Graph Colouring and Optimisation Problems

Cycles Girth Bipartite graphs Weighting

Lecturer: Georgy Gimel'farb

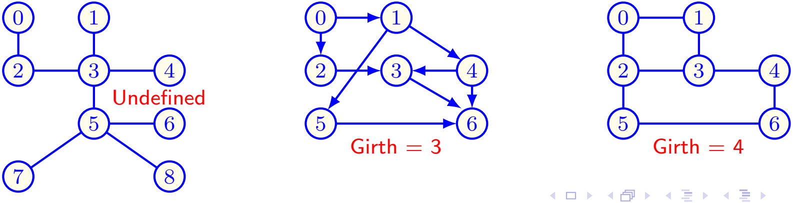
COMPSCI 220 Algorithms and Data Structures

1. Cycles and girth
2. Bipartite graphs
3. Maximum matchings in bipartite graphs
4. Weighted digraphs and optimisation problems
5. Distance and diameter in the unweighted case

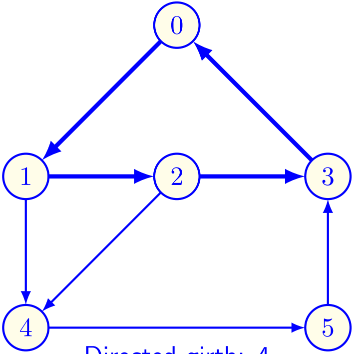
Girth of a Graph (Digraph)

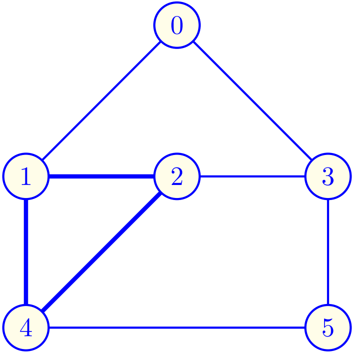
For a graph (with a cycle), the length of the shortest cycle is called the **girth** of the graph.

- If the graph has no cycles, then the girth is undefined, but may be viewed as $+\infty$.
- For a digraph, we use the term girth for its underlying graph.
- We use a (maybe, non-standard) term **directed girth** for the length of the smallest directed cycle.



Girth / Directed Girth: An Example





In general, girth $\leq$ directed girth

Exception: a directed graph can have a cycle of length 2, which is not a cycle in the underlying graph.

Computing the Girth of a Graph G

Length of a shortest cycle containing a given vertex v in G:

- Do `BFSvisit`(v) (if v is on at least one cycle, it will be found):
  - If a GREY neighbour is met, i.e., if an edge (x, y) is explored from x, but y is already GREY, continue only to the end of the current level and then stop.
  - For each edge (x, y), as above on this level:
    - Let v be the lowest common ancestor of x and y in the BFS tree.
    - Then there is a cycle containing x, y, v of length $l = d(v, x) + d(v, y) + 1$ °⁾.
    - Report the minimum l obtained along the current level.

To compute girth, run the above procedure once for each $v \in V(G)$ and take the minimum.

°⁾ $d(v, u)$ – the length of a path of tree arcs from v to u.

Computing the Girth of a Graph G

Length of a shortest cycle containing a given vertex v in G:

- Do BFSvisit(v) (if v is on at least one cycle, it will be found):
  - If a GREY neighbour is met, i.e., if an edge (x, y) is explored from x, but y is already GREY, continue only to the end of the current level and then stop.
  - For each edge (x, y), as above on this level:
    - Let v be the lowest common ancestor of x and y in the BFS tree.
    - Then there is a cycle containing x, y, v of length $l = d(v, x) + d(v, y) + 1$ °).
    - Report the minimum l obtained along the current level.

To compute girth, run the above procedure once for each $v \in V(G)$ and take the minimum.

°) $d(v, u)$ – the length of a path of tree arcs from v to u.

Computing the Girth of a Graph G

Length of a shortest cycle containing a given vertex v in G:

- Do BFSvisit(v) (if v is on at least one cycle, it will be found):
  - If a GREY neighbour is met, i.e., if an edge (x, y) is explored from x, but y is already GREY, continue only to the end of the current level and then stop.
  - For each edge (x, y), as above on this level:
    - Let v be the lowest common ancestor of x and y in the BFS tree.
    - Then there is a cycle containing x, y, v of length $l = d(v, x) + d(v, y) + 1$ °).
    - Report the minimum l obtained along the current level.

