Paths in Graphs: Most Direct Route

Michael Levin

Higher School of Economics

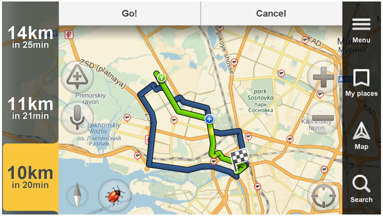
Graph Algorithms
Data Structures and Algorithms

Outline

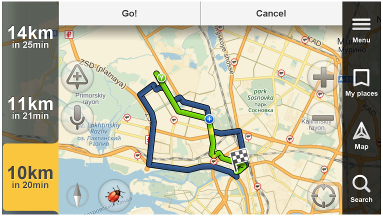
Applications

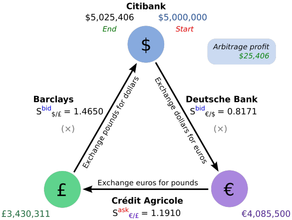
Applications

Applications



Applications





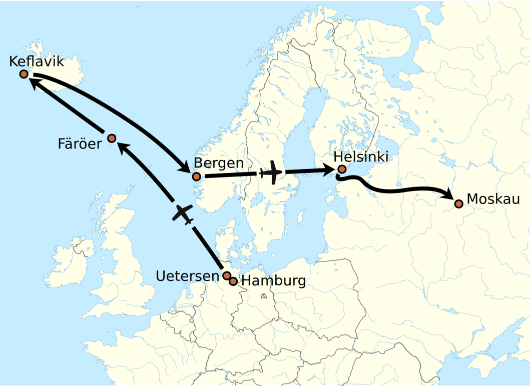
By John Shandy - Own work, CC BY-SA 3.0

The most direct route

What is the minimum number of flight segments to get from Hamburg to Moscow?

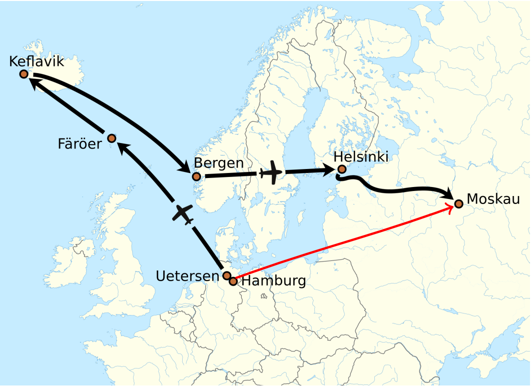
The most direct route

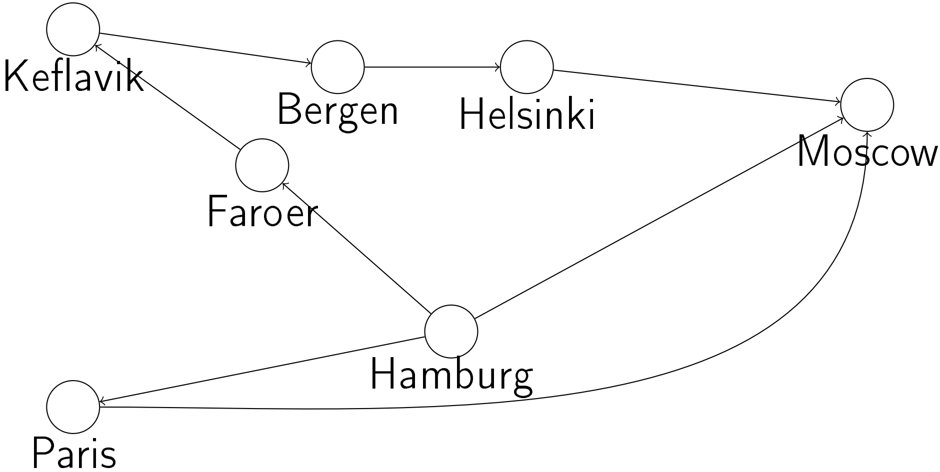
What is the minimum number of flight segments to get from Hamburg to Moscow?

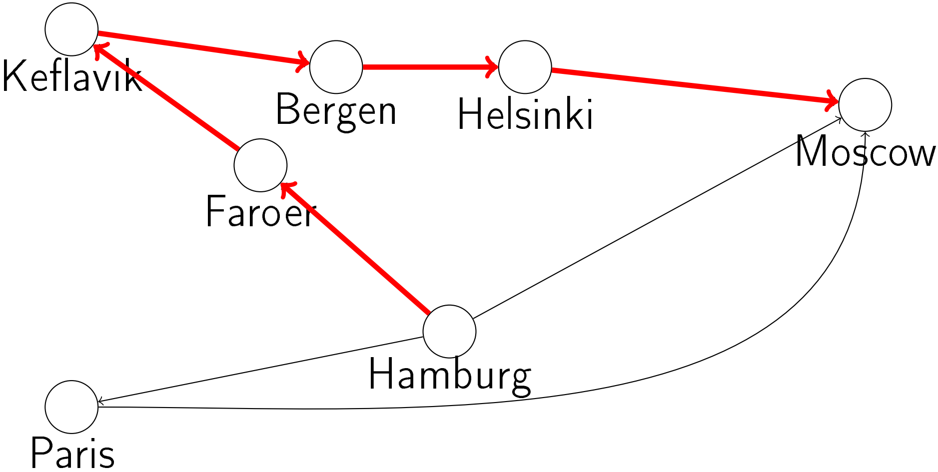


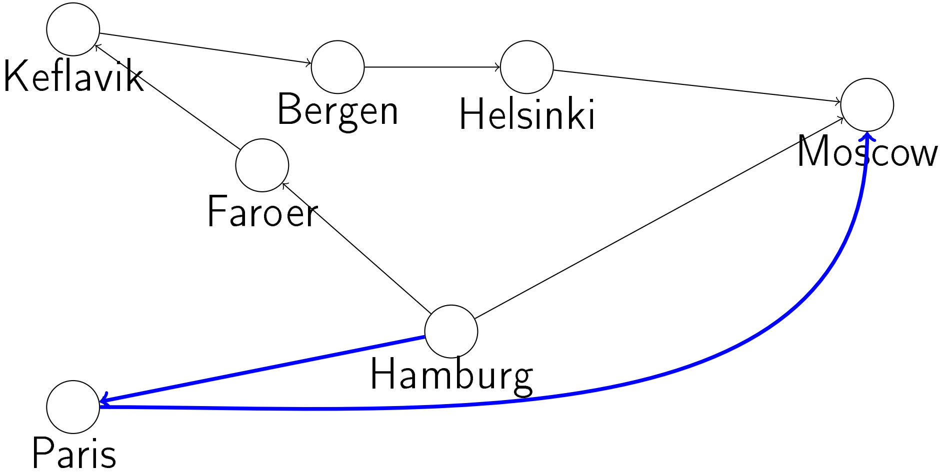
The most direct route

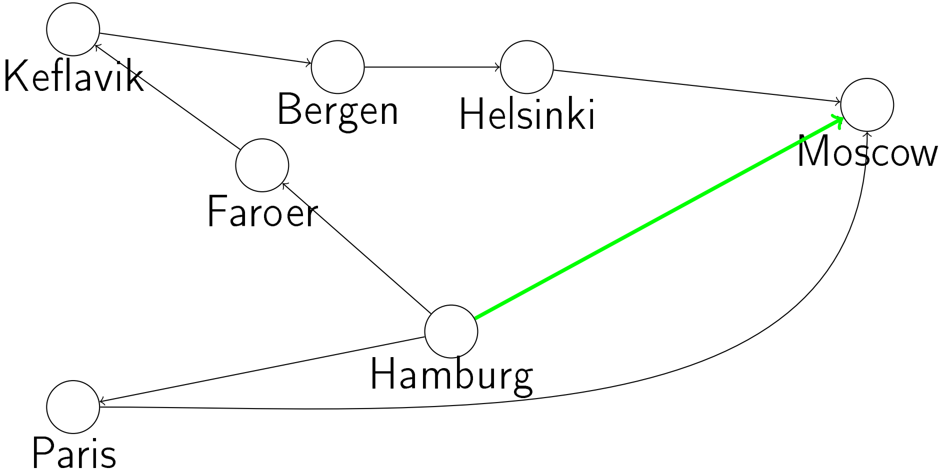
What is the minimum number of flight segments to get from Hamburg to Moscow?




Keflavik
Bergen
Helsinki
Moscow
Faroer
Hamburg
Paris

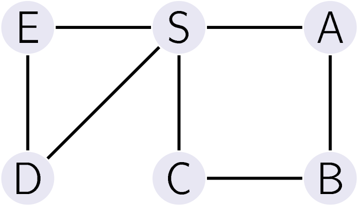

Keflavik
Bergen
Helsinki
Moscow
Faroer
Hamburg
Paris


Keflavik
Bergen
Helsinki
Moscow
Faroer
Hamburg
Paris


Keflavik
Bergen
Helsinki
Moscow
Faroer
Hamburg
Paris

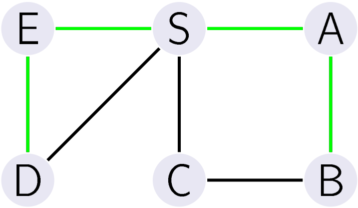
Paths and lengths

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



Paths and lengths

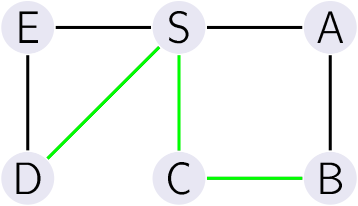
Length of the path $L(P)$ is the number of edges in the path.



$$L(D - E - S - A - B) = 4$$

Paths and lengths

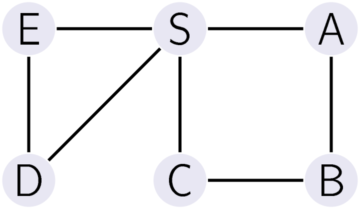
Length of the path $L(P)$ is the number of edges in the path.



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

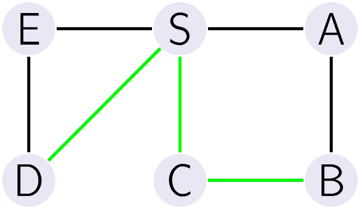
Distances

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



Distances

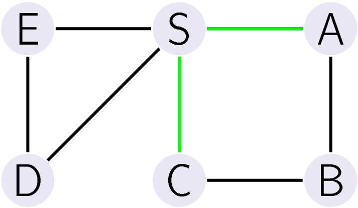
The distance between two vertices is the length of the shortest path between them.



$d(D, B) = 3$

Distances

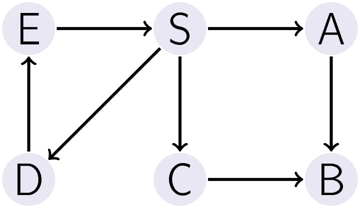
The distance between two vertices is the length of the shortest path between them.



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

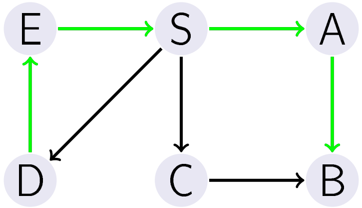
Distances

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



Distances

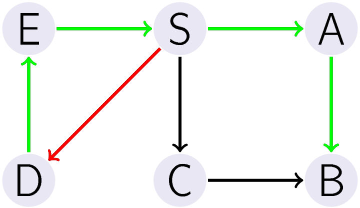
The distance between two vertices is the length of the shortest path between them.



$d(D, B) = 4$

Distances

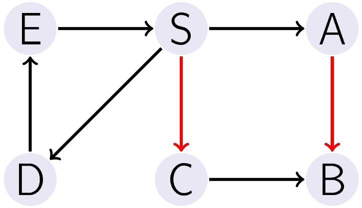
The distance between two vertices is the length of the shortest path between them.



$d(D, B) = 4$

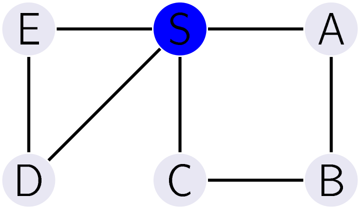
Distances

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

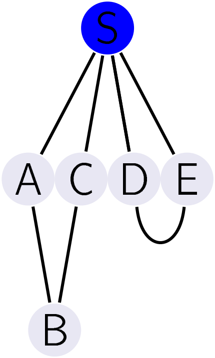


$d(D, B) = 4$
$d(C, A) = \infty$

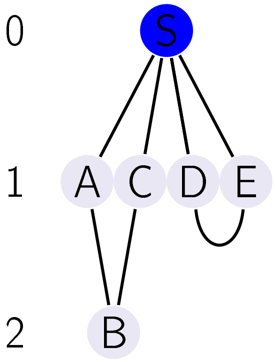
Distances



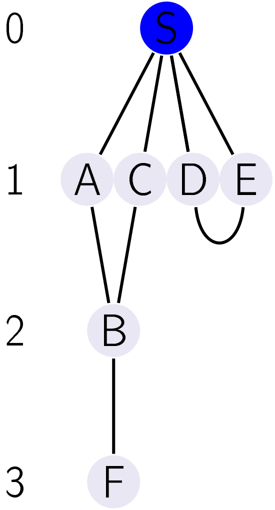
Distance layers



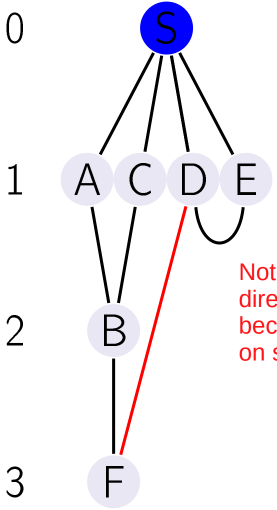
Distance layers



Distance layers

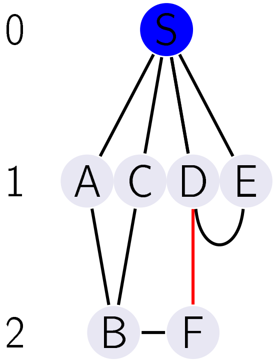


Distance layers

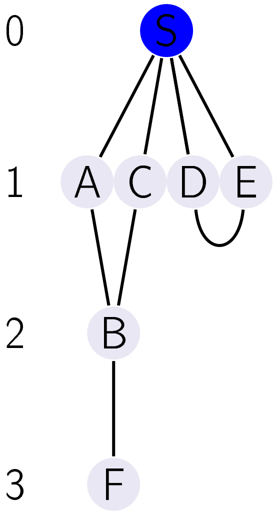


Note that F cannot be directly connected to D because then F needs to be on second level

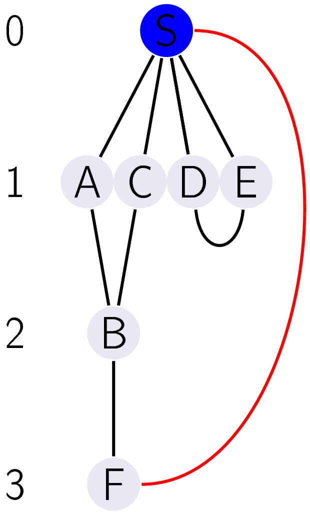
Distance layers



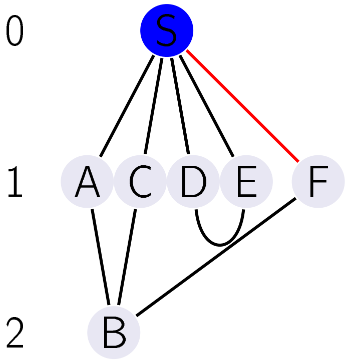
Distance layers



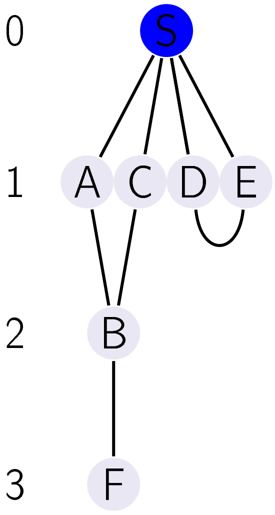
Distance layers



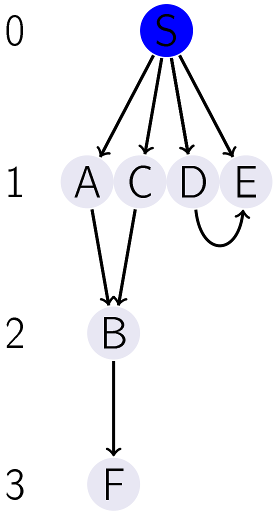
Distance layers



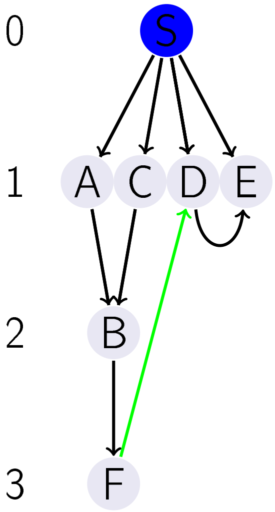
Distance layers



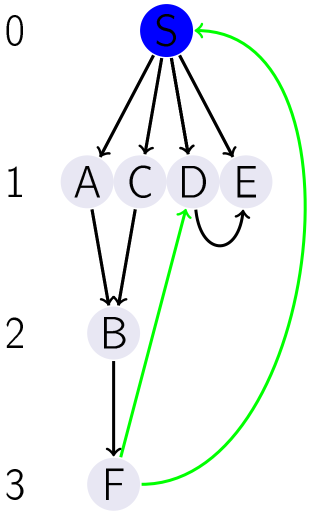
Distance layers



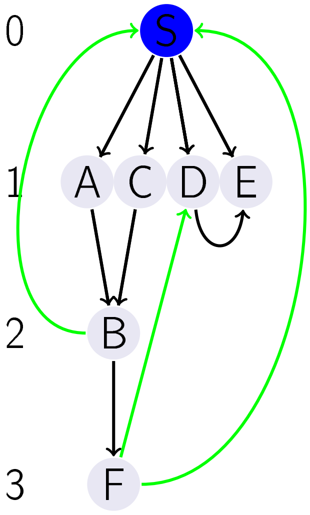
Distance layers



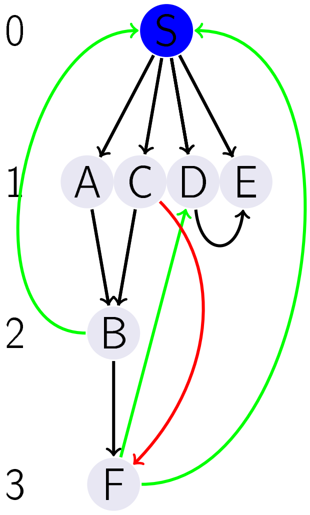
Distance layers



Distance layers



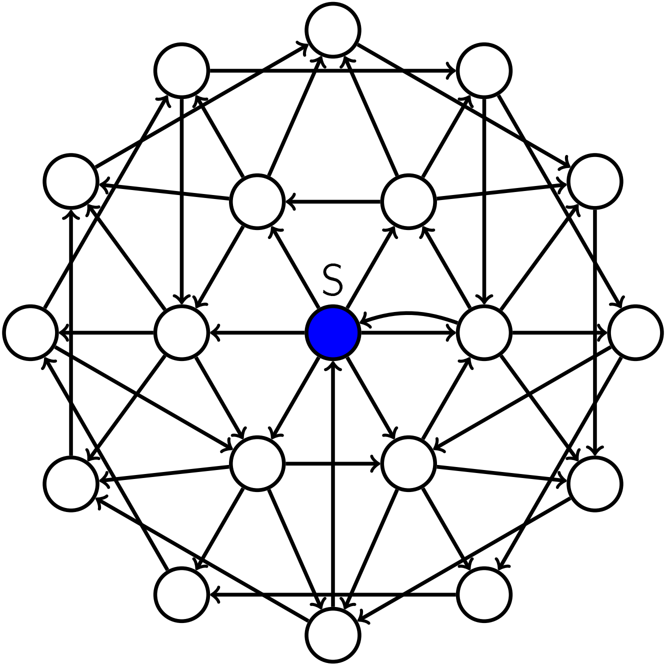
Distance layers

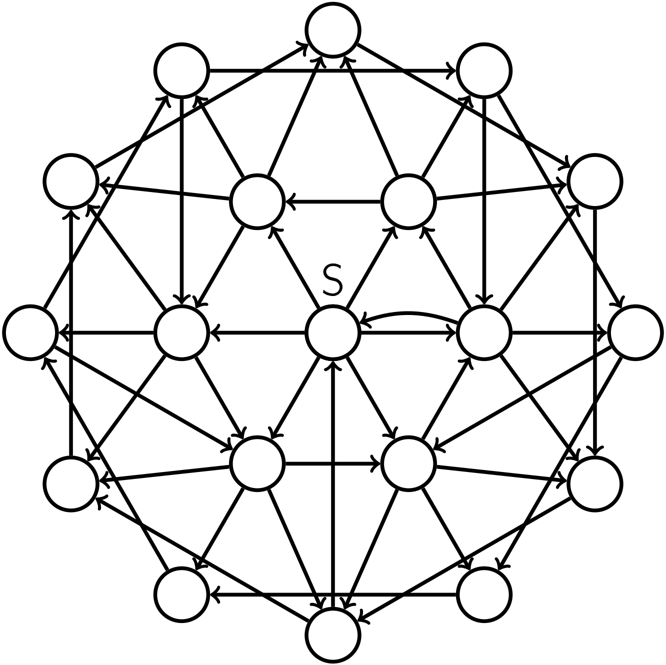


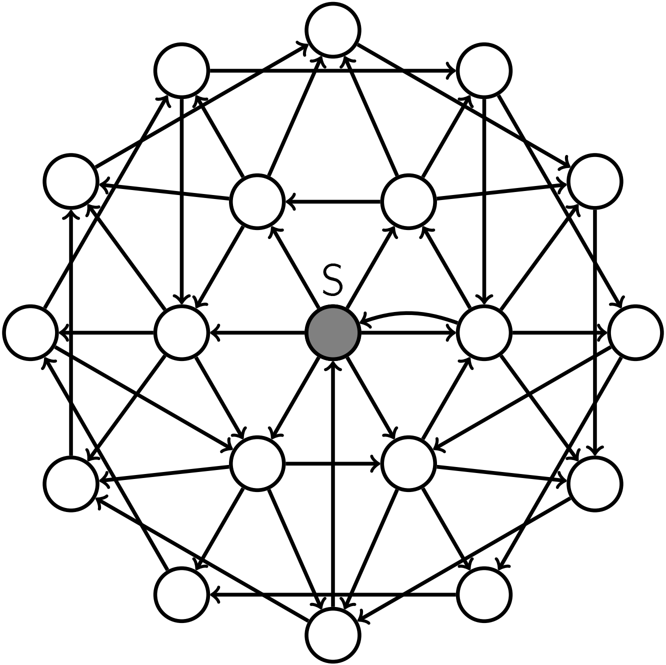
Note that we can have the connection in opposite direction as the shortest path from S does not change. However not in the same direction as then F needs to be brought to level 2

