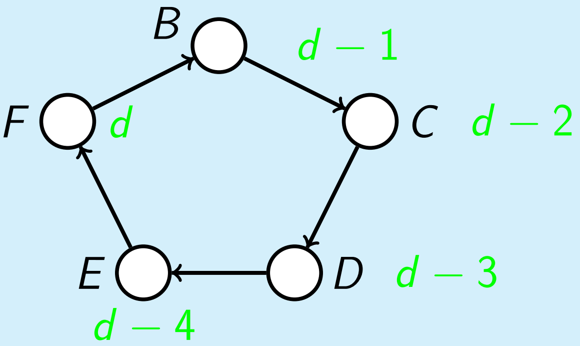
- Only one outgoing edge from each node
- Distance decreases after going by edge

Proof



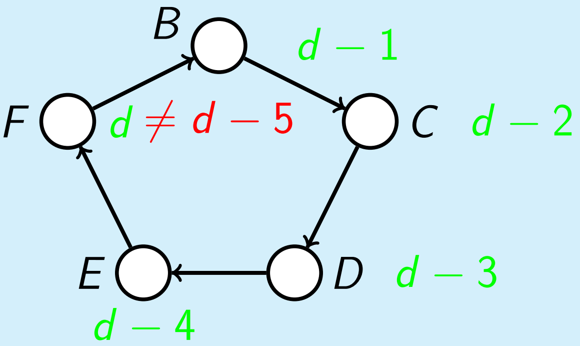
- Only one outgoing edge from each node
- Distance decreases after going by edge

Proof



- Only one outgoing edge from each node
- Distance decreases after going by edge

Proof



- Only one outgoing edge from each node
- Distance decreases after going by edge

Constructing shortest-path tree

BFS(G, A)

```
for all u ∈ V:
  dist[u] ← ∞, prev[u] ← nil
dist[A] ← 0
Q ← {A} {queue containing just A}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1, prev[v] ← u
```

Reconstructing Shortest Path

```
ReconstructPath(A, u, prev)

result ← empty
while u ≠ A:
    result.append(u)
    u ← prev[u]
return Reverse(result)
```

Reconstructing Shortest Path

```
ReconstructPath(A, u, prev)

result ← empty
while u ≠ A:
   result.append(u)
   u ← prev[u]
return Reverse(result)
```

Reconstructing Shortest Path

```
ReconstructPath(A, u, prev)

result ← empty
while u ≠ A:
    result.append(u)
    u ← prev[u]
return Reverse(result)
```

Reconstructing Shortest Path

```
ReconstructPath(A, u, prev)

result ← empty
while u ≠ A:
    result.append(u)
    u ← prev[u]
return Reverse(result)
```

Reconstructing Shortest Path

ReconstructPath(A, u, prev)

```
result ← empty
while u ≠ A:
    result.append(u)
    u ← prev[u]
return Reverse(result)
```

Reconstructing Shortest Path

ReconstructPath(A, u, prev)

```
result ← empty
while u ≠ A:
    result.append(u)
    u ← prev[u]
return Reverse(result)
```

Conclusion

- Can find the minimum number of flight segments to get from one city to another

Conclusion

- Can find the minimum number of flight segments to get from one city to another
- Can reconstruct the optimal path

Conclusion

- Can find the minimum number of flight segments to get from one city to another
- Can reconstruct the optimal path
- Can build the tree of shortest paths from one origin

Conclusion

- Can find the minimum number of flight segments to get from one city to another
- Can reconstruct the optimal path
- Can build the tree of shortest paths from one origin
- Works in $O(|E| + |V|)$