To compute girth, run the above procedure once for each $v \in V(G)$ and take the minimum.

°) $d(v, u)$ – the length of a path of tree arcs from v to u.

Computing the Girth of a Graph G

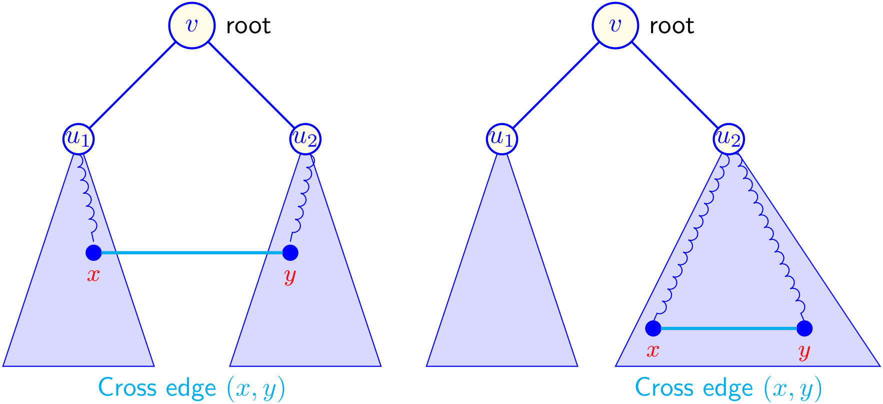
Length of a shortest cycle containing a given vertex v in G:

- Do BFSvisit(v) (if v is on at least one cycle, it will be found):
  - If a GREY neighbour is met, i.e., if an edge (x, y) is explored from x, but y is already GREY, continue only to the end of the current level and then stop.
  - For each edge (x, y), as above on this level:
    - Let v be the lowest common ancestor of x and y in the BFS tree.
    - Then there is a cycle containing x, y, v of length $l = d(v, x) + d(v, y) + 1$ [∘)].
    - Report the minimum l obtained along the current level.

To compute girth, run the above procedure once for each $v \in V(G)$ and take the minimum.

[∘)] $d(v, u)$ – the length of a path of tree arcs from v to u.

Computing the Girth of a Graph G

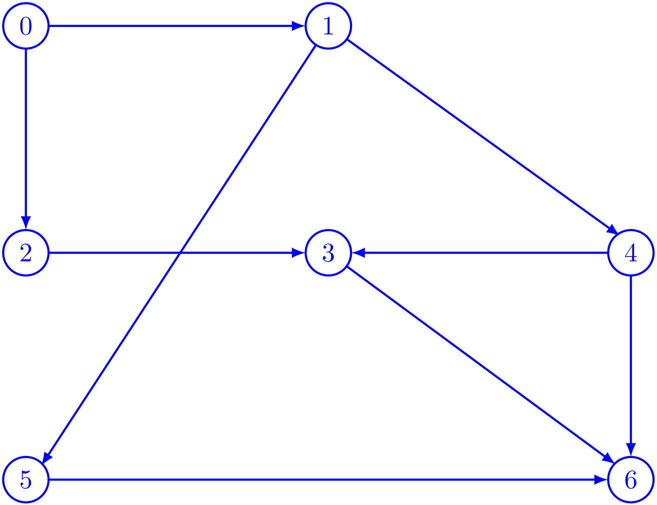


(x, y) is a cross edge because neither x, nor y is an ancestor of other in the BFS tree.

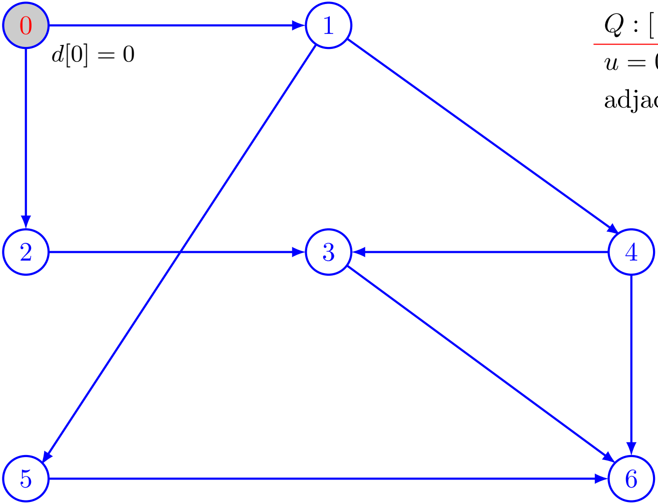
Length of the cycle containing the root v: $dist[x] + dist[y] + 1$