Outline

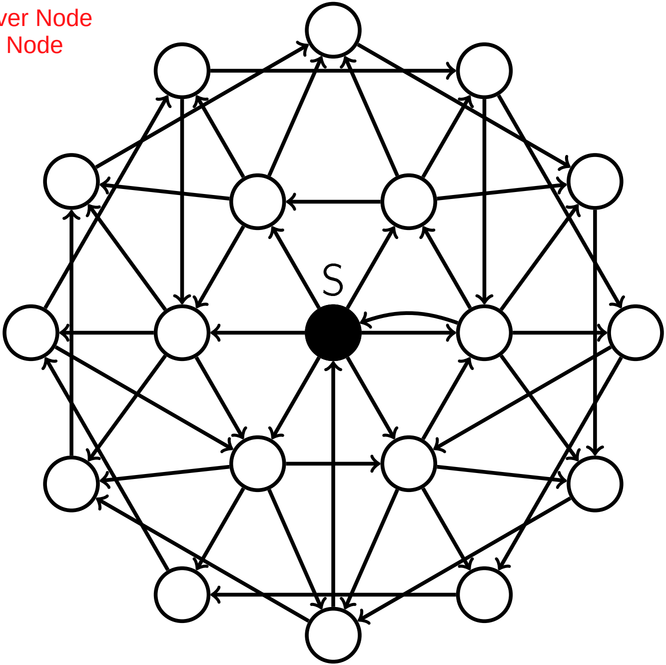
1. Paths and Distances
2. Breadth-first Search
3. Implementation and Analysis
4. Proof of Correctness
5. Shortest-path Tree


S

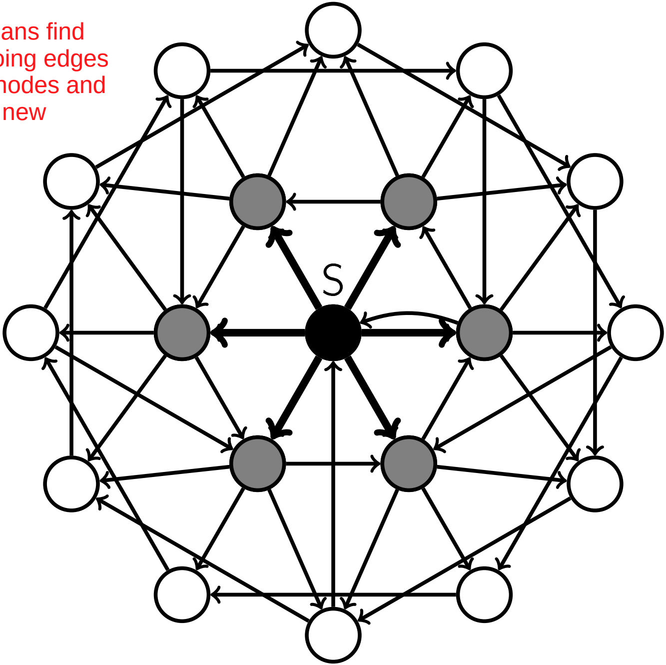

S

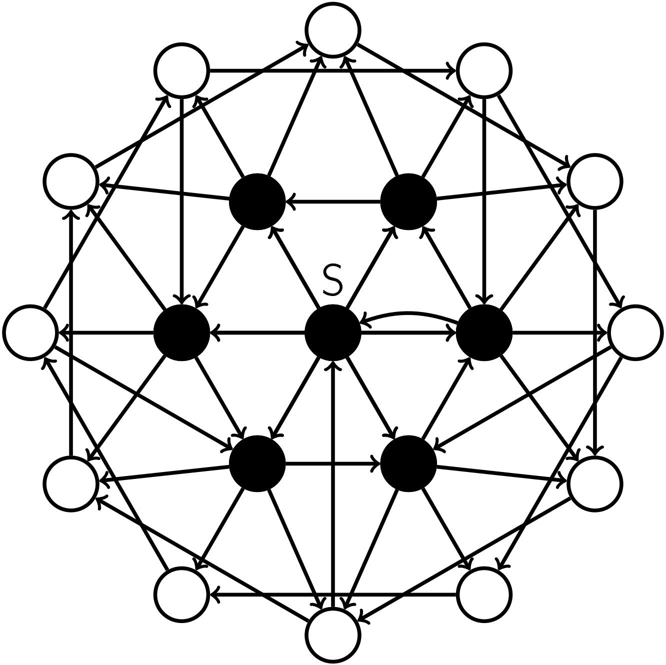

S

Grey - Discover Node
Black - Color Node



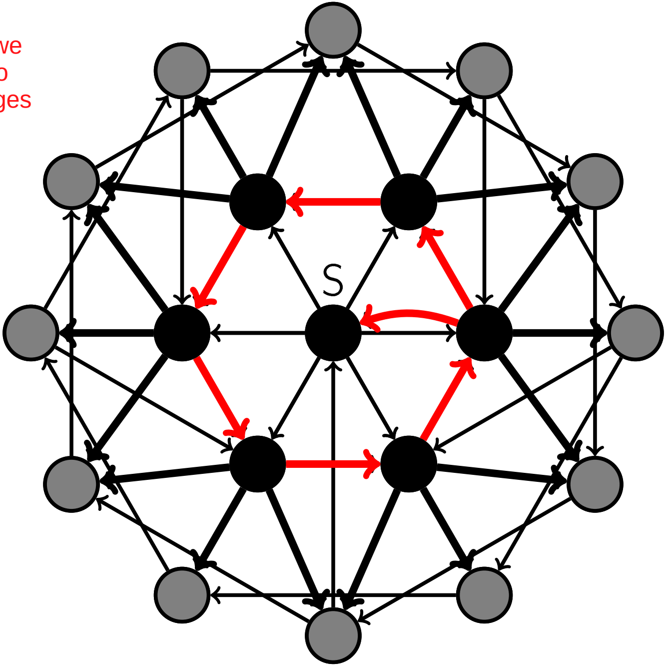
Process means find all the outgoing edges from those nodes and discovering new nodes

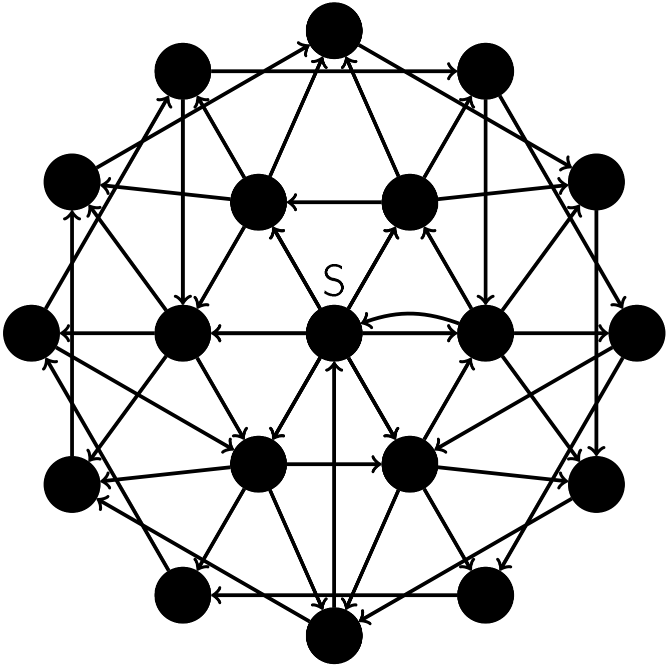


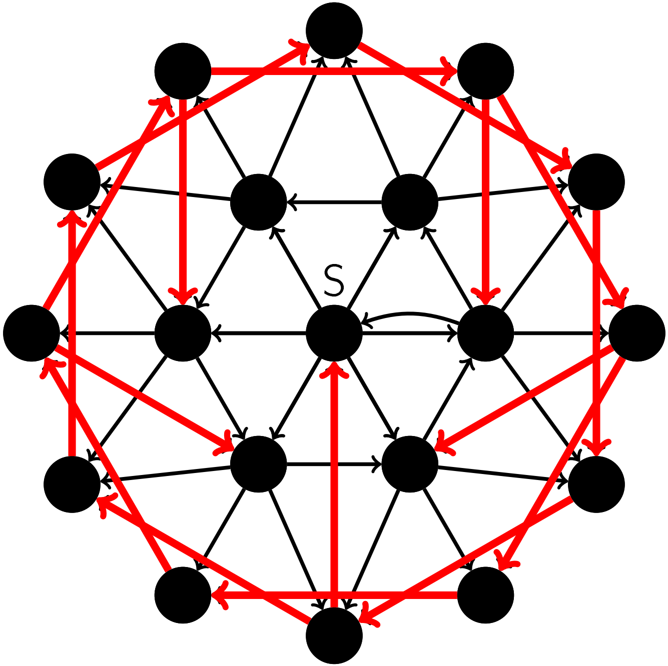

S

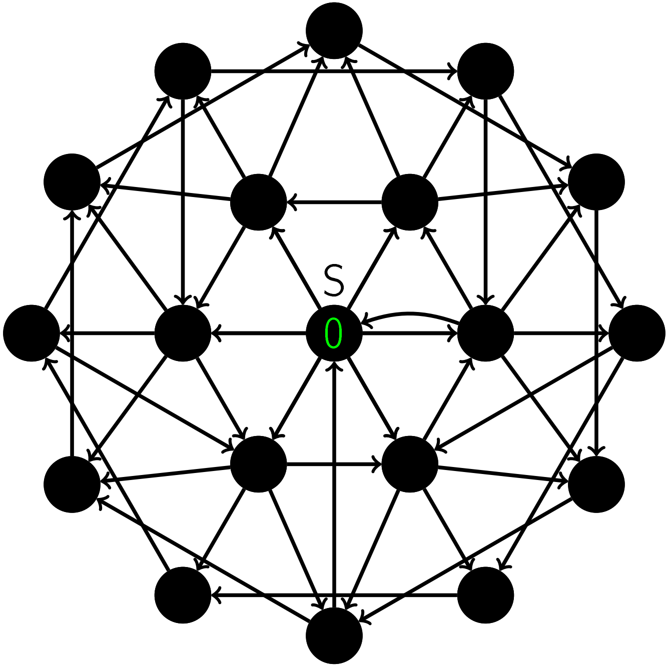
Notice that we don't need to discover edges we have already discovered

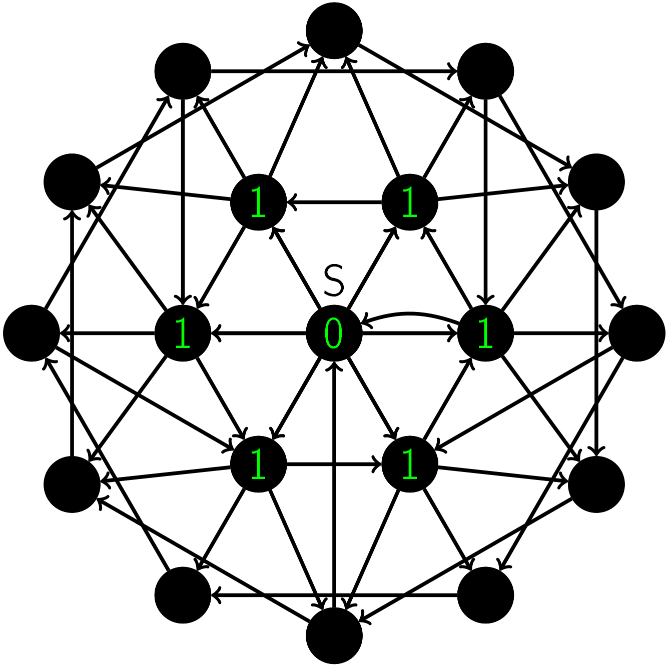
Red means nodes at the end are already discovered and we don't need to do anything



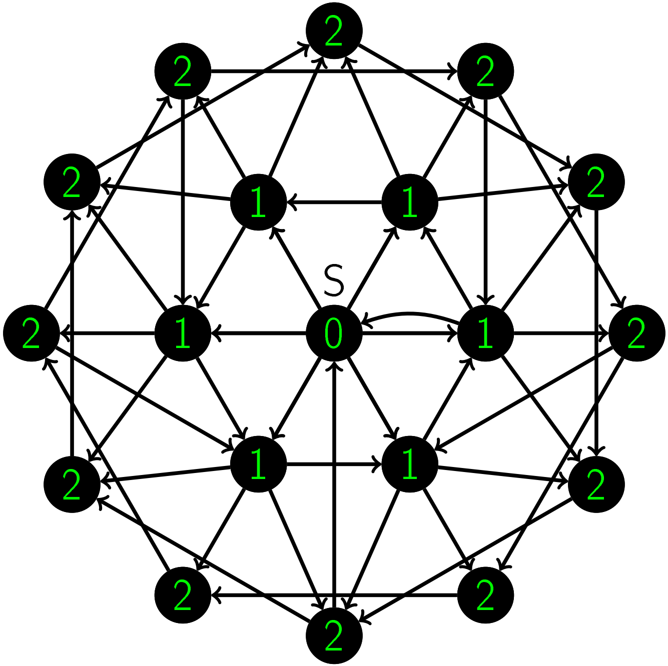

S


S

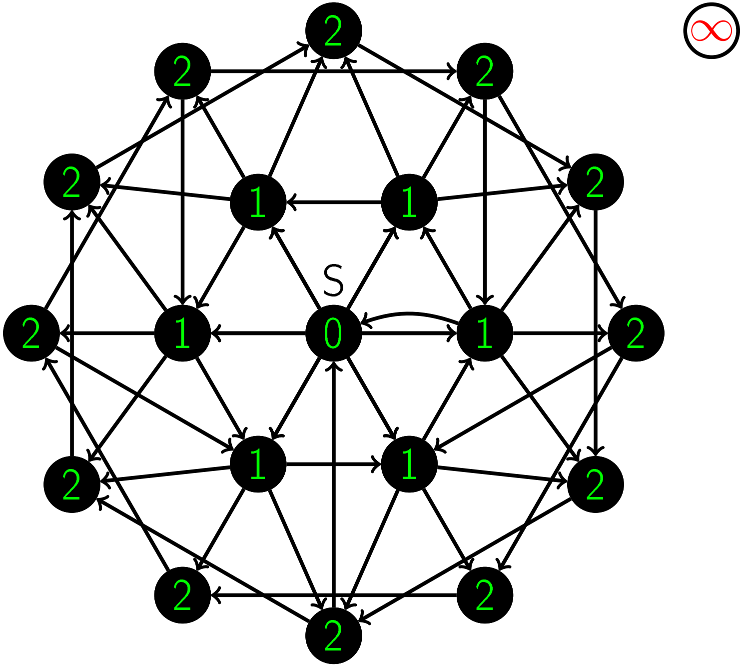

S
0



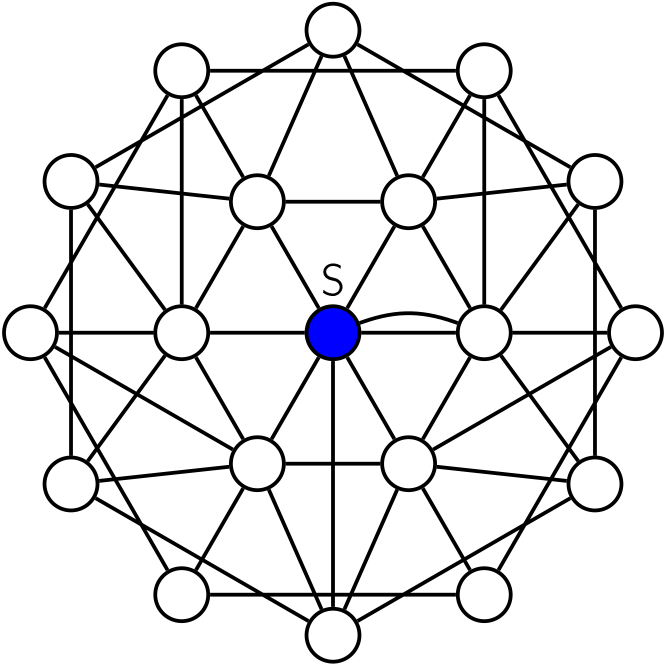
S
0
1
1
1
1
1
1

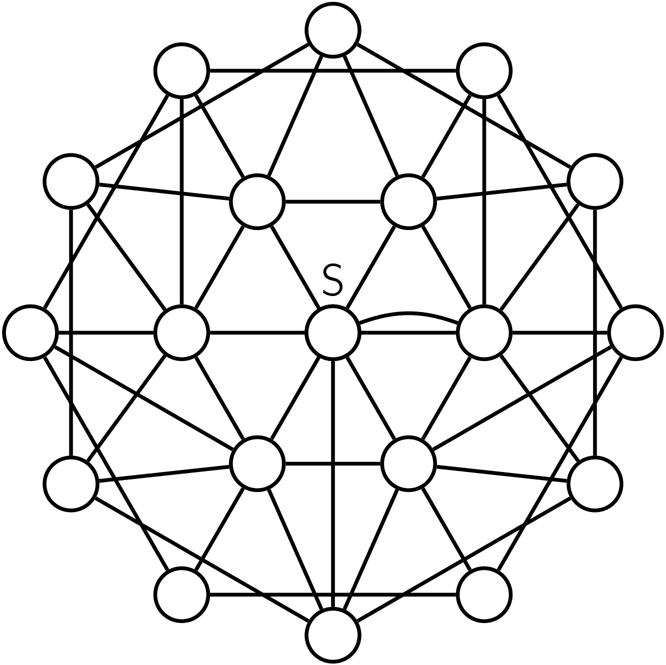


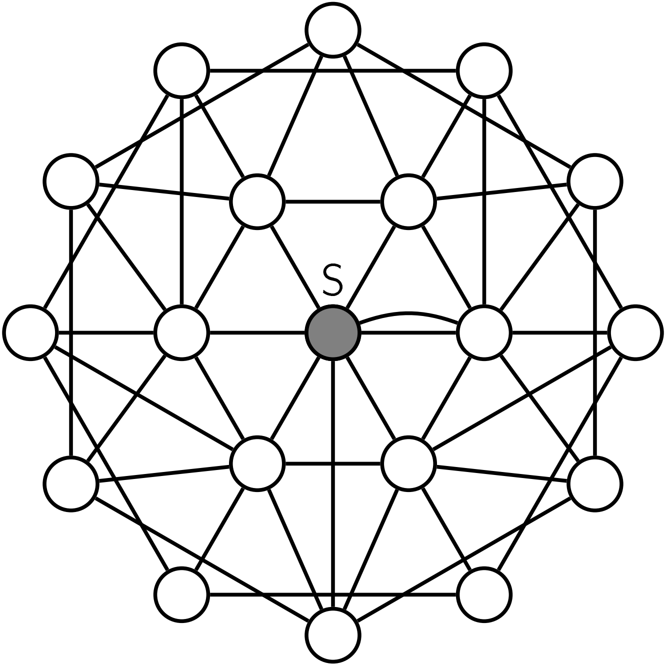
2
2
2
2
1
1
2
S
2
1
0
1
2
2
1
1
2
2
2
2

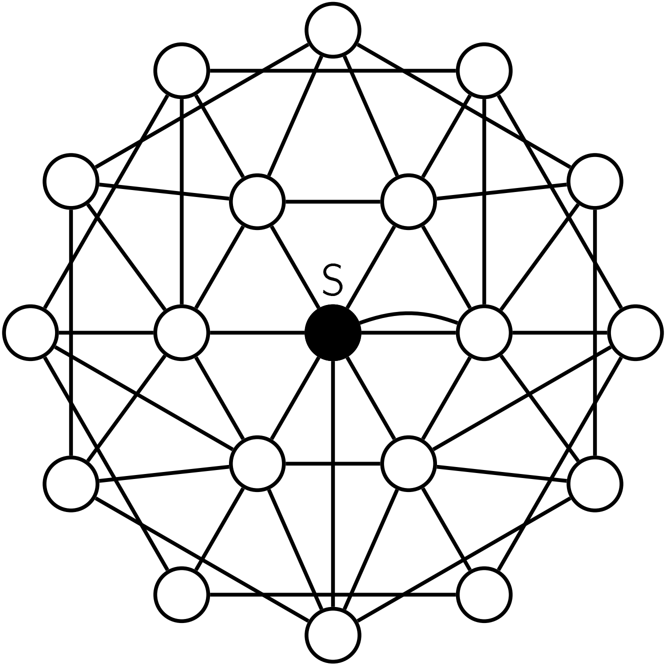


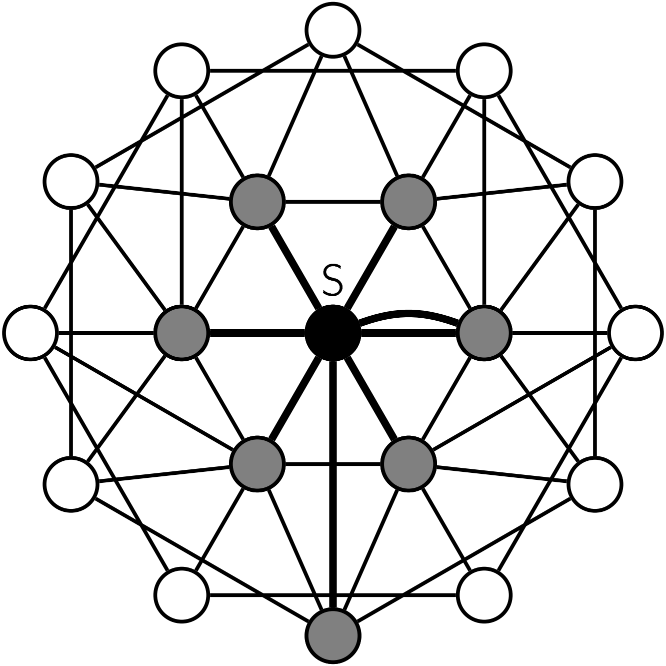
2
2
2
2
2
1
1
2
S
2
1
0
1
2
2
1
1
2
2
2
2
∞