($dist[u]$ – distance from the root v)

Towards Finding Girth of a Graph G



Towards Finding Girth of a Graph G: BFSvisit(0)

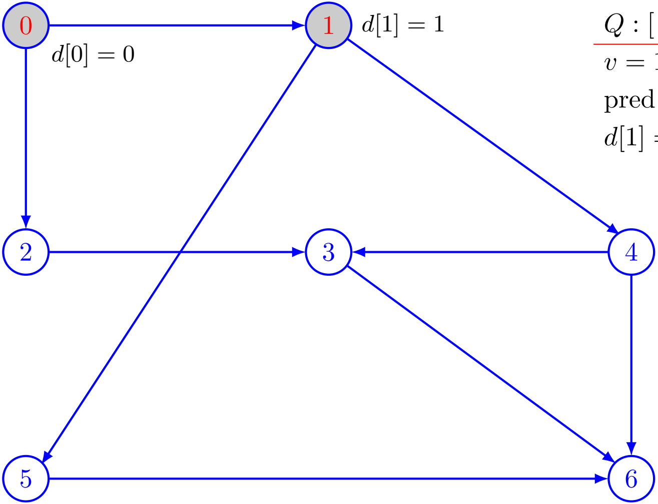


$Q : [\,0\,]$

$u = 0 \leftarrow Q.\texttt{peek}()$

$\text{adjacency}(0) = \{1, 2\}$

Towards Finding Girth of a Graph G: $\texttt{BFSvisit}(0)$



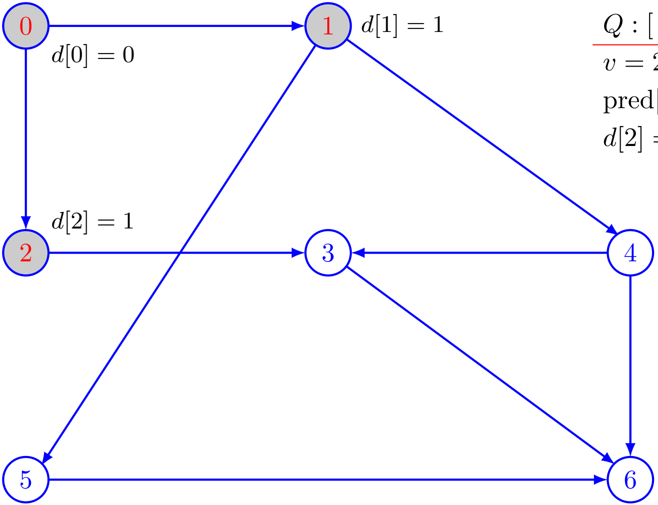
$Q : [\,1\;0\,]$

$v = 1 \Leftarrow \{1, 2\}$

$\text{pred}[1] = 0$

$d[1] = d[0] + 1 = 1$

Towards Finding Girth of a Graph G: BFSvisit(0)



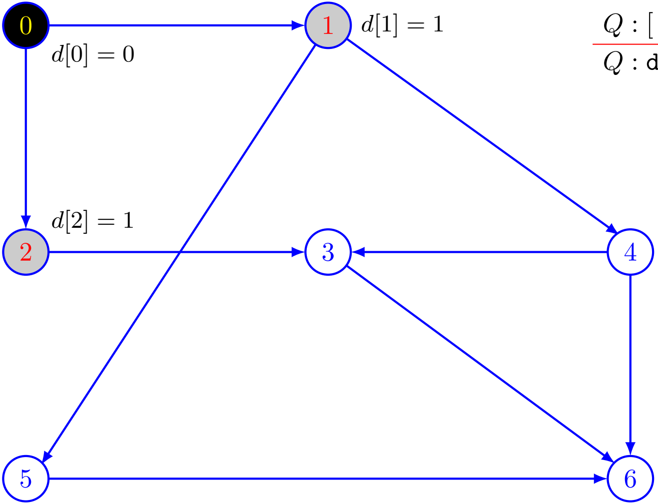
$Q : [\,2\;1\;0\,]$

$v = 2 \Leftarrow \{1, 2\}$

$\text{pred}[2] = 0$

$d[2] = d[0] + 1 = 1$

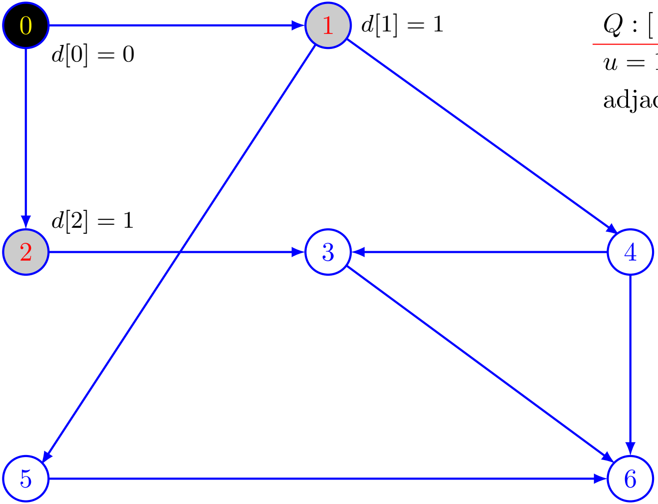
Towards Finding Girth of a Graph G: $\texttt{BFSvisit}(0)$



$Q : [\,2\ 1\,]$

$Q : \texttt{delete}()$

Towards Finding Girth of a Graph G: $\texttt{BFSvisit}(0)$

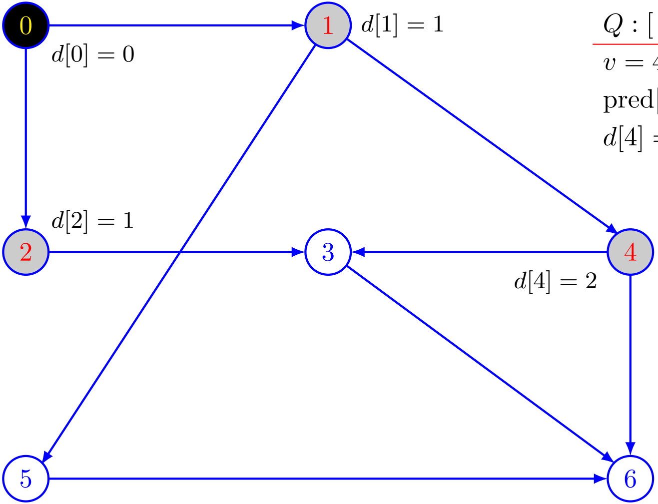


$Q : [\,2\;1\,]$

$u = 1 \leftarrow Q.\texttt{peek}()$

$\text{adjacency}(1) = \{4, 5\}$

Towards Finding Girth of a Graph G: BFSvisit(0)