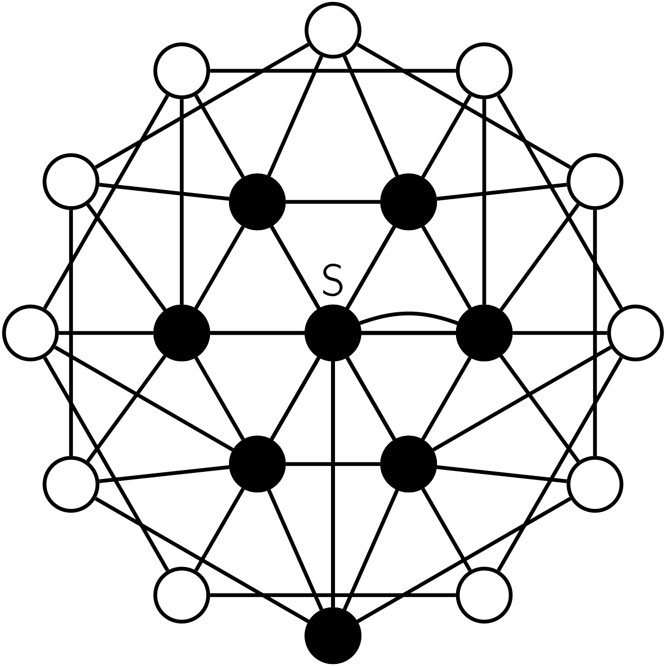

S

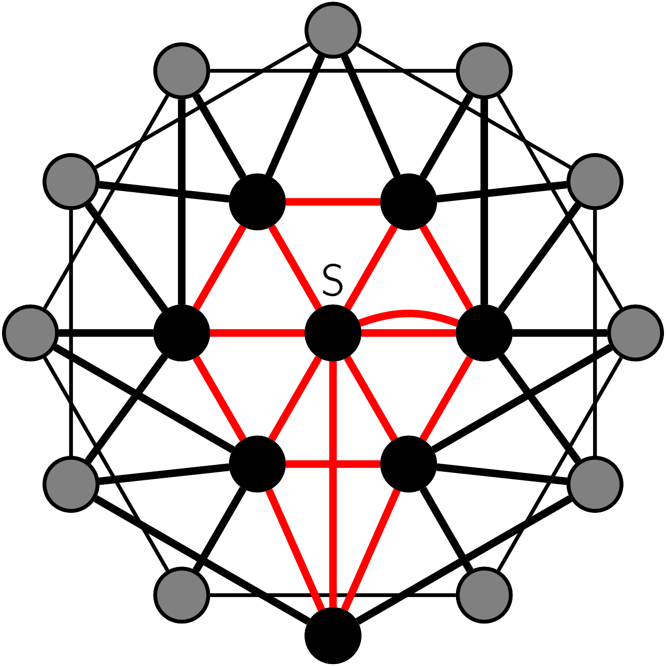

S

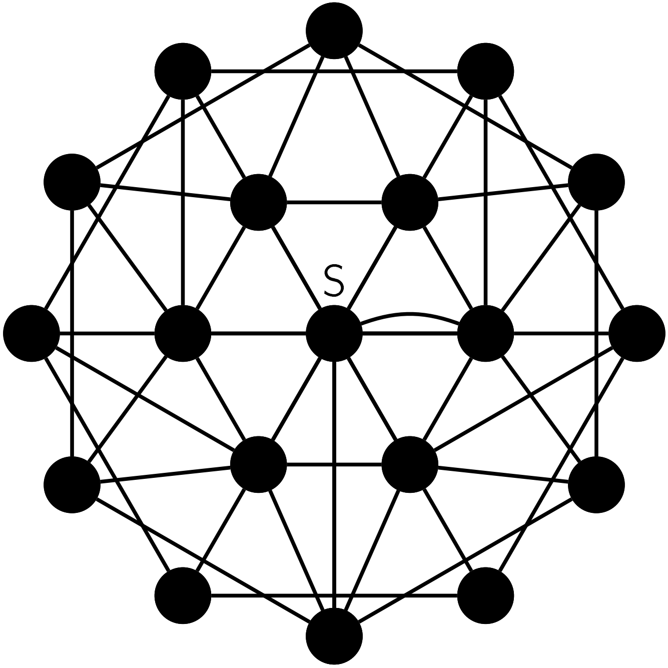

S

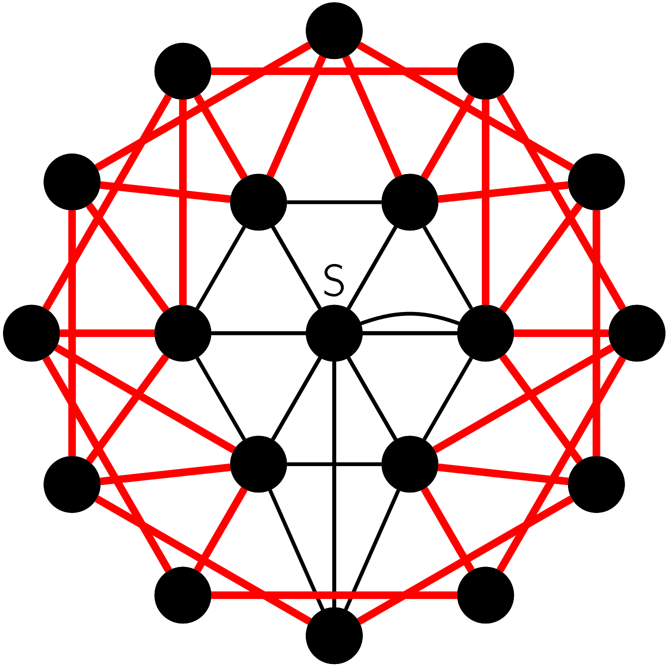

S

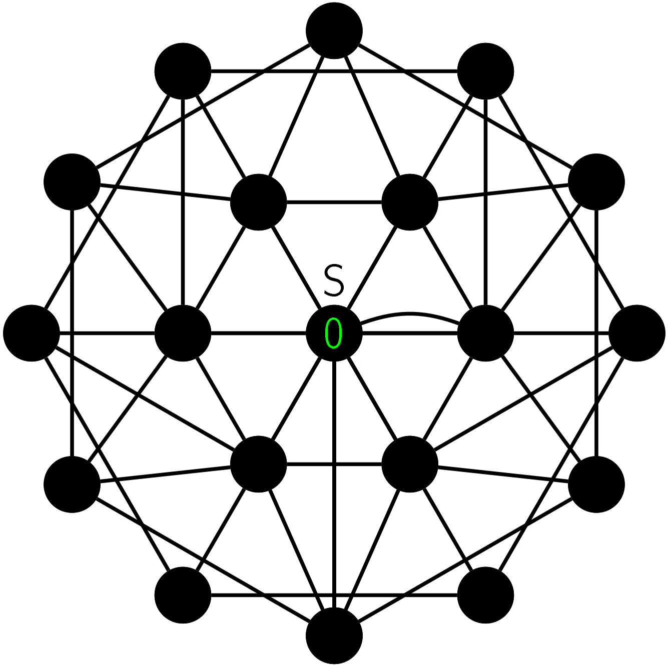

S

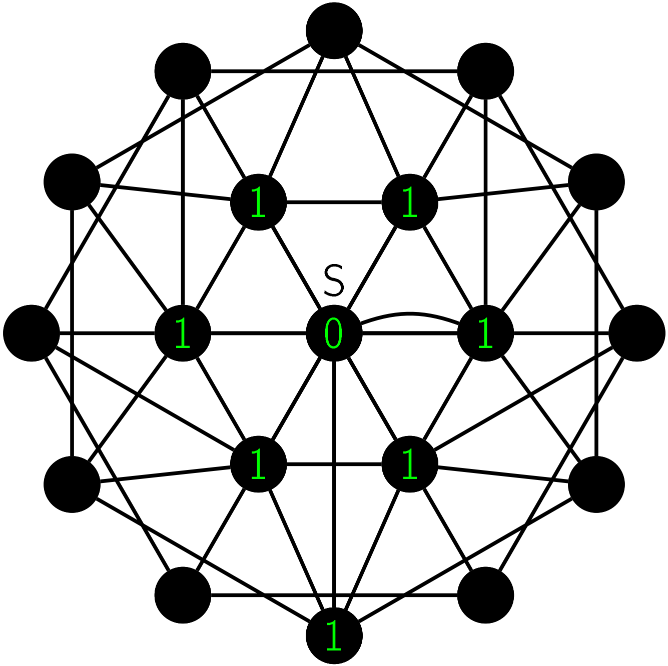

S

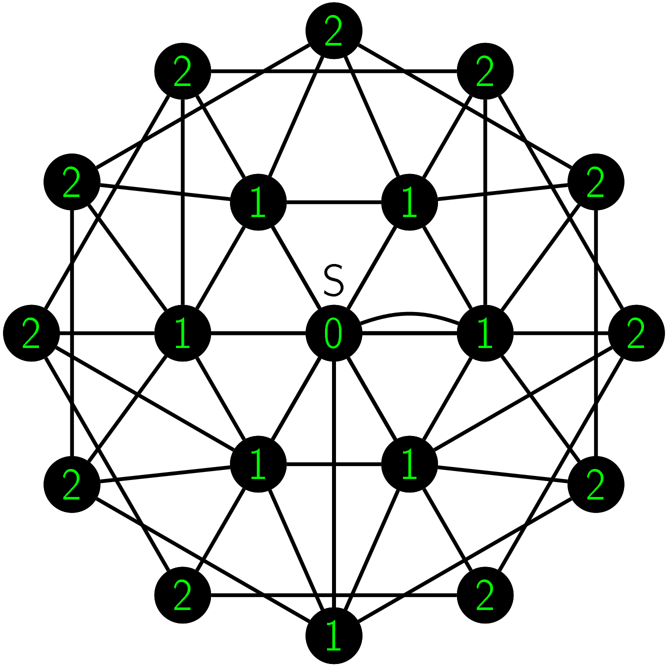

S


S

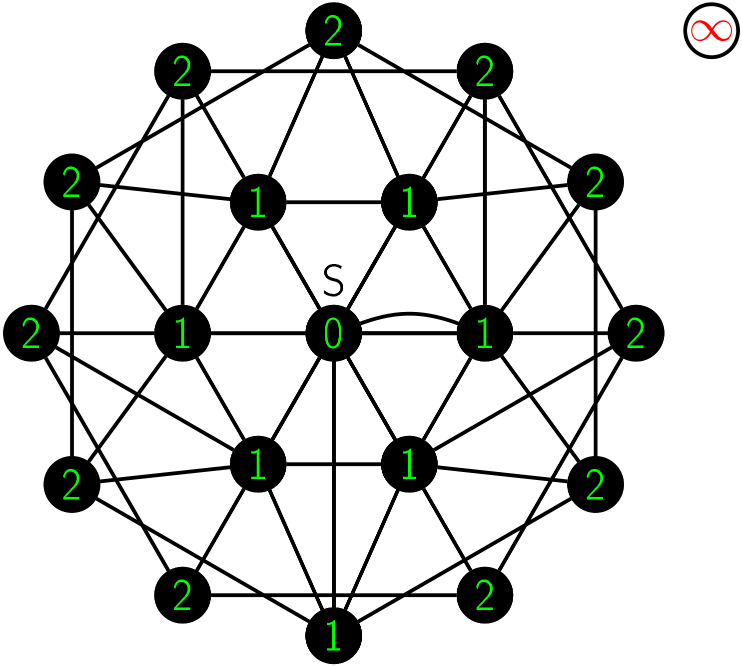

S

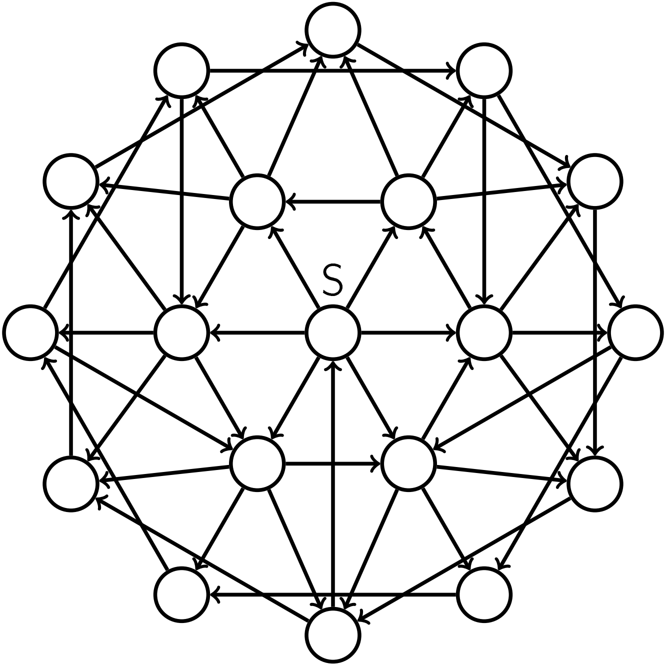

S
0

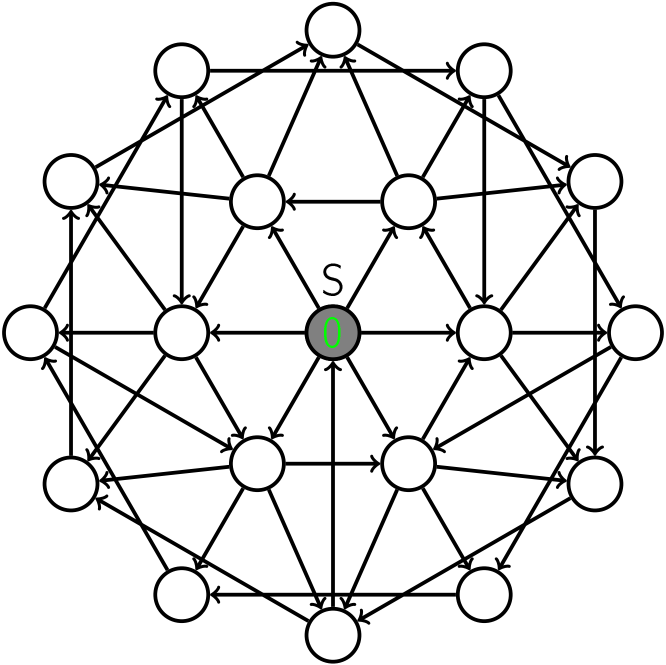

S
0
1
1
1
1
1
1
1
1



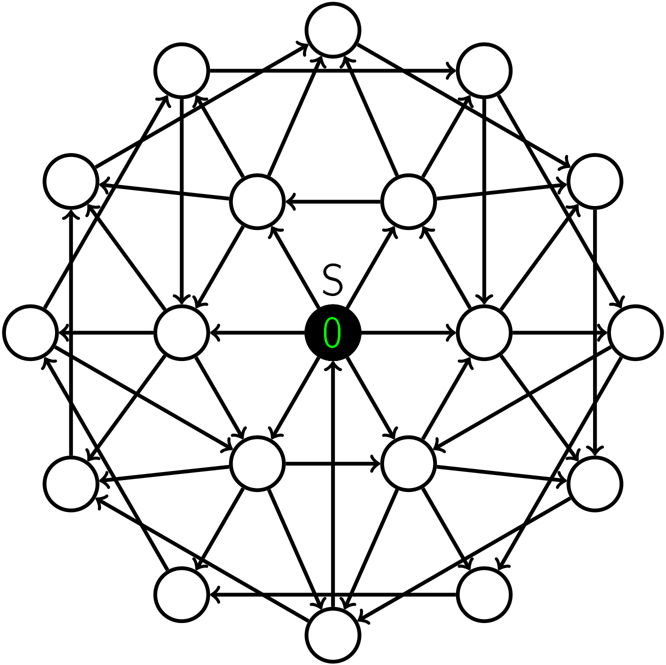
S
0
1
2


∞
2
2
2
2
2
1
1
S
2
1
0
1
2
2
1
1
2
2
2
1


S

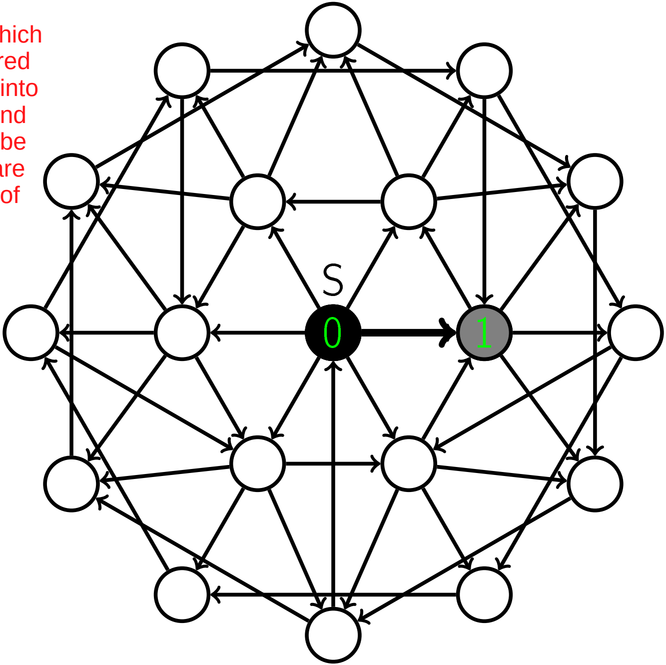


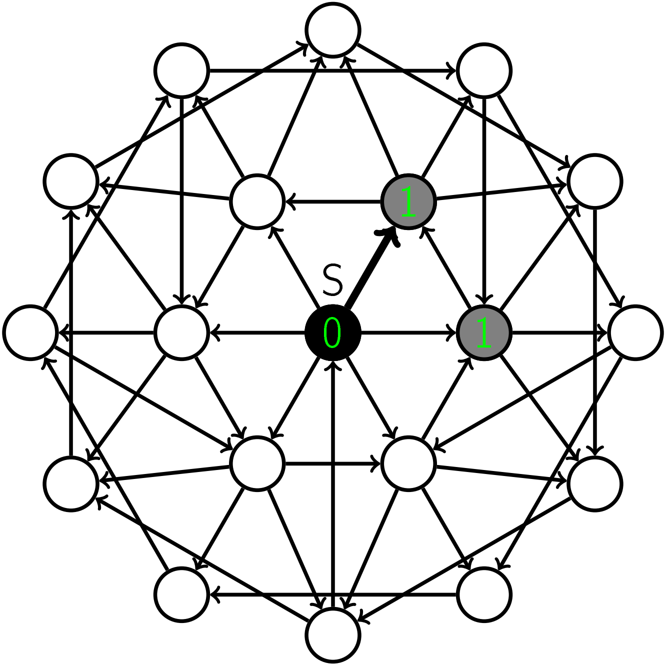
S
0

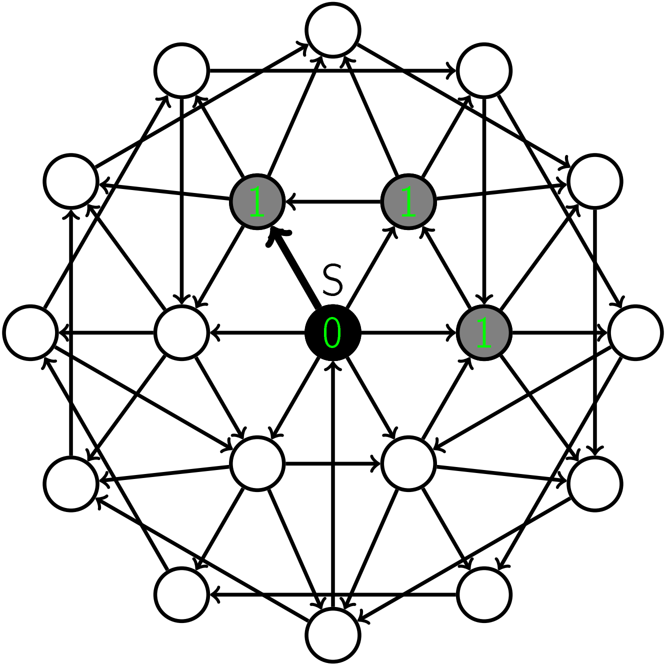

S
0

The node which are discovered are pushed into the queue and the node to be processed are popped out of

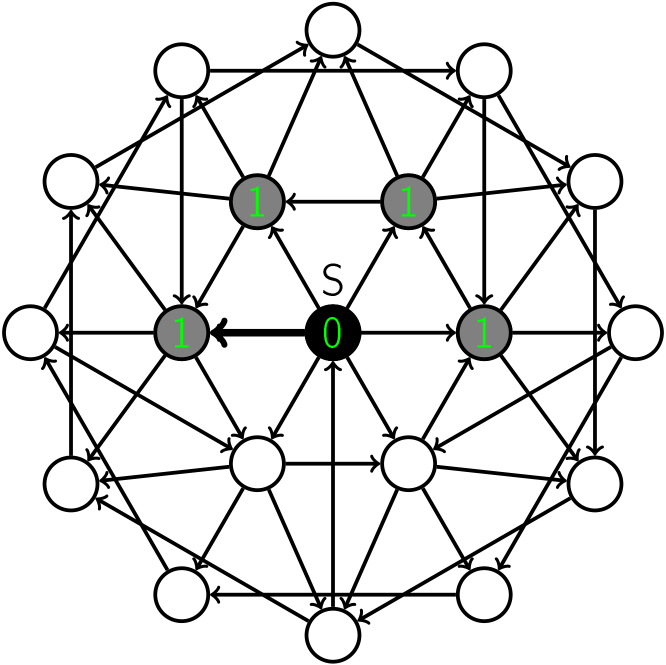
Hence the nodes discovered earlier will also be processed earlier

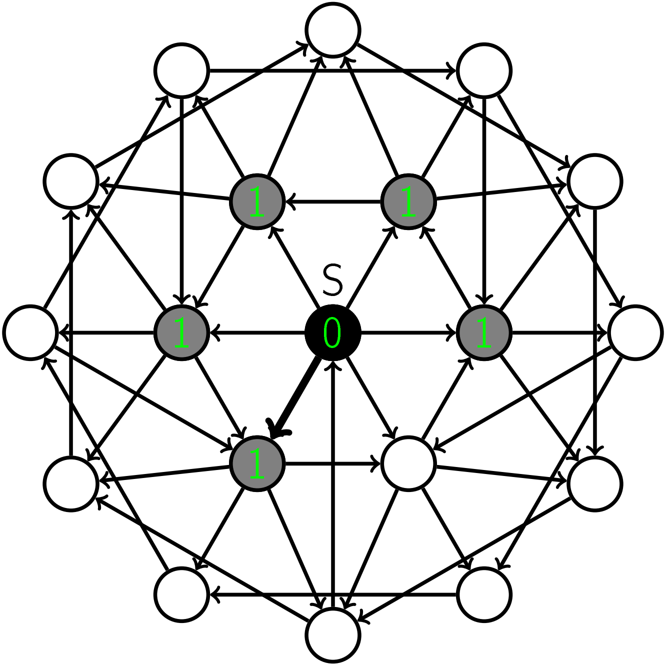


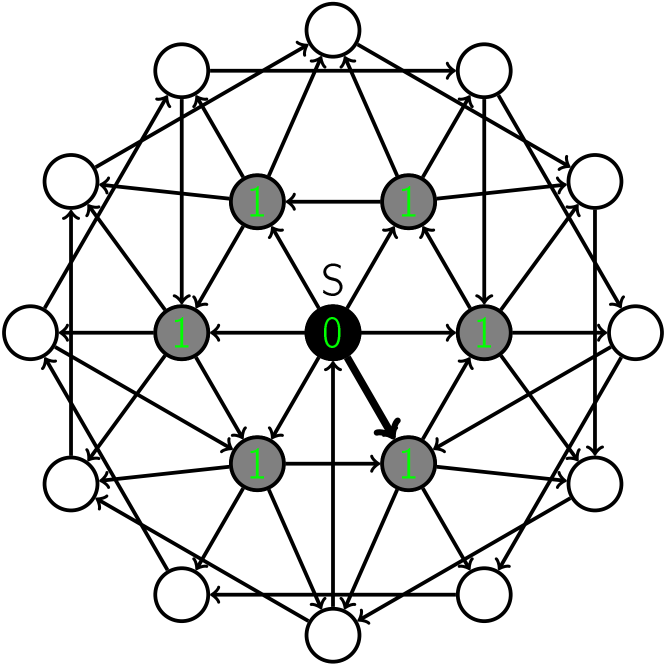

S
0
1
1

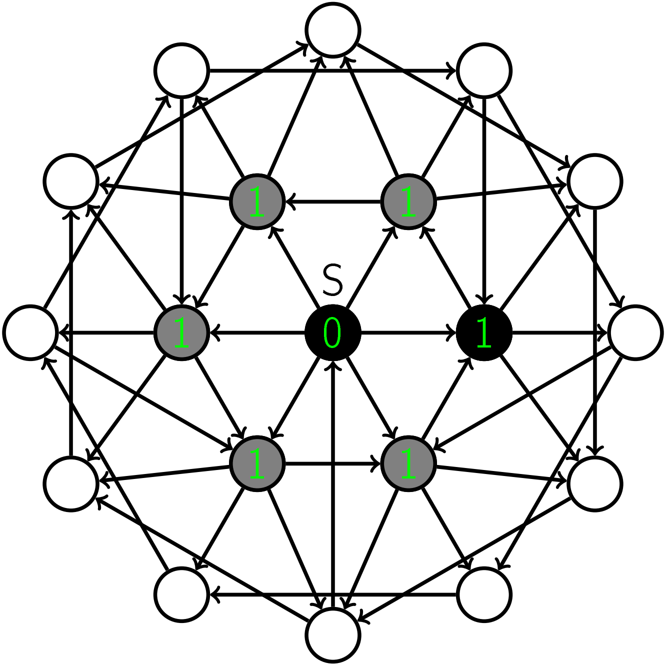


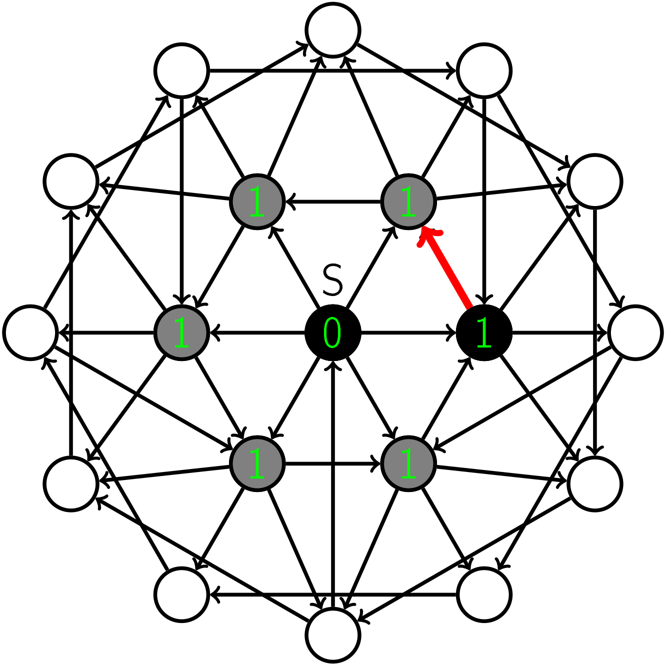
S
0
1
1
1
1


S
0
1
1
1
1

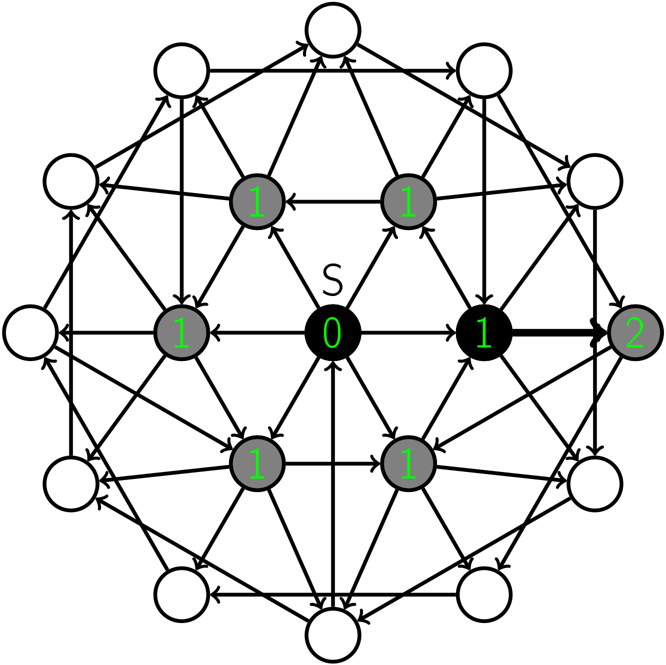

S
0
1
1
1
1
1
1

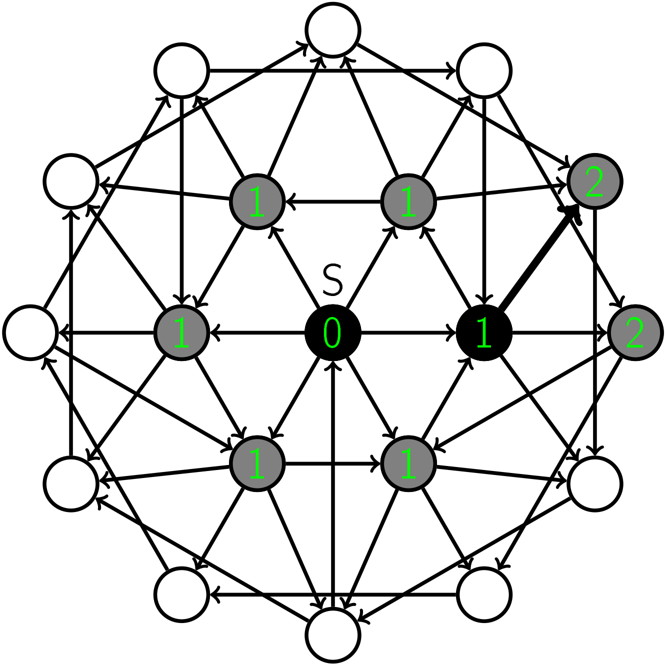

S
0
1
1
1
1
1
1


S
0
1
1
1
1
1
1
1

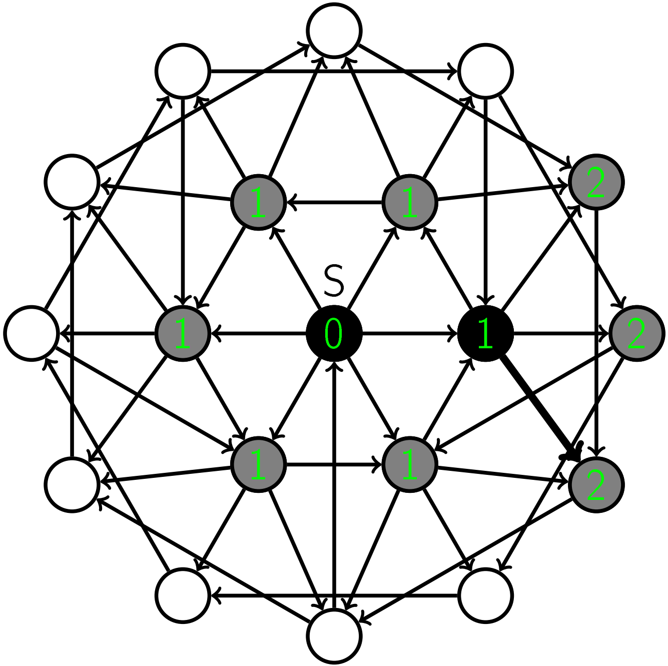


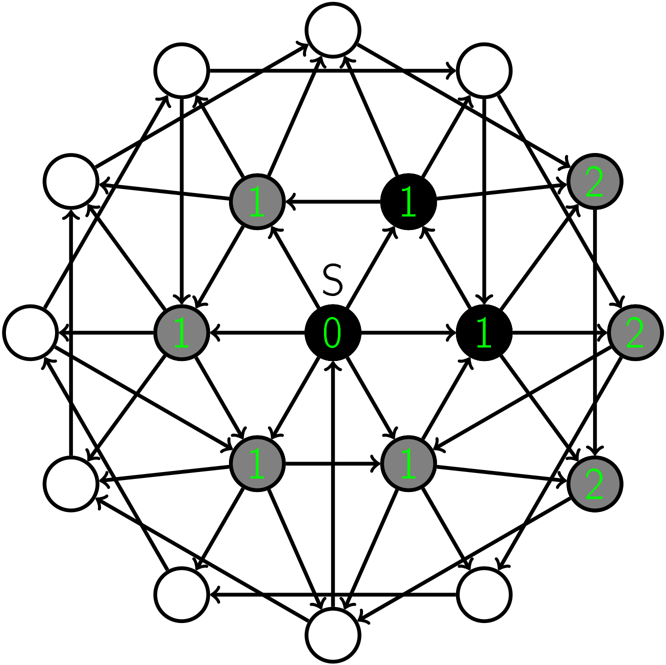
S
0
1
1
1
1
1
1


S
0
1
1
1
1
1
1
1
2

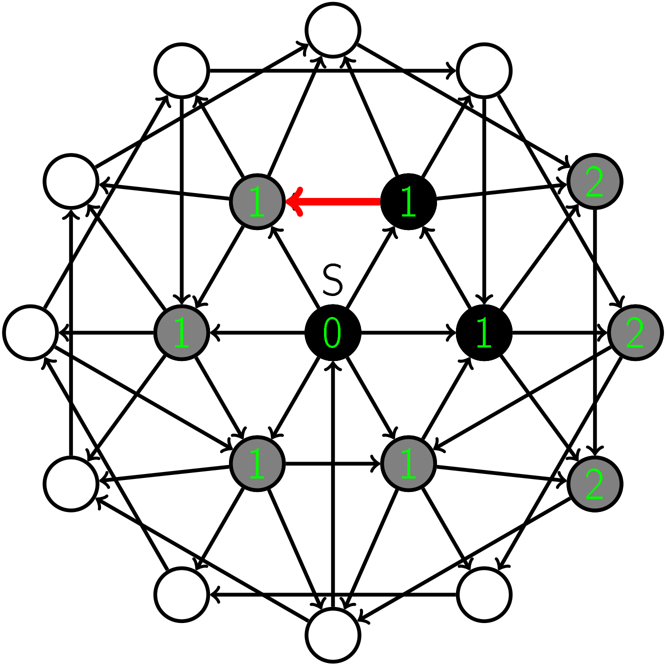


S
0
1
1
1
1
1
1
1
2
2

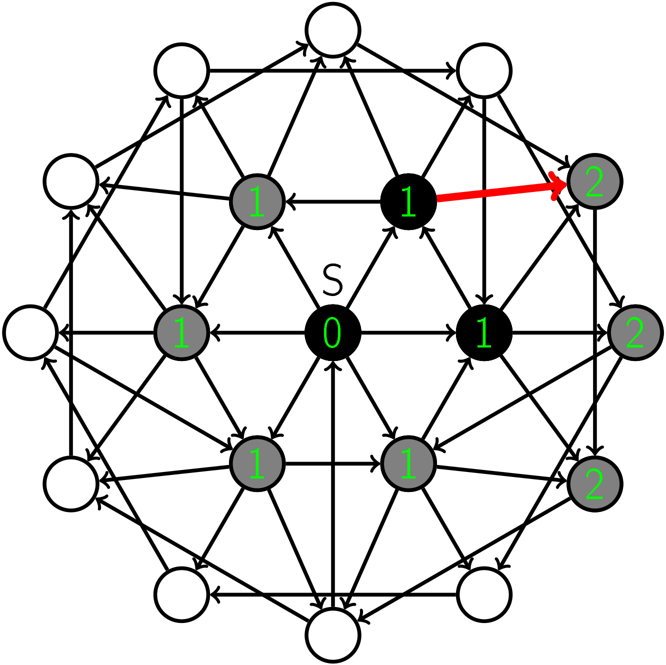

S
0
1
1
1
1
1
1
1
2
2
2

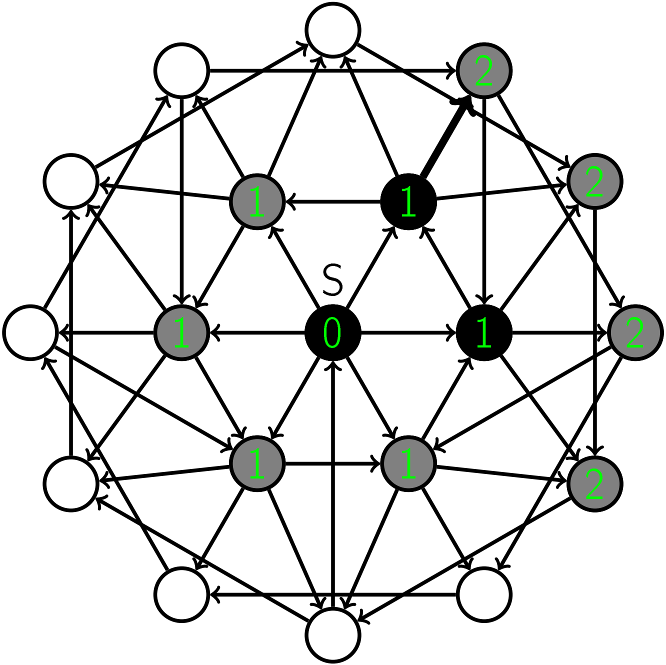


S
0
1
1
1
1
1
1
2
2
2

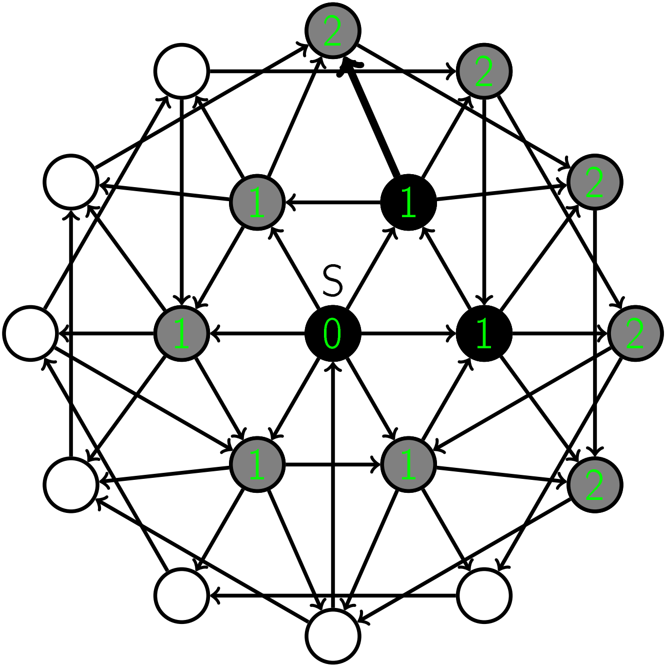


S
0
1
1
1
1
1
1
2
2
2

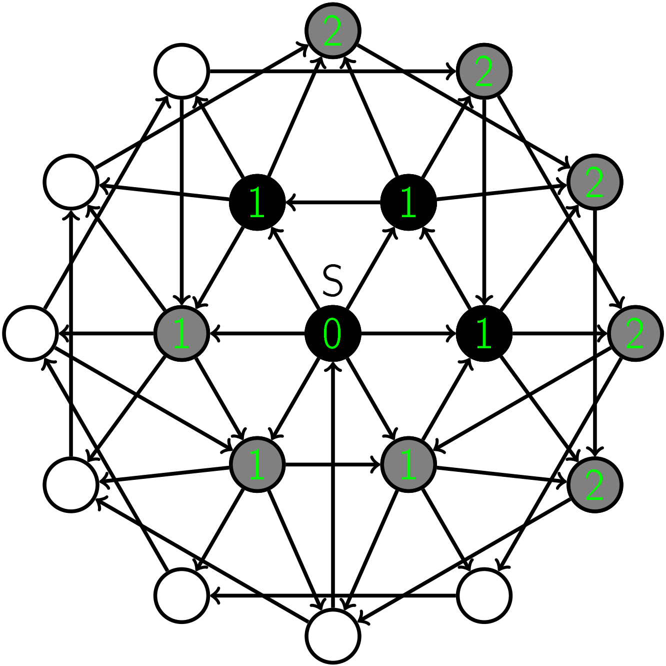

S
0
1
1
1
1
1
1
2
2
2

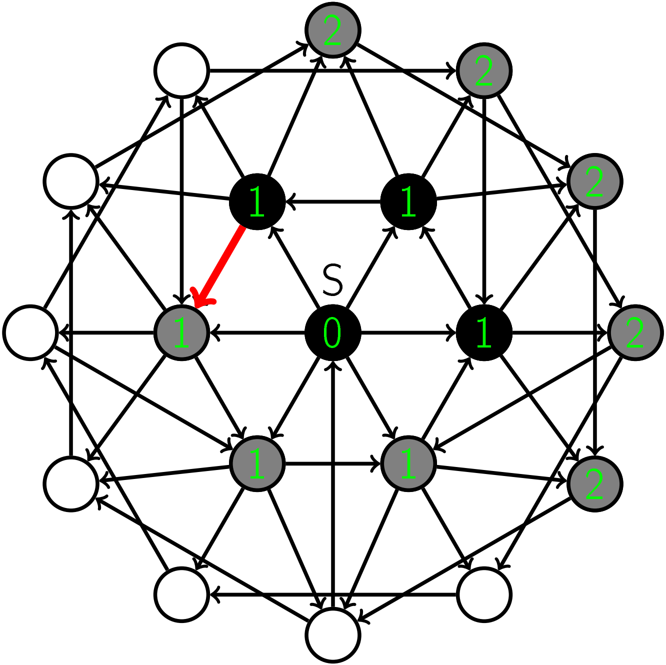


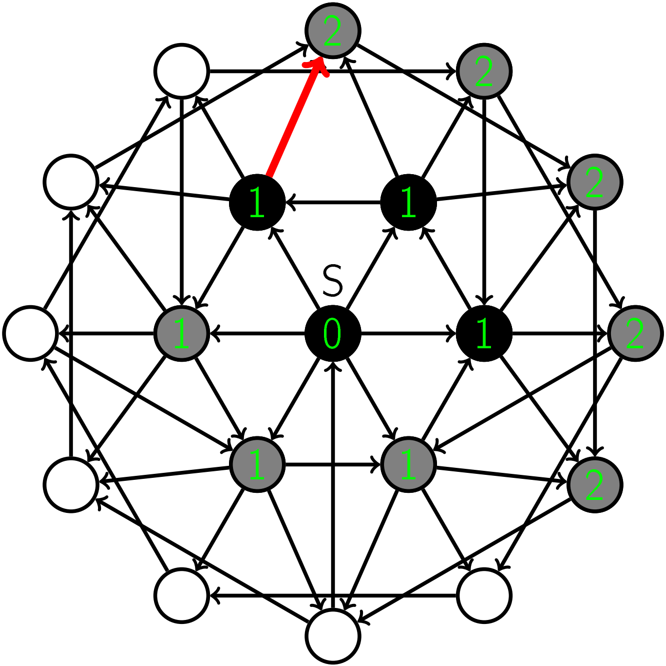
S
0
1
1
1
1
1
1
2
2
2
2



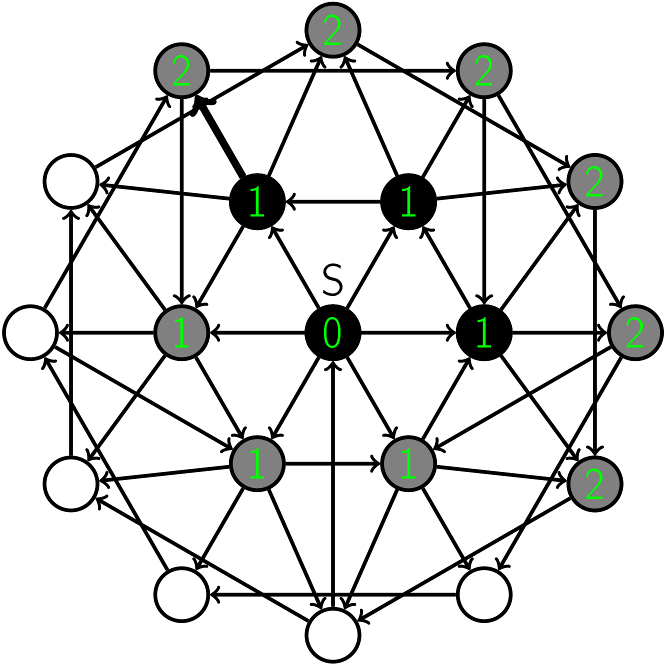
S
0
1
1
1
1
1
1
2
2
2
2
2


S
0
1
1
1
1
1
1
2
2
2
2
2

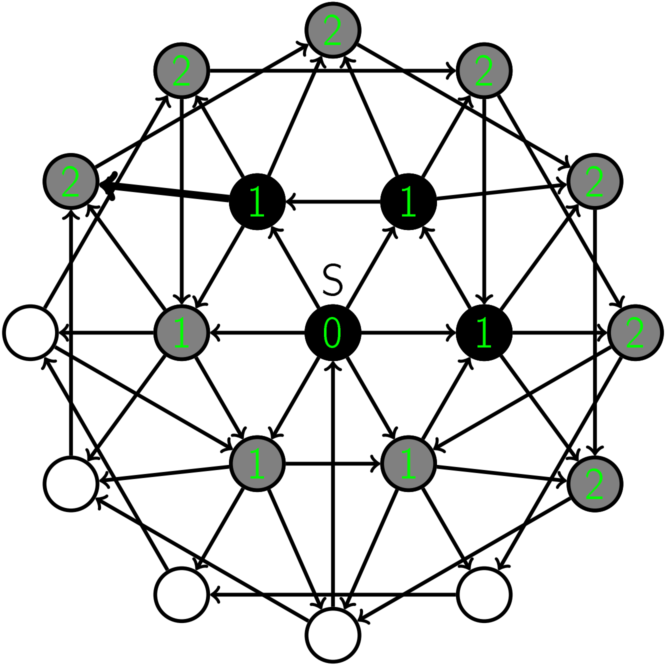

2
2
2
1
1
S
1
0
1
2
1
1
2

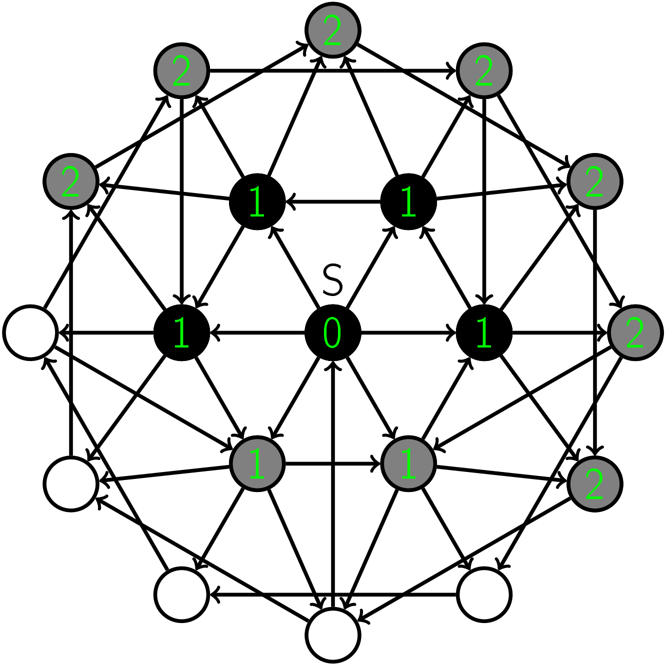


S
0
1
1
1
1
1
1
2
2
2
2
2

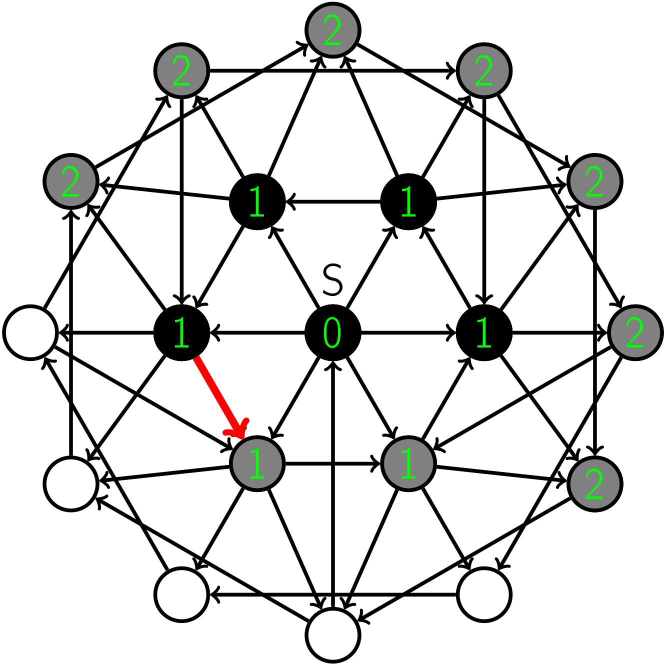


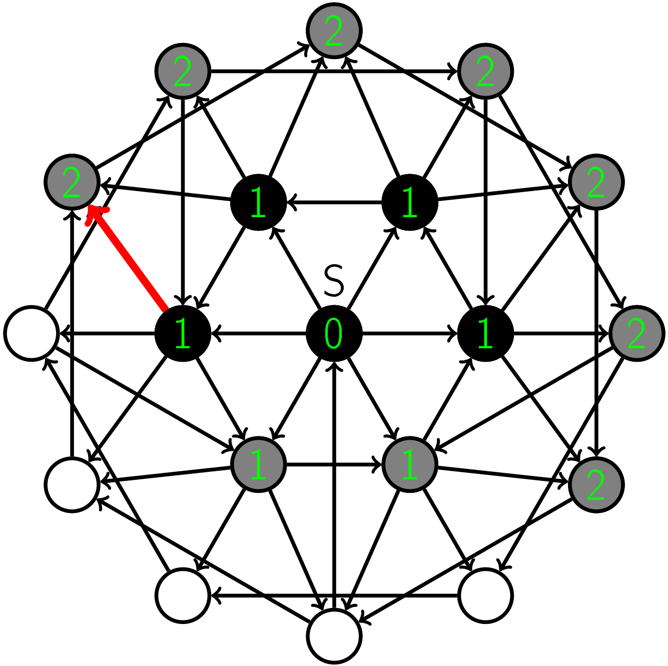
2
2
2
2
1
1
S
1
1
0
1
2
1
1
2


2
2
2
2
2
1
1
S
1
0
1
2
1
1
2

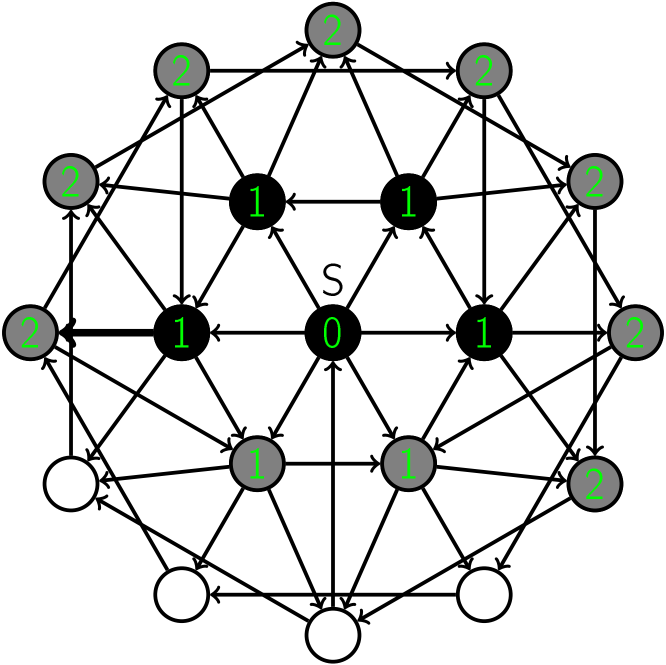


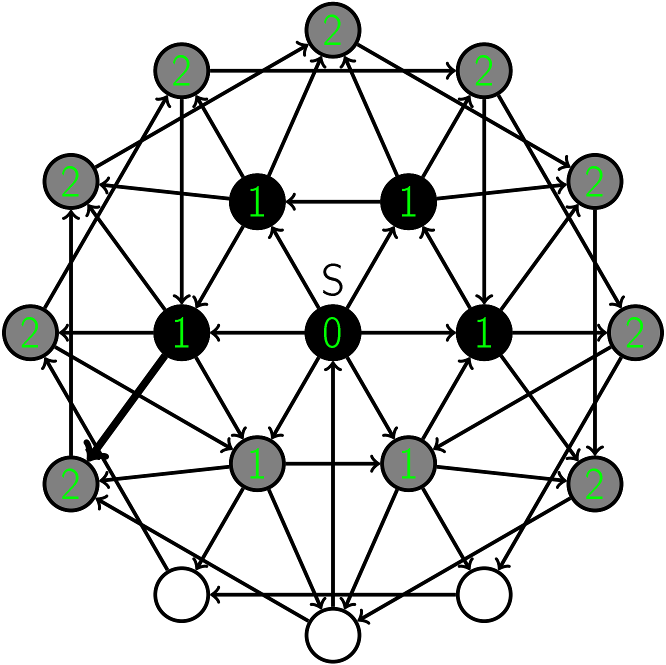
2
2
2
2
2
1
1
S
1
0
1
2
1
1
2


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2

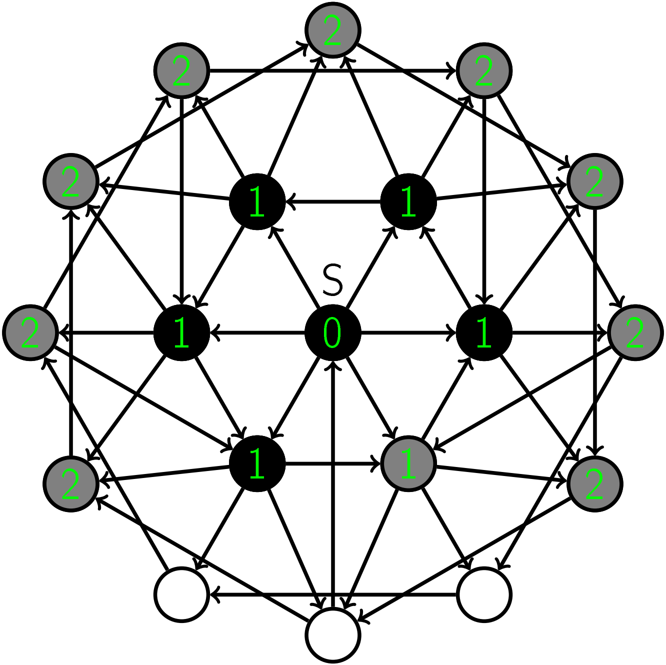


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2

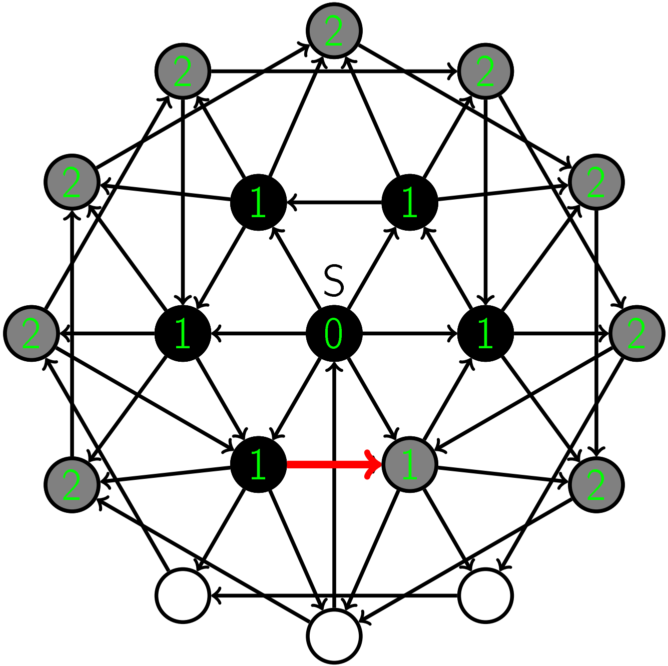

2
2
2
2
2
1
1
S
2
1
0
1
2
1
1
2



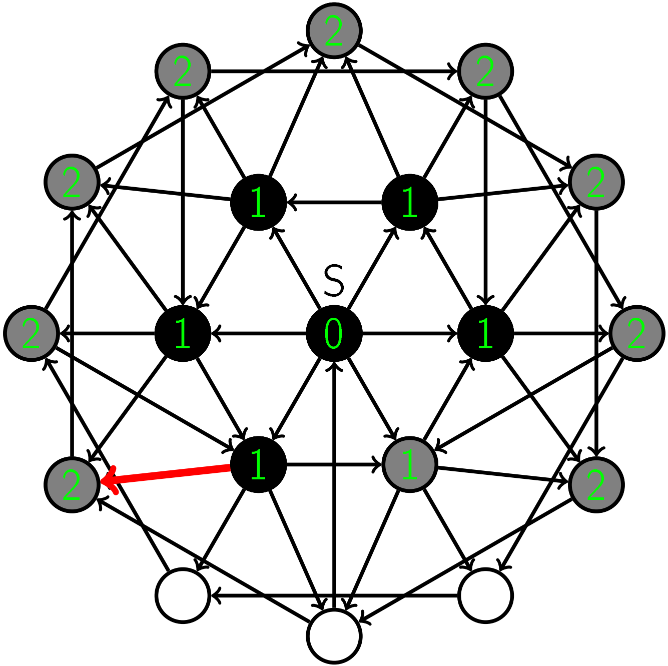
S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2

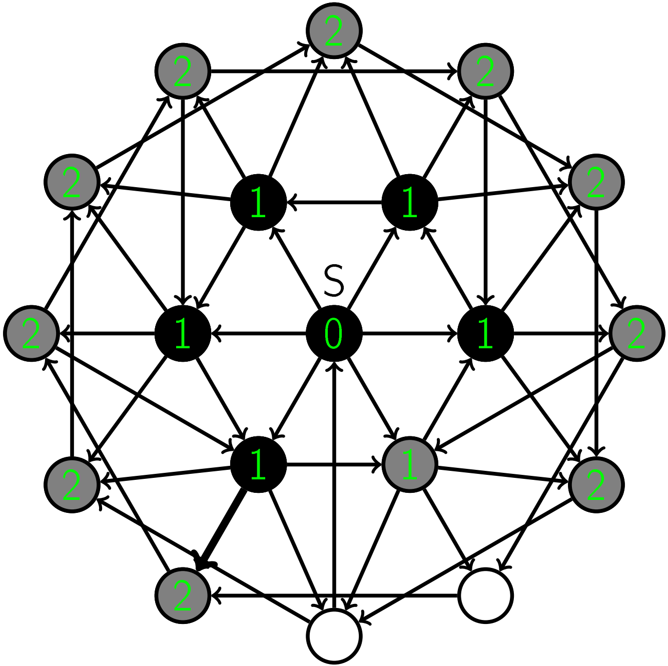


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2

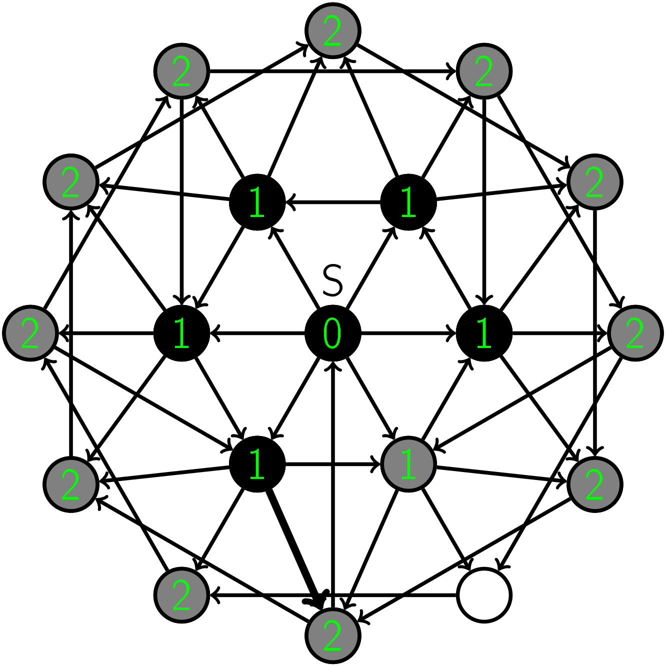


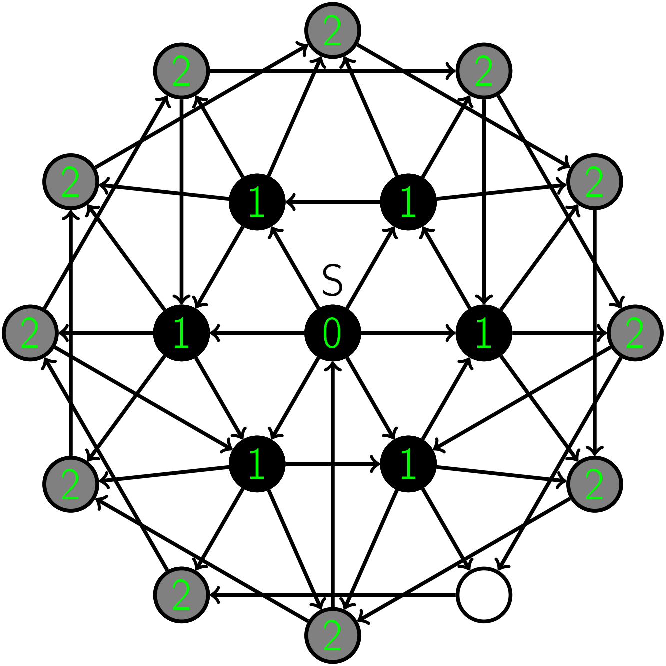
2
2
2
2
2
1
1
S
2
1
0
1
2
1
1
2
2

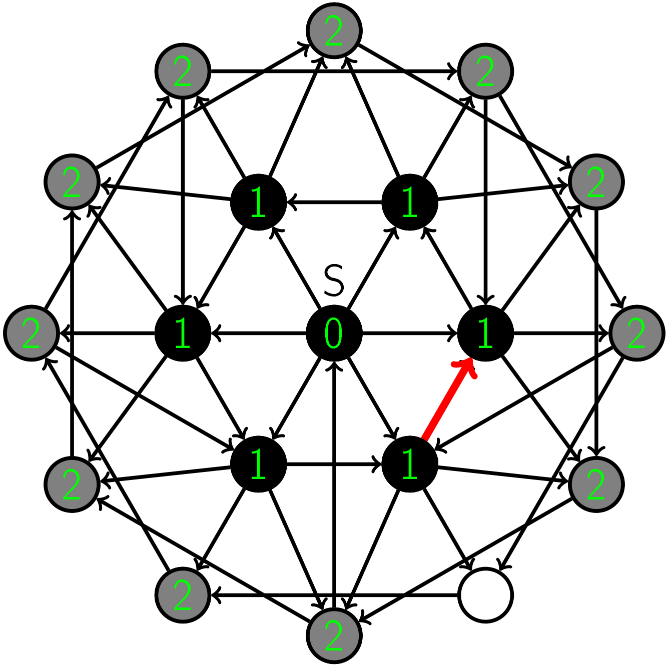

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2



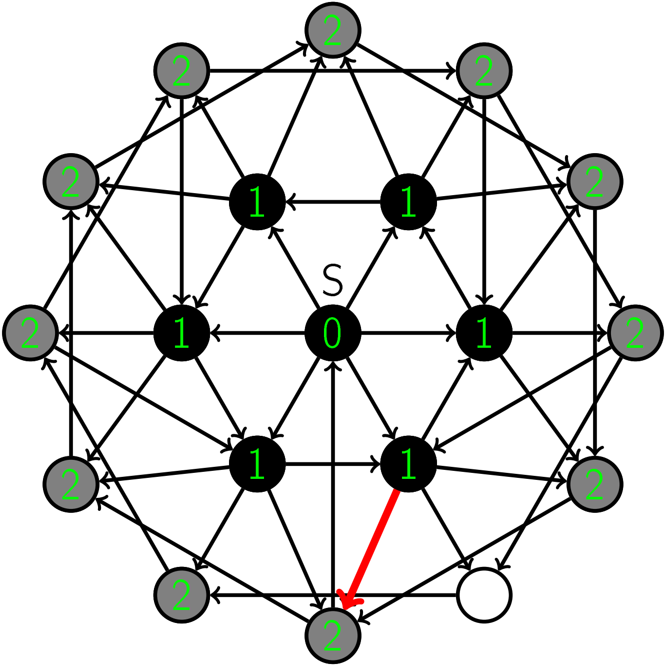
S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2

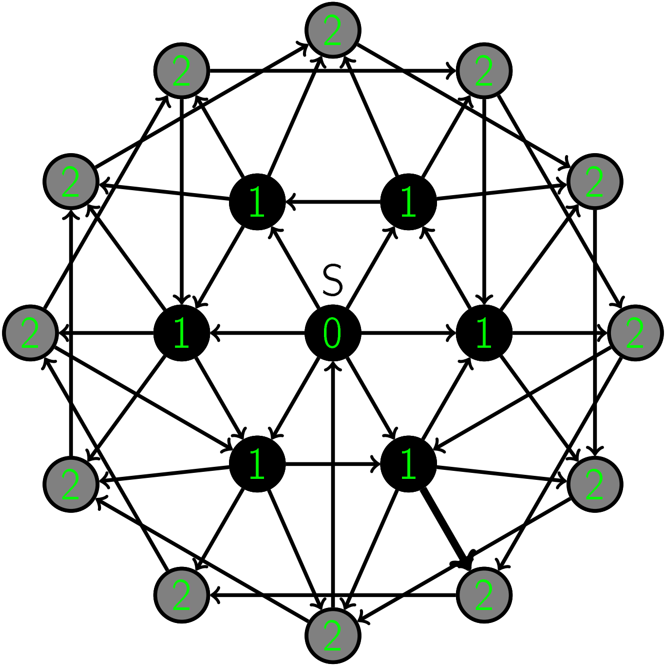

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2

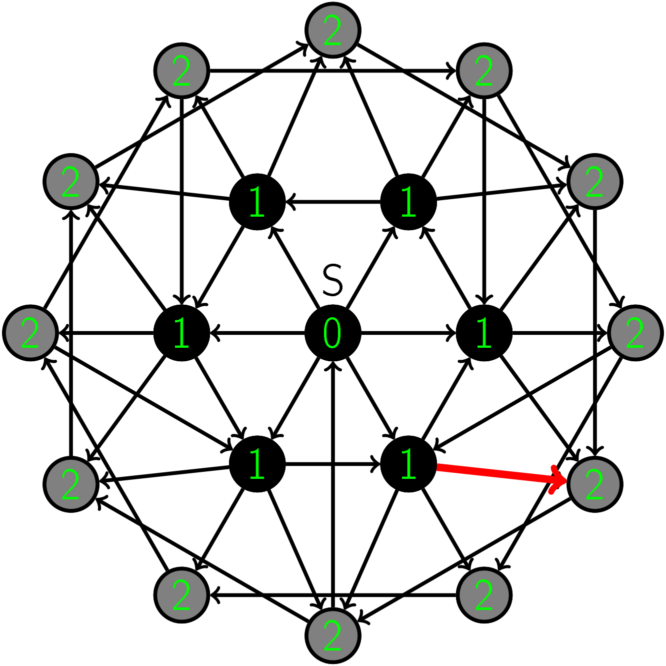


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2

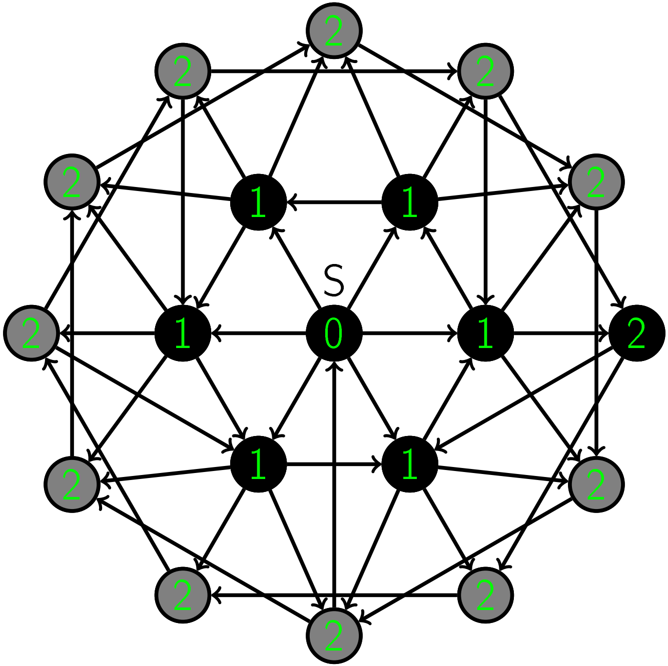

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

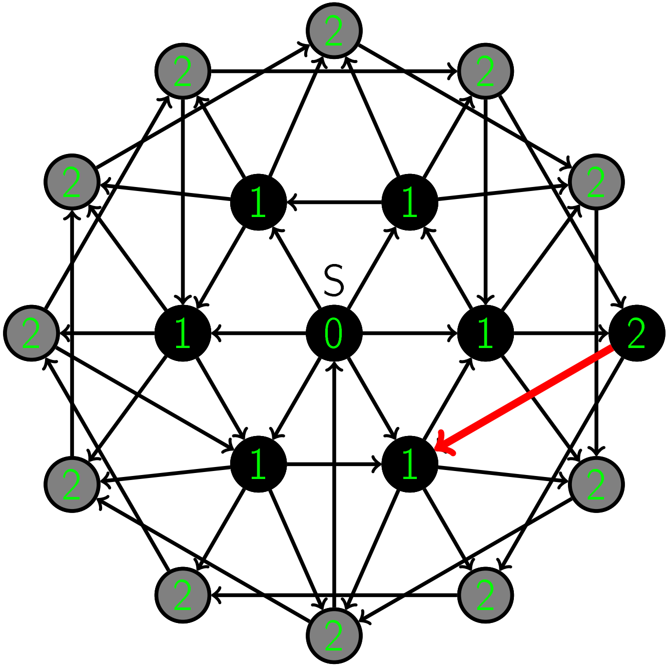


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

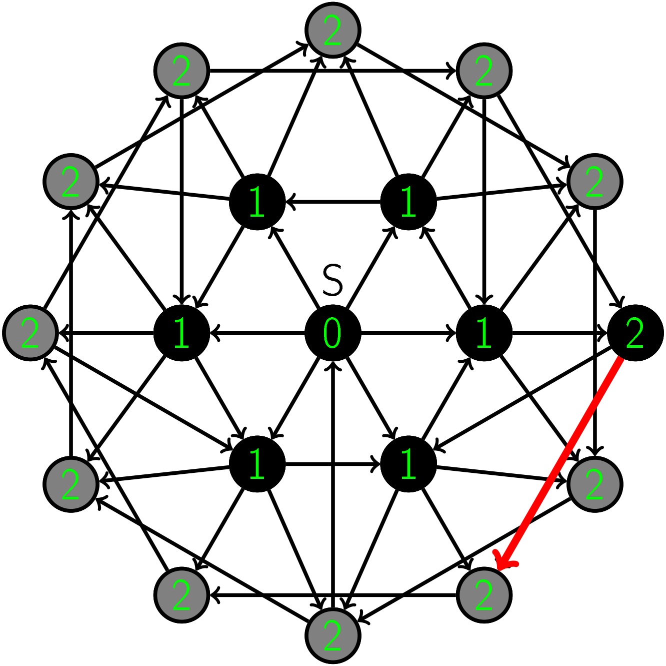


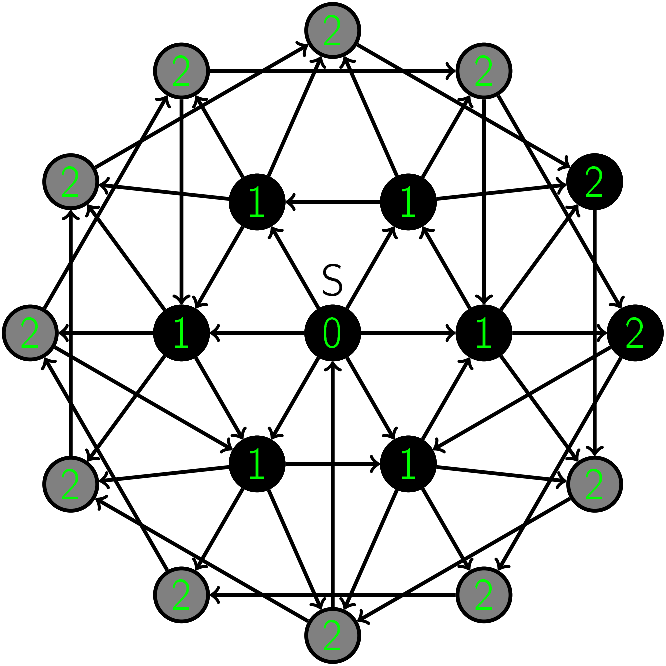
S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

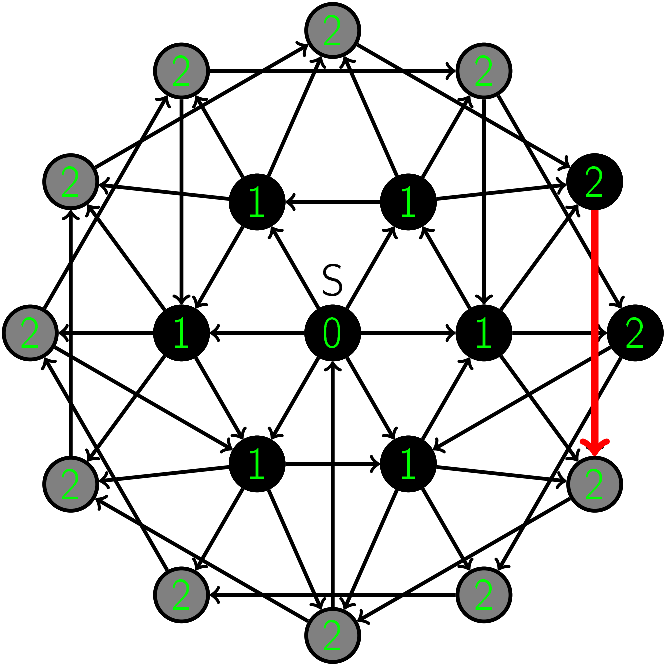

S
0
1
2

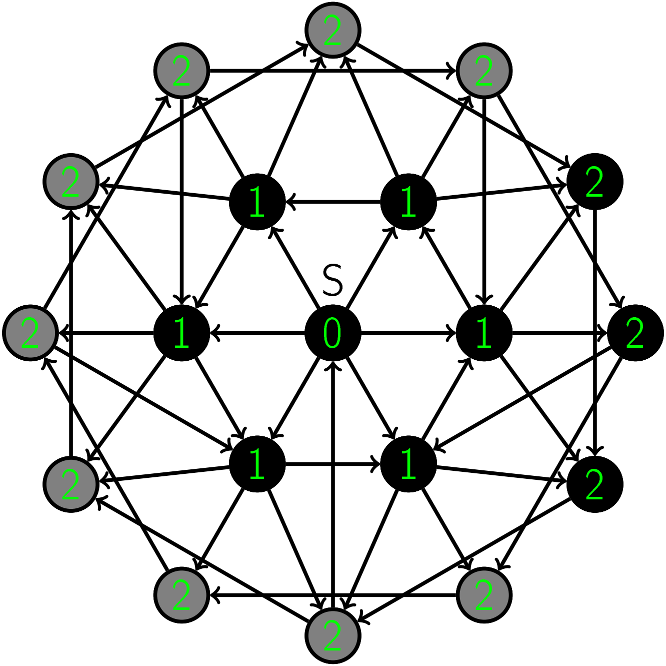


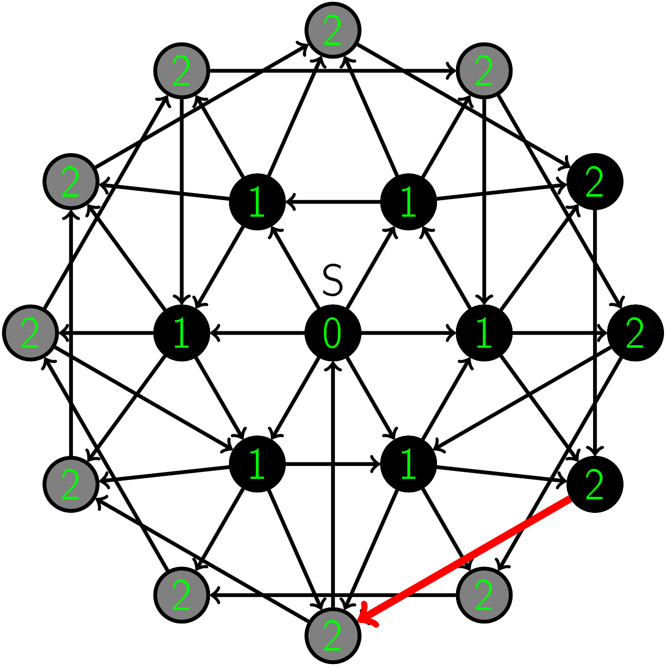
S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2
2

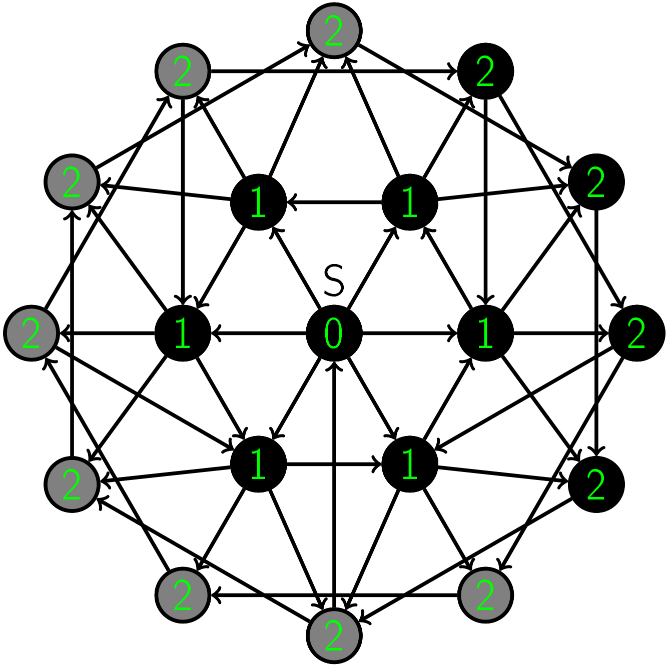

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

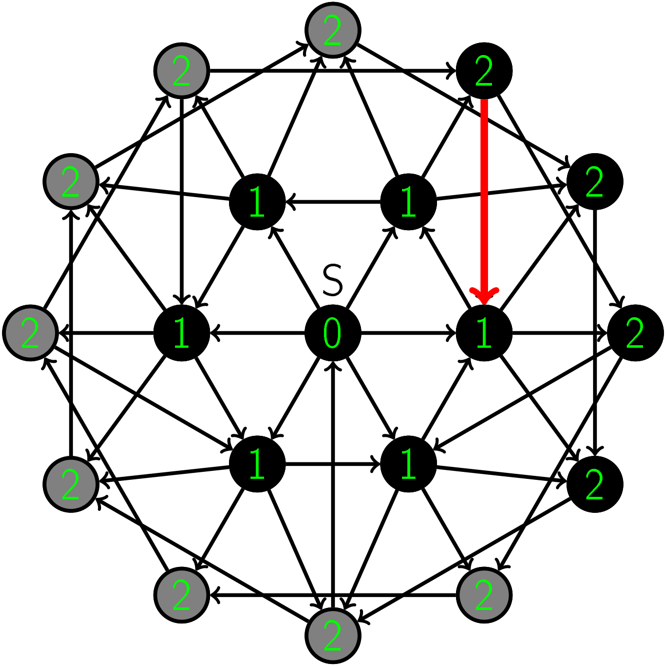

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

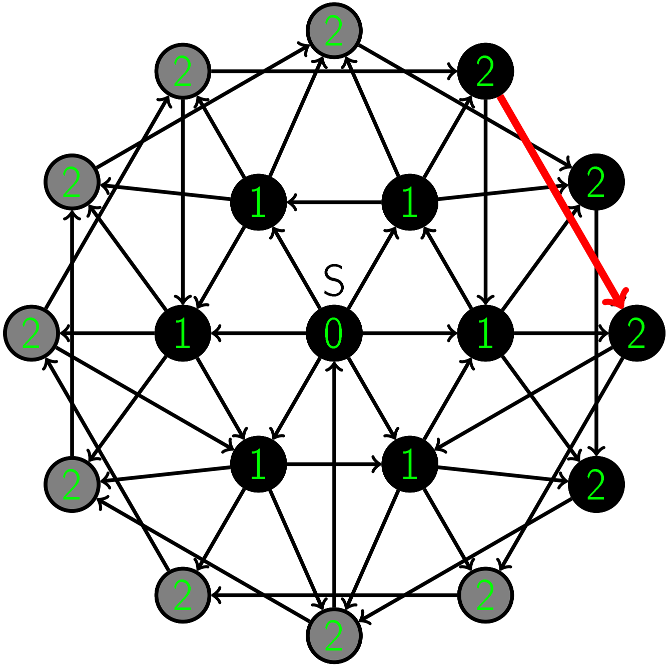

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

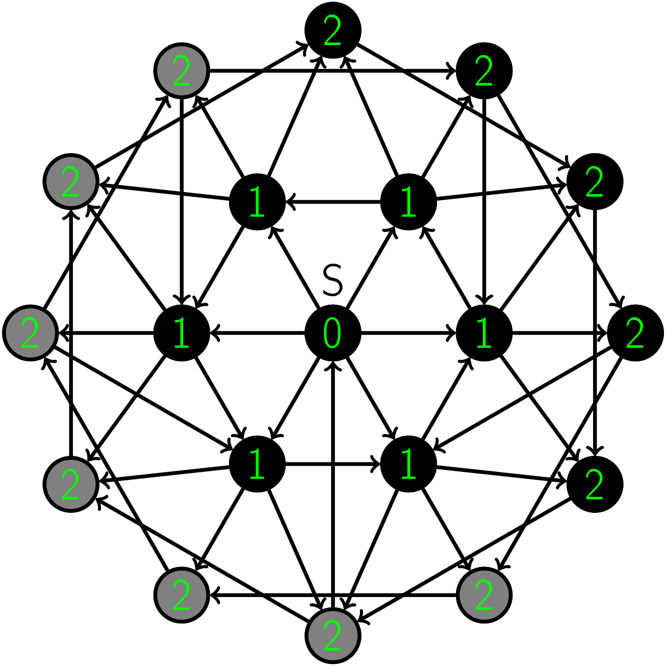


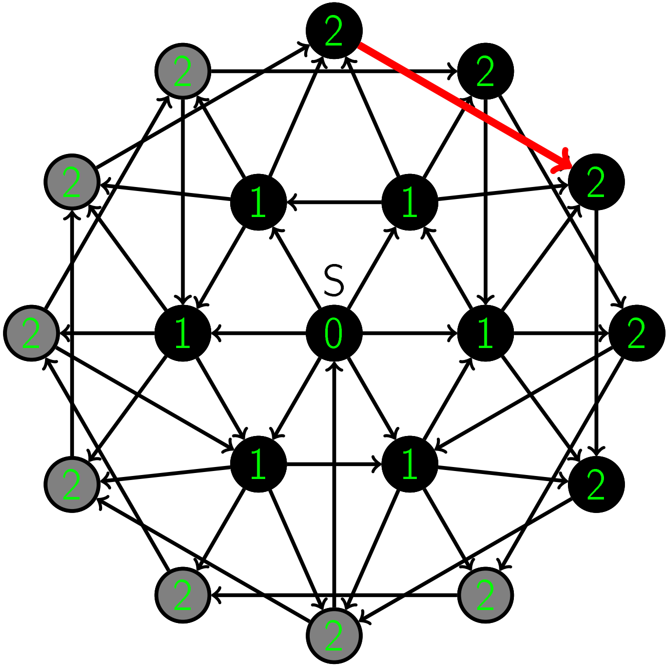
S
0
1
2

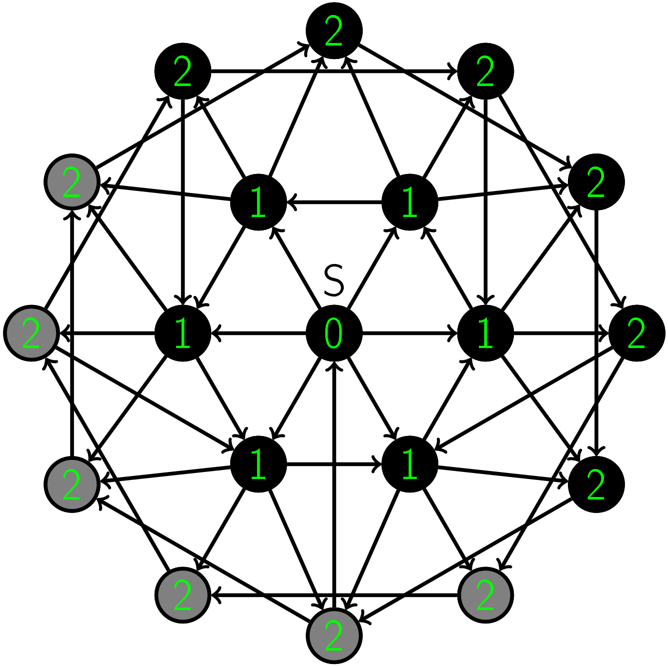

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

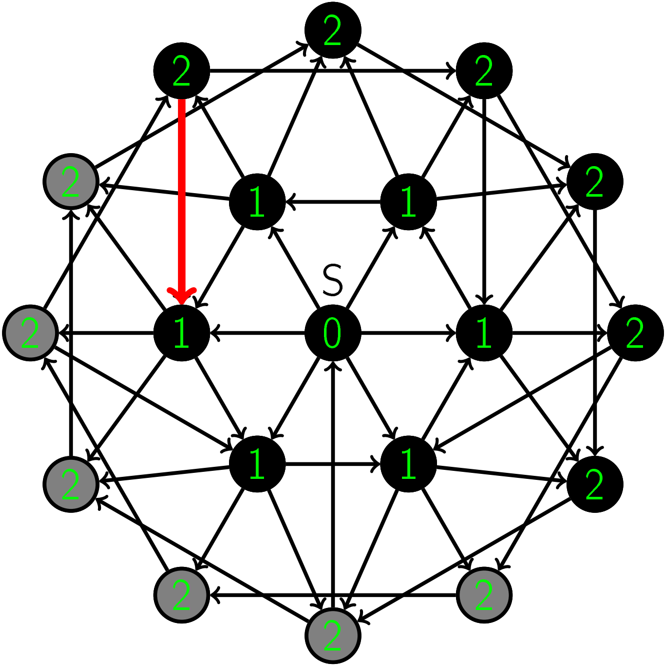

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2


S
0
1
2

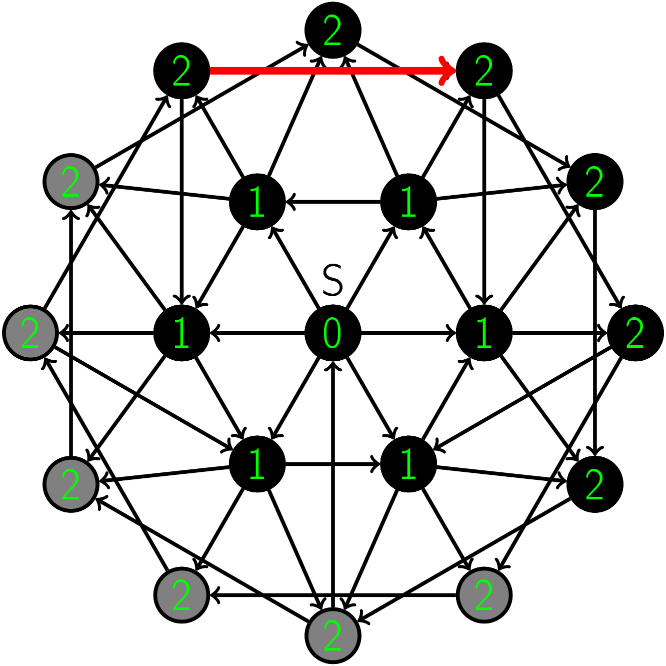

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

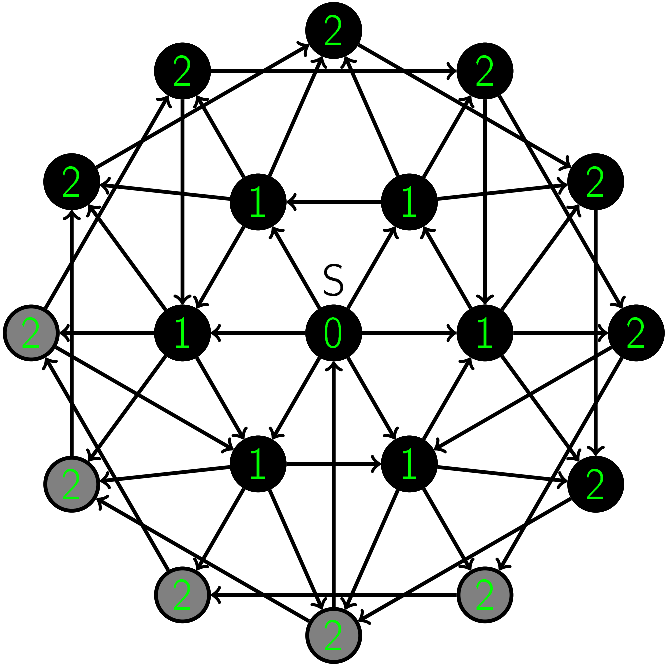


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

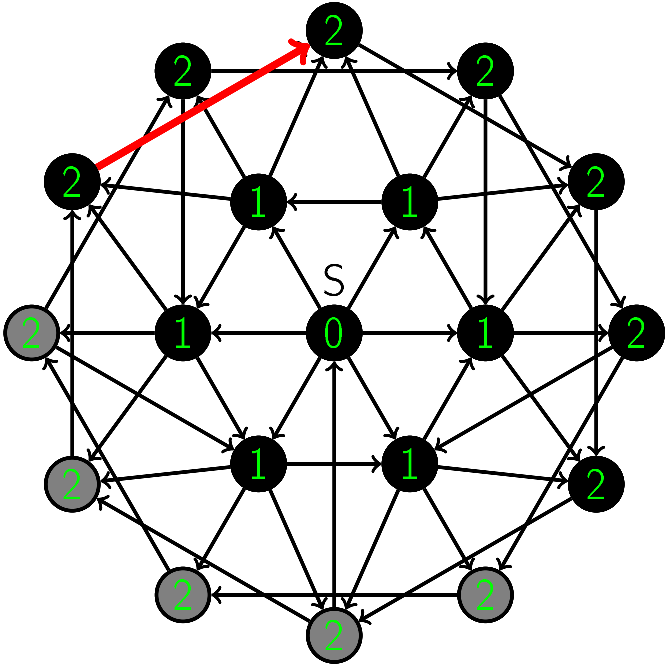


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

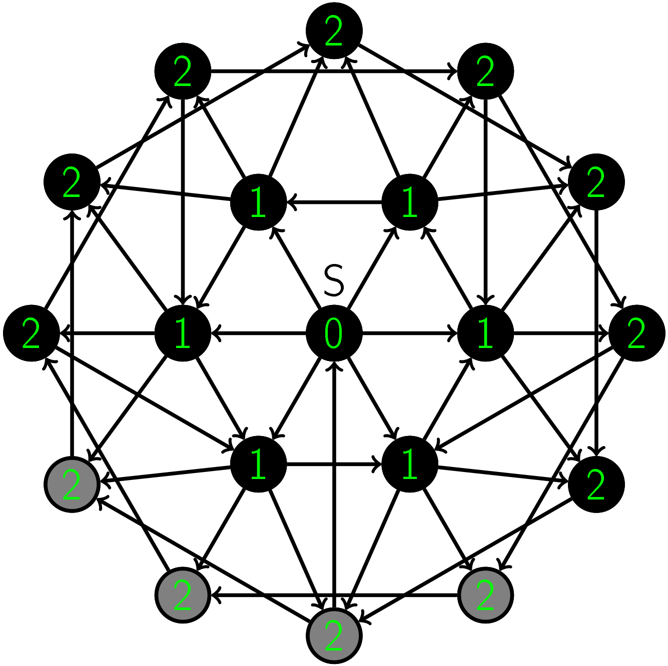

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2



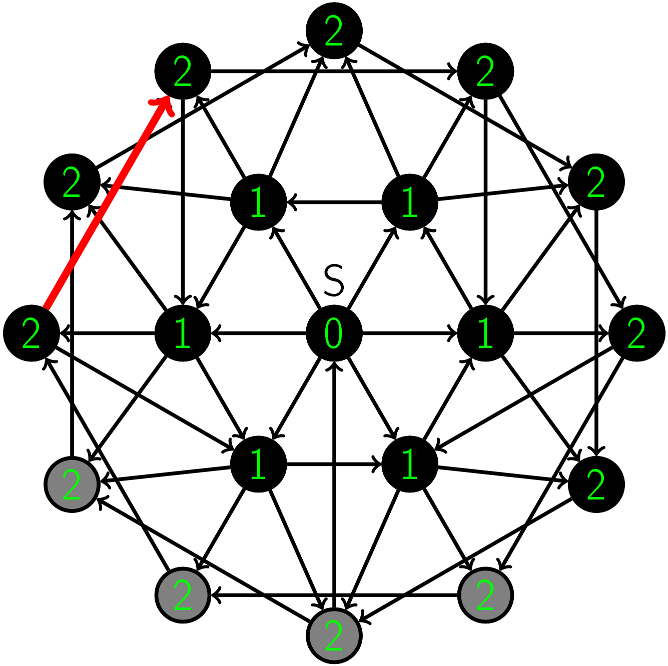
2
2
2
2
1
1
2
S
2
1
0
1
2
2
1
1
2
2
2
2

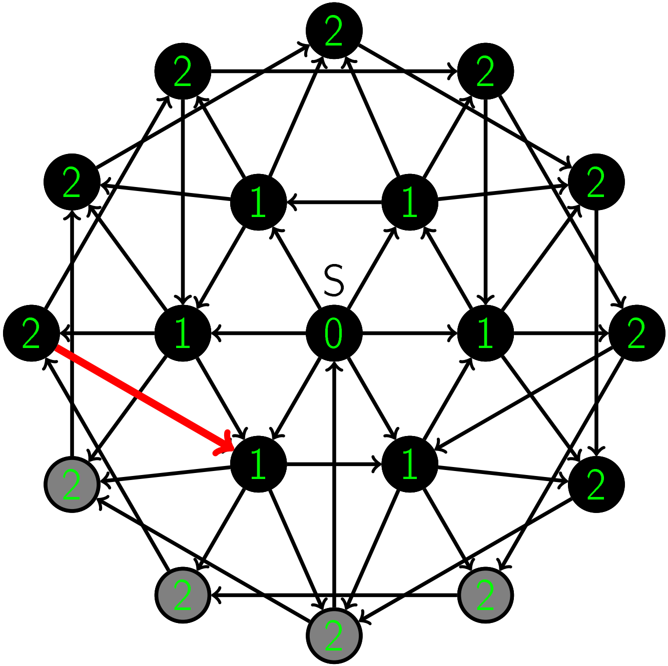


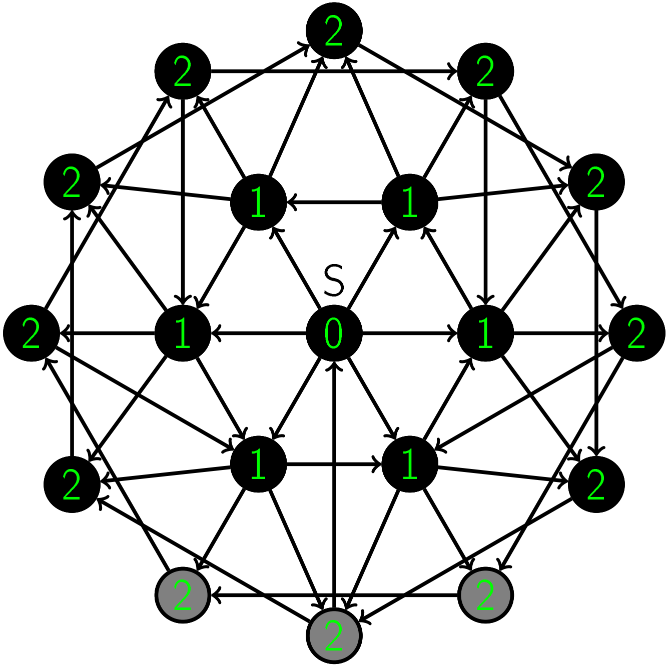
S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

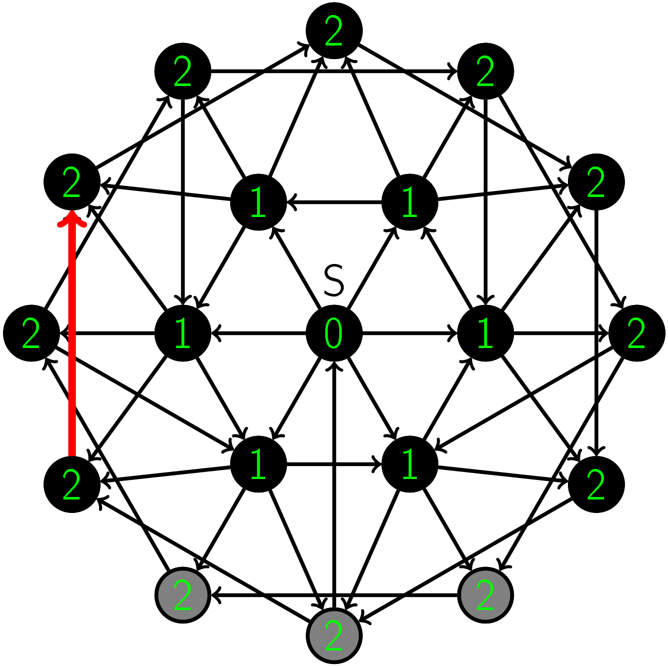


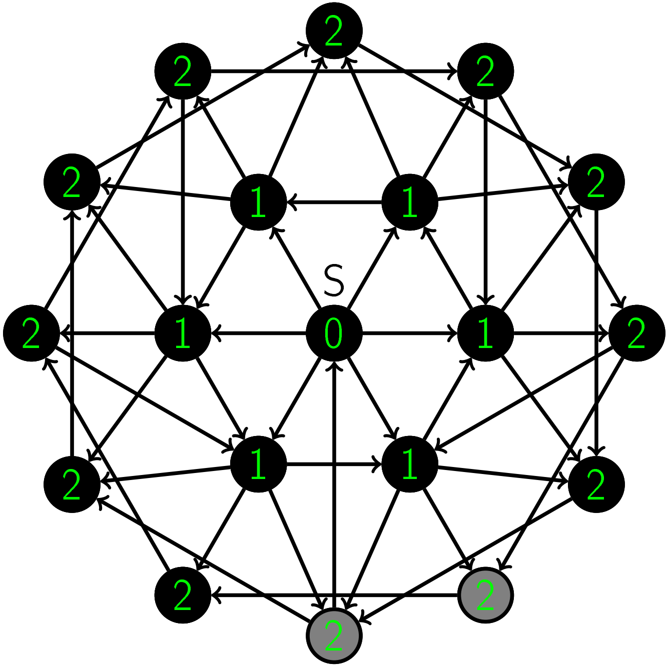
S
0
1
2

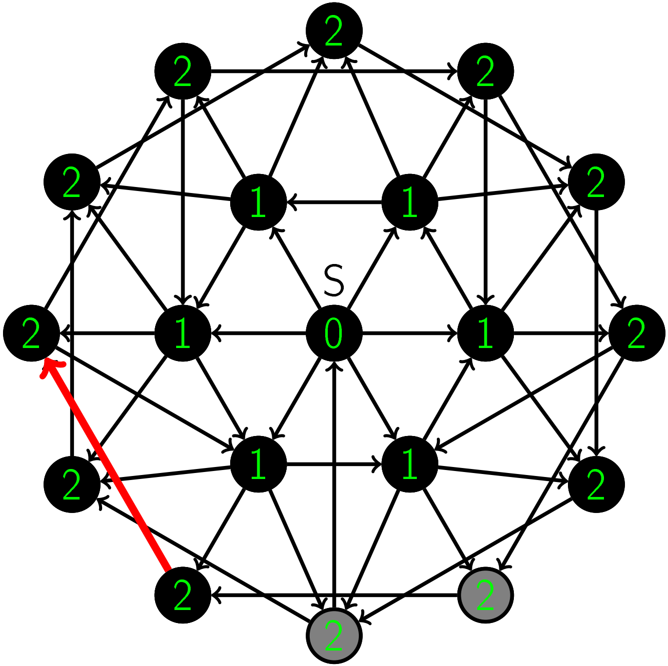

S


2
2
2
2
1
1
2
S
2
1
0
1
2
1
1
2
2
2
2
2

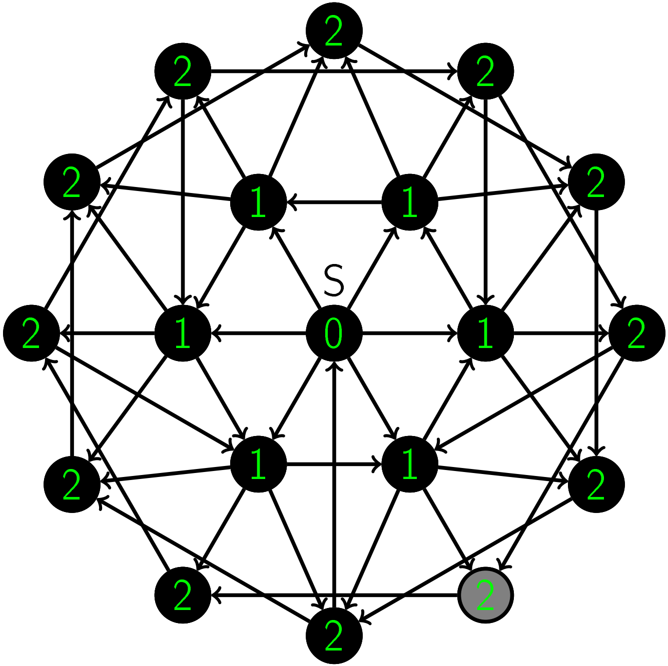

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

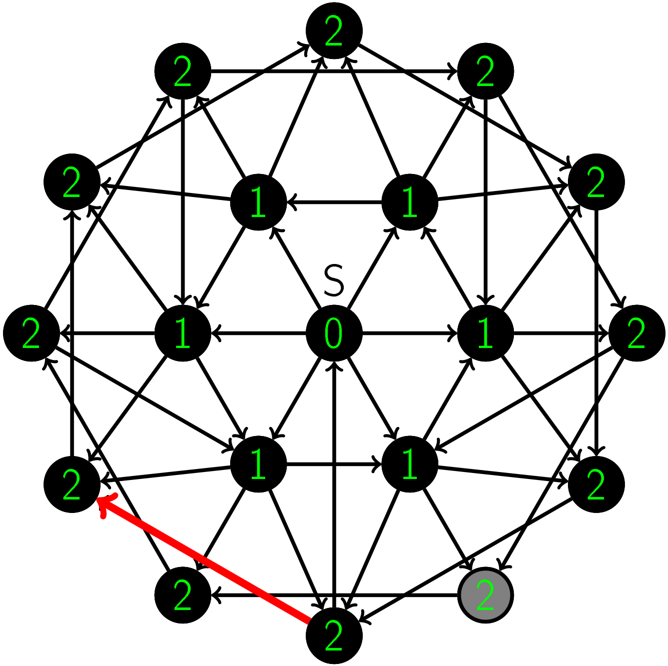

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

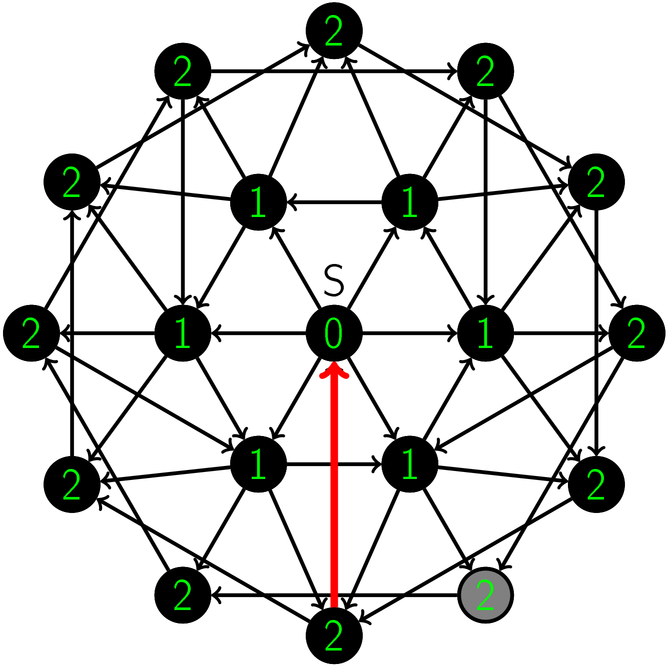

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

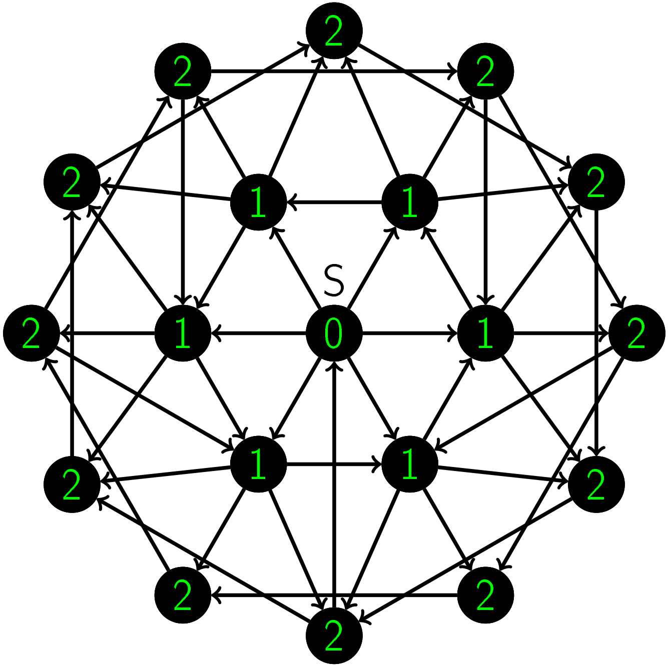


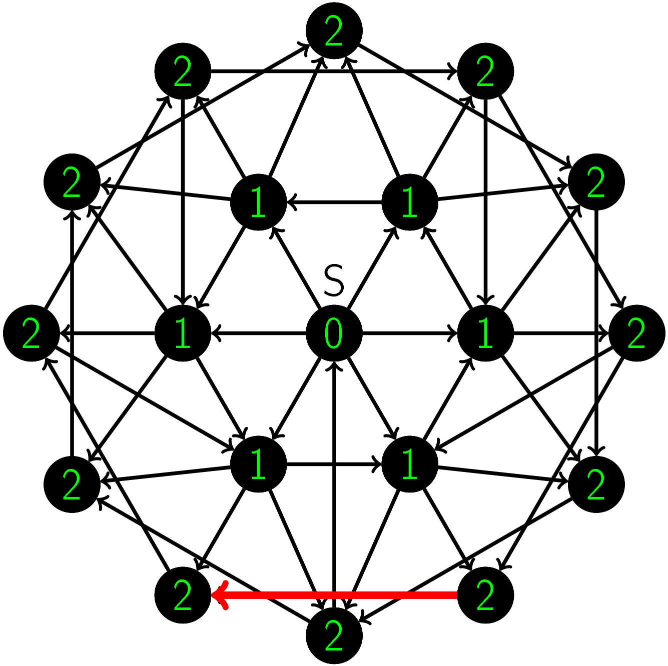
S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

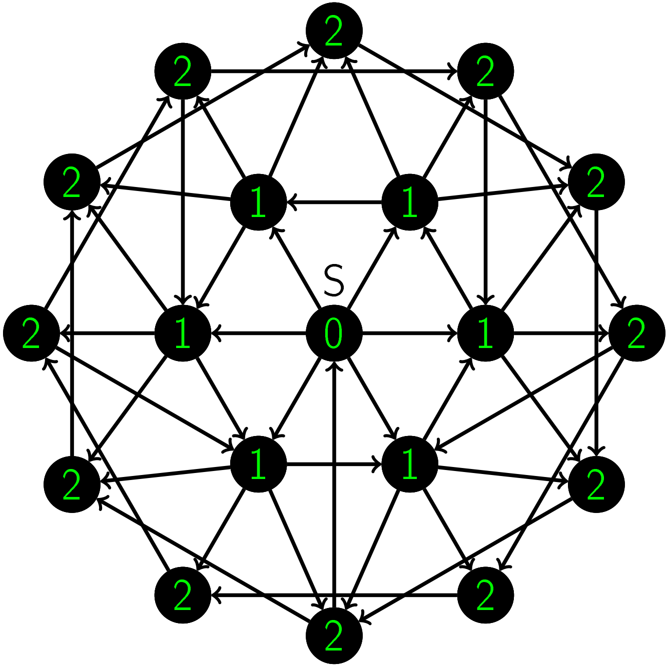

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2


S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2

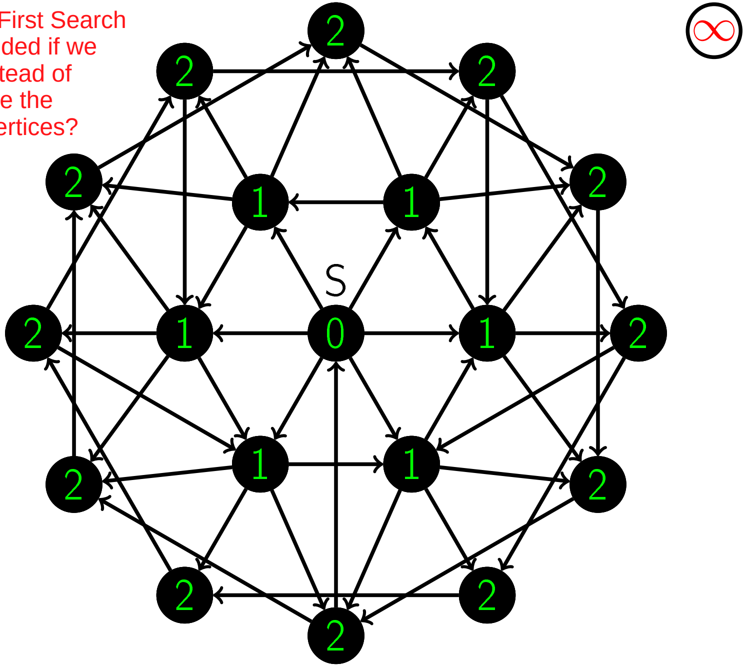

S
0
1
1
1
1
1
1
2
2
2
2
2
2
2
2
2
2
2
2



S
0
1
2

Will Breadth-First Search work as intended if we use stack instead of queue to store the discovered vertices?

Answer - NO



Outline

1. Paths and Distances
2. Breadth-first Search
3. Implementation and Analysis
4. Proof of Correctness
5. Shortest-path Tree

Breadth-first search

$\texttt{BFS}(G, S)$ G as graph and S as input node

$\texttt{for all } u \in V:$

$\quad \texttt{dist}[u] \leftarrow \infty$ All nodes has distance infinity

$\texttt{dist}[S] \leftarrow 0$ S set to distance 0

$Q \leftarrow \{S\}$ {queue containing just S} Push S to 0 hence discovered

$\texttt{while } Q \texttt{ is not empty:}$

$\quad u \leftarrow \texttt{Dequeue}(Q)$ Now dequeue and process nodes one by one

$\quad \texttt{for all } (u, v) \in E:$

$\quad\quad \texttt{if dist}[v] = \infty:$ If distance is not infinity means the node has already been discovered and nothing needs to be done

$\quad\quad\quad \texttt{Enqueue}(Q, v)$

$\quad\quad\quad \texttt{dist}[v] \leftarrow \texttt{dist}[u] + 1$ distance equal to distance of previous layer plus 1

Note that distance in infinity not available in many languages hence you can take very high number like a number greater then the max no of nodes or edges

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

External While loop is at most number of nodes |V|

- Each edge is examined either once (for directed graphs) or twice (for undirected graphs)

total no of iterations of internal for loop is atmost |E|

$\square$

Outline

Reachability

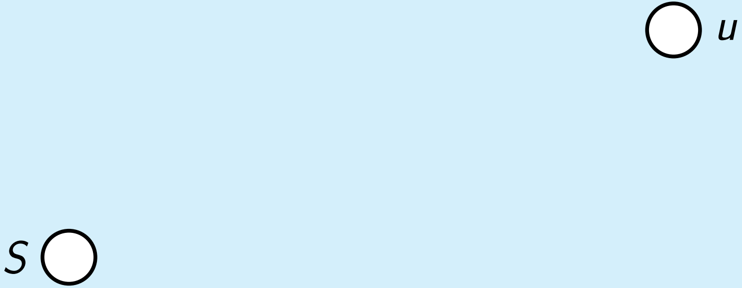
Definition

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

Lemma

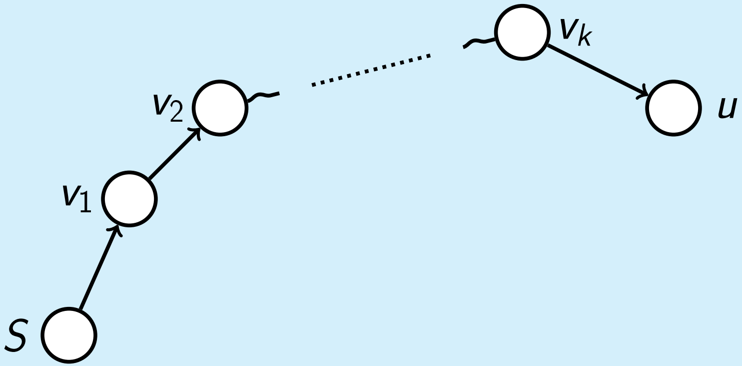
Reachable nodes are discovered at some point, so they get a finite distance estimate from the source. Unreachable nodes are not discovered at any point, and the distance to them stays infinite.

Proof



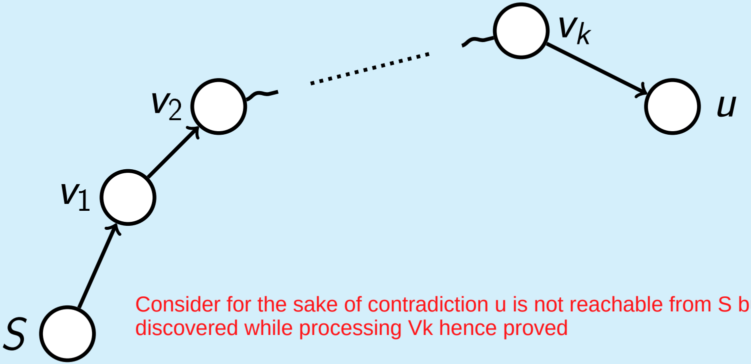
- u — reachable undiscovered closest to S

Proof



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

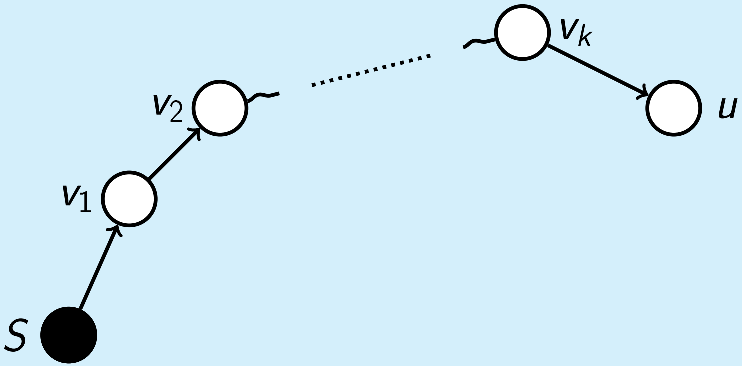
Proof



Consider for the sake of contradiction u is not reachable from S but u gets discovered while processing Vk hence proved

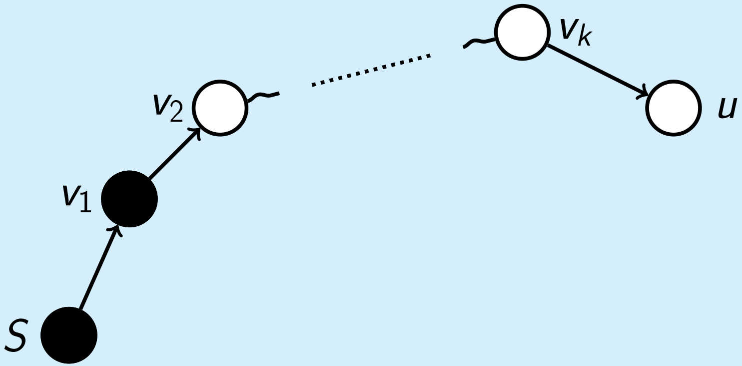
- u — reachable undiscovered closest to S
- $S - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



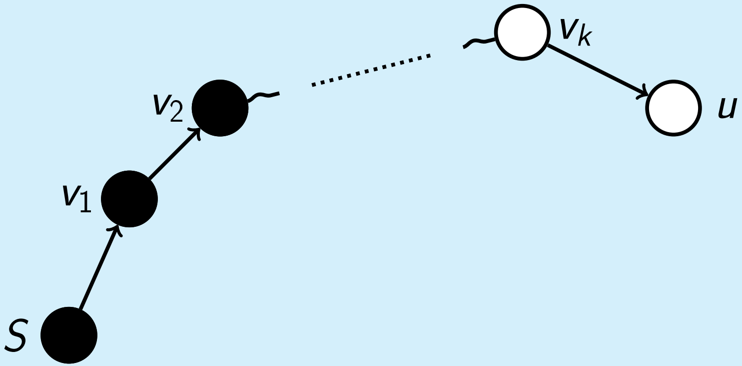
- u — reachable undiscovered closest to S
- $S - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



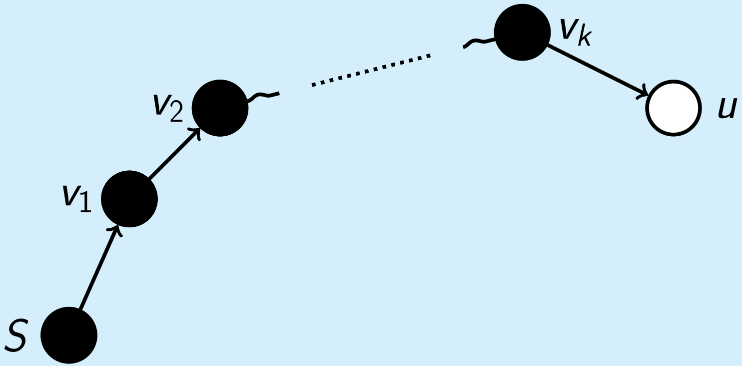
- u — reachable undiscovered closest to S
- $S - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



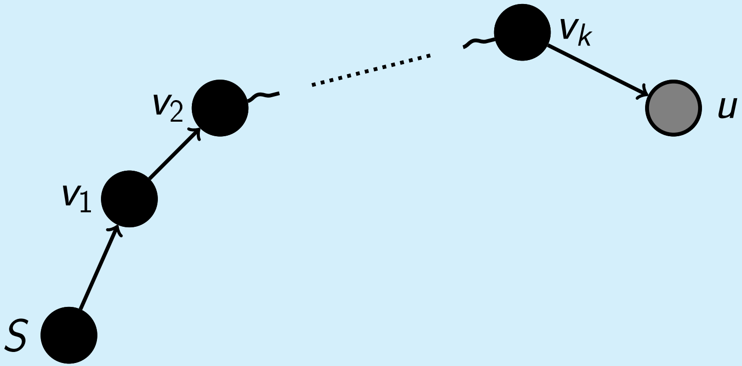
- u — reachable undiscovered closest to S
- $S - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



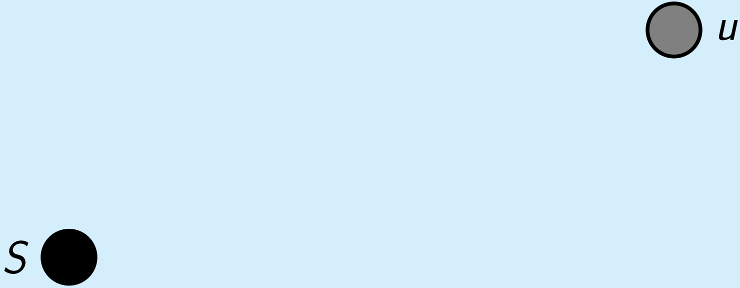
- u — reachable undiscovered closest to S
- $S - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



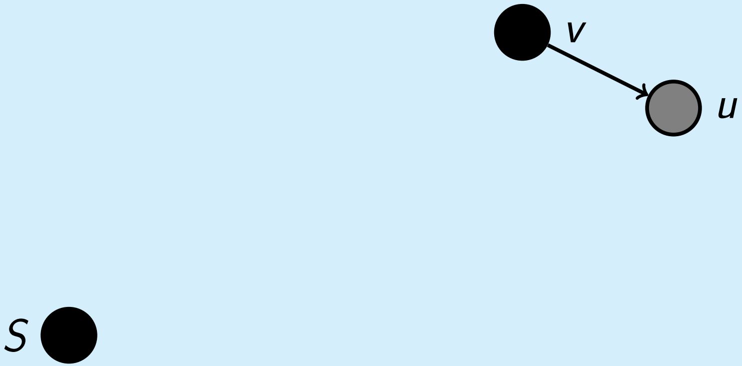
- u — reachable undiscovered closest to S
- $S - v_1 - \cdots - v_k - u$ — shortest path
- u is discovered while processing v_k

Proof



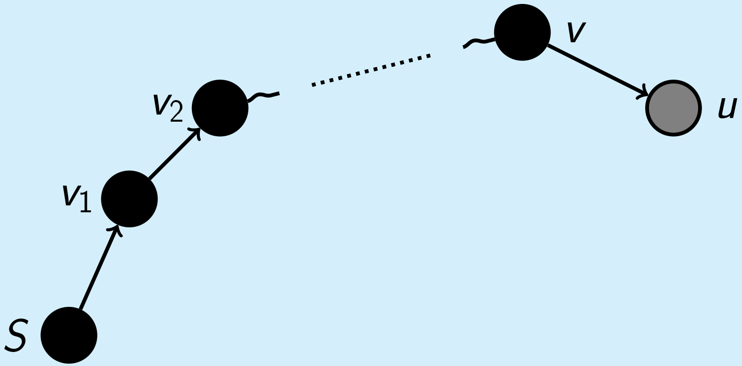
- u — first unreachable discovered

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 Contradiction □

Order Lemma

Lemma

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

This is very understandable no need to break head with the proof

Order Lemma Proof

Consider the first time the order was broken

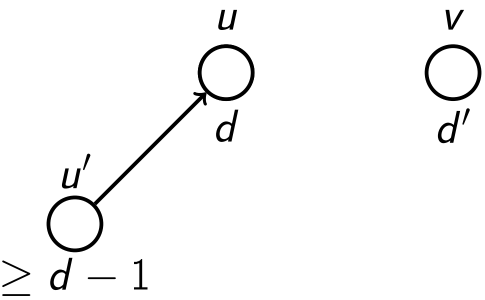
Order Lemma Proof

Lets consider by contradiction again that u is discovered and processed while v is not yet discovered

Consider the first time the order was broken

$d' \leq d$

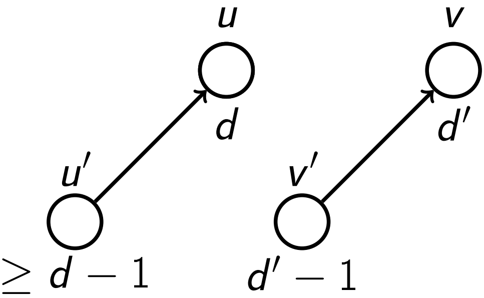
Order Lemma Proof



Consider the first time the order was broken

$d' \leq d$

Order Lemma Proof

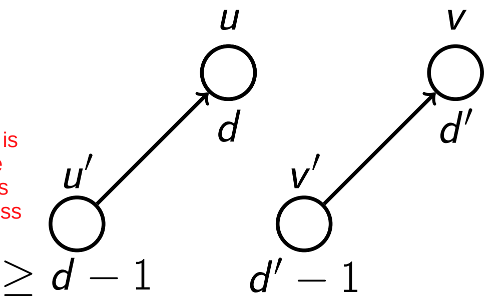


Consider the first time the order was broken

$d' \leq d$

Order Lemma Proof

For u` the distance is >= d-1 because the distance to previous node can be one less or equal to d

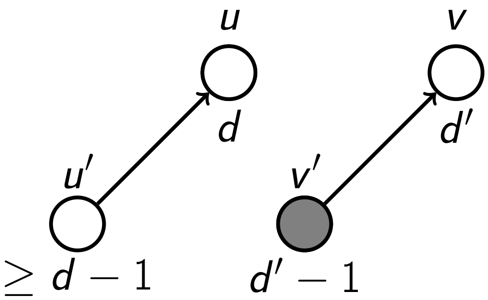


distance to v` is d`-1 because distance to v is shortest path hence distance to v` cannot be greater than d`-1 because if that is the case then it is not on the shortest path and if it is smaller then d`-1 then the distance to v will be less then d` and hence it is not the shortest path

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

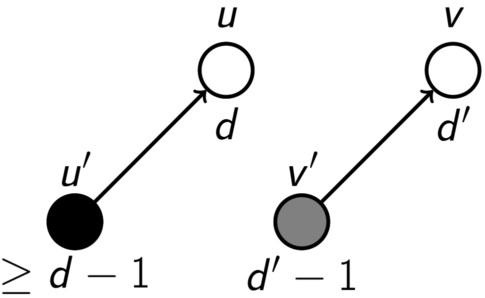
Order Lemma Proof



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

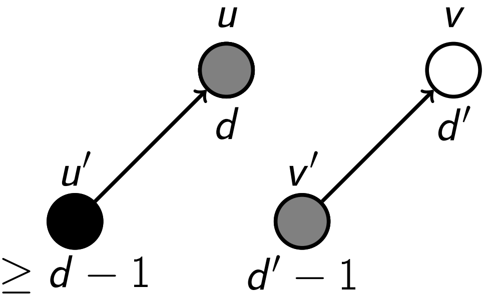
Order Lemma Proof



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

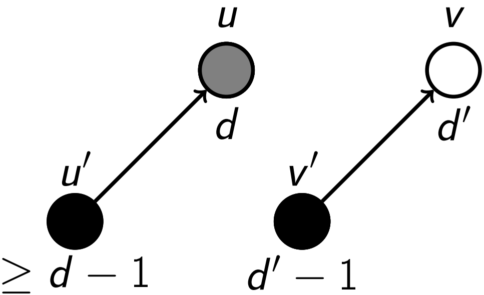
Order Lemma Proof



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

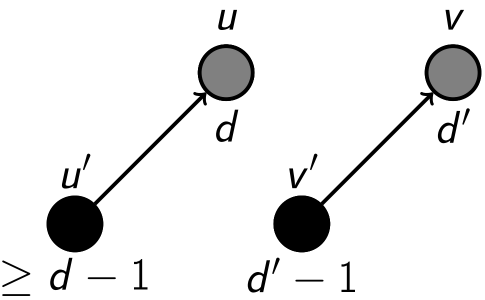
Order Lemma Proof



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

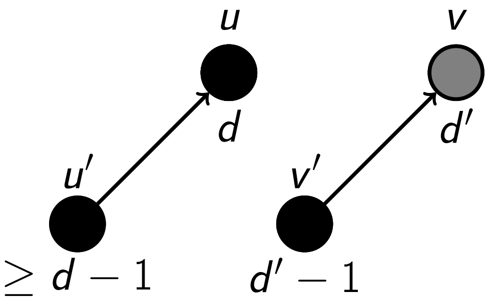
Order Lemma Proof



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

Correct distances

Lemma

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

Correct distances

Proof

- Use mathematical induction

Correct distances

Proof

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

Correct distances

Proof

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

Correct distances

Proof

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

Correct distances

Proof

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

Correct distances

Proof

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

Correct distances

Proof

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

Correct distances

Proof

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

Queue property

Queue:	d	d	d	$\dots$	d	d	$d+1$	$d+1$	$\dots$	$d+1$

Lemma

At any moment, if the first node in the queue is at distance d from S, then all the nodes in the queue are either at distance d from S or at distance $d+1$ from S. 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

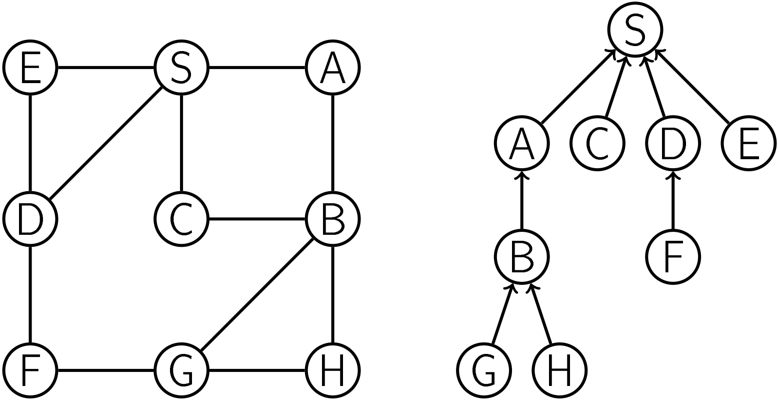
Queue property

Proof

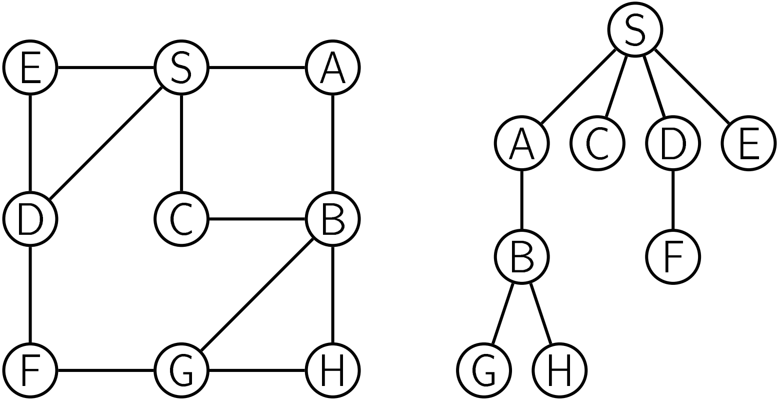
- All nodes at distance d were enqueued before first such node is dequeued, so they go before nodes at distance $d + 1$
- Nodes at distance $d - 1$ were enqueued before nodes at d, so they are not in the queue anymore
- Nodes at distance $> d + 1$ will be discovered when all d are gone $\square$

Outline

Shortest-path tree



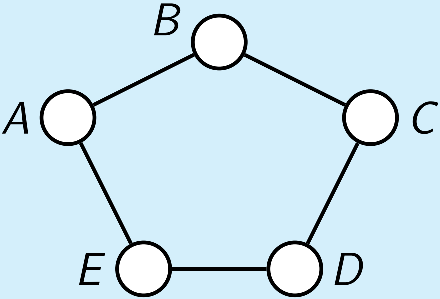
Shortest-path tree



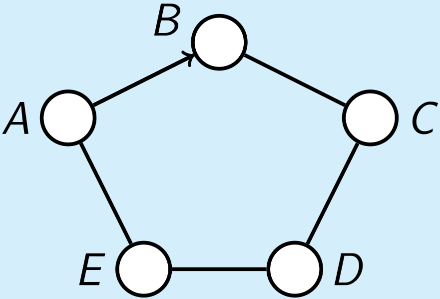
Lemma

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

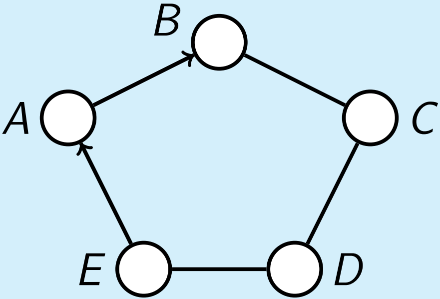
Proof



Proof

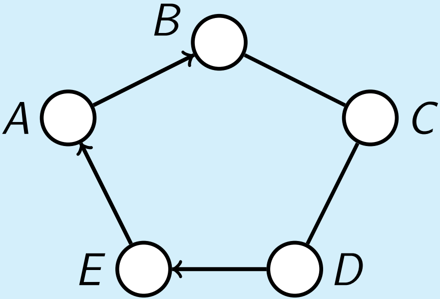


Proof



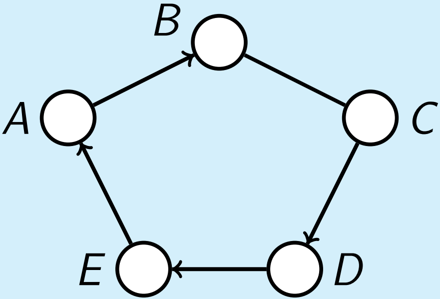
- Only one outgoing edge from each node

Proof



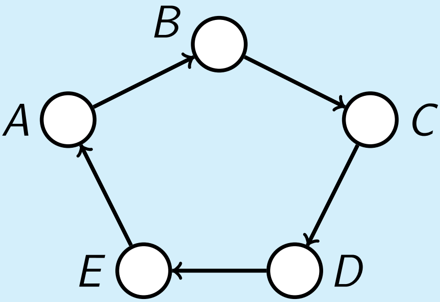
- Only one outgoing edge from each node

Proof



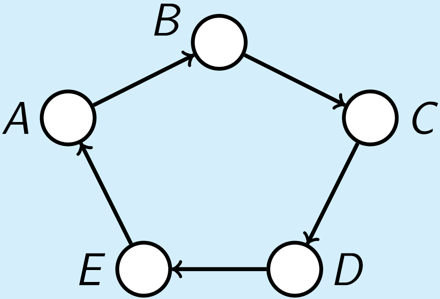
- Only one outgoing edge from each node

Proof



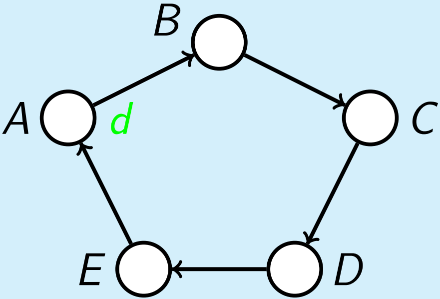
- Only one outgoing edge from each node

Proof



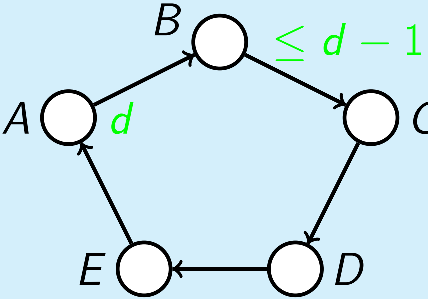
- Only one outgoing edge from each node
- Distance to S decreases after going by edge

Proof



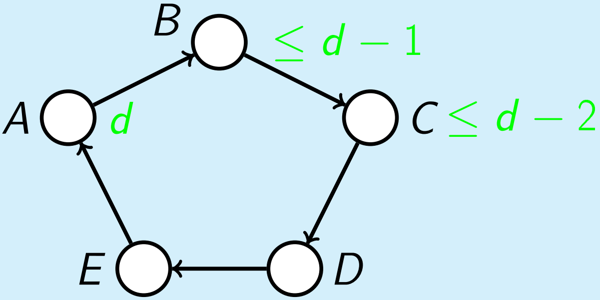
- Only one outgoing edge from each node
- Distance to S decreases after going by edge

Proof



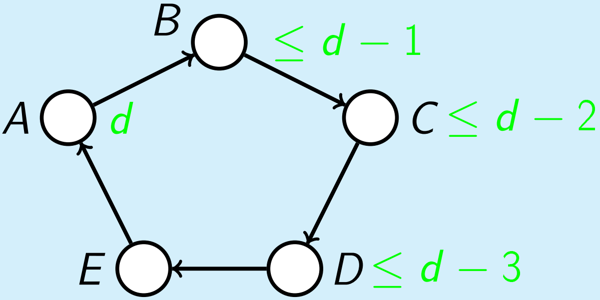
- Only one outgoing edge from each node
- Distance to S decreases after going by edge

Proof



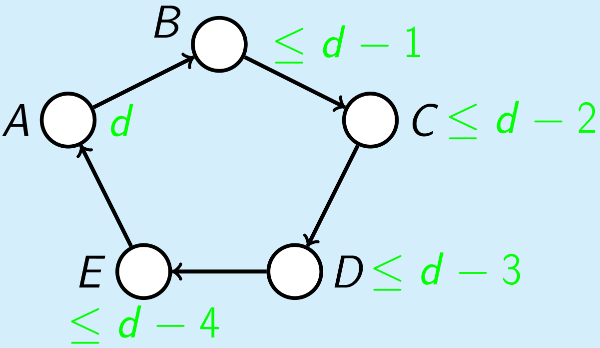
- Only one outgoing edge from each node
- Distance to S decreases after going by edge

Proof



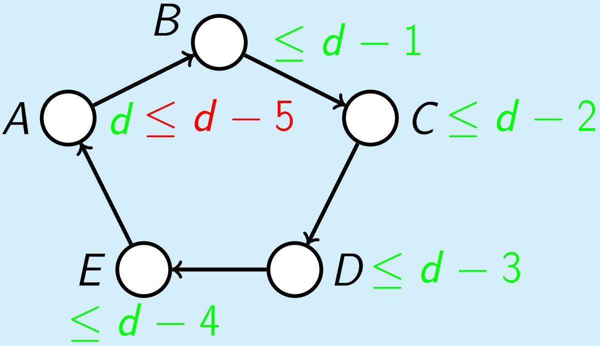
- Only one outgoing edge from each node
- Distance to S decreases after going by edge

Proof



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

Proof



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

Constructing shortest-path tree

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

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

Reconstructing Shortest Path

ReconstructPath(S, u, prev)

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

because we started in reverse order from u to S

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 Meaning can find the shortest path from origin node to all nodes and not just target
- Works in $O(|E| + |V|)$