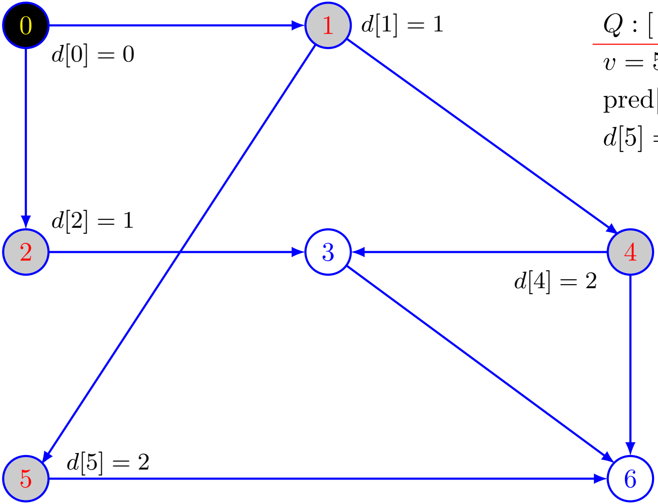
$Q : [\,4\;2\;1\,]$

$v = 4 \Leftarrow \{4, 5\}$

$\mathrm{pred}[4] = 1$

$d[4] = d[1] + 1 = 2$

Towards Finding Girth of a Graph G: $\texttt{BFSvisit}(0)$



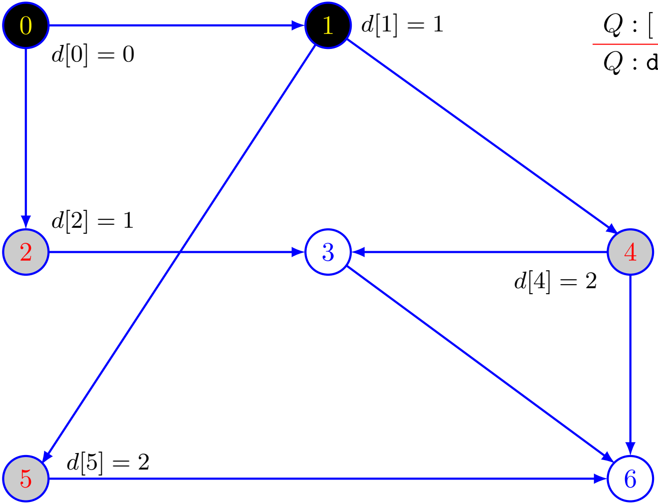
$Q : [\,5\;4\;2\;1\,]$

$v = 5 \Leftarrow \{4, 5\}$

$\mathrm{pred}[5] = 1$

$d[5] = d[1] + 1 = 2$

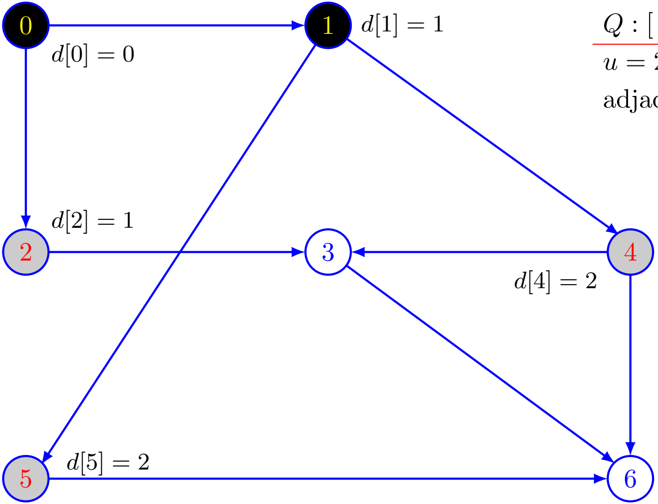
Towards Finding Girth of a Graph G: BFSvisit(0)



Q : [5 4 2]

Q : delete()

Towards Finding Girth of a Graph G: BFSvisit(0)

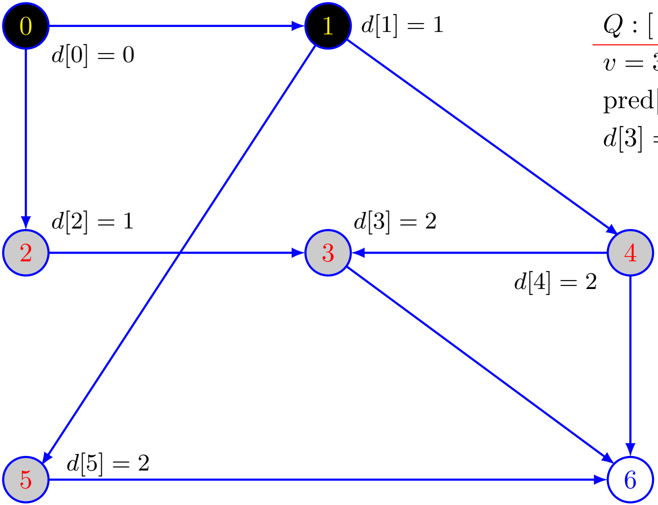


$Q : [\,5\ 4\ 2\,]$

$u = 2 \leftarrow Q.\texttt{peek}()$

$\text{adjacency}(2) = \{3\}$

Towards Finding Girth of a Graph G: $\texttt{BFSvisit}(0)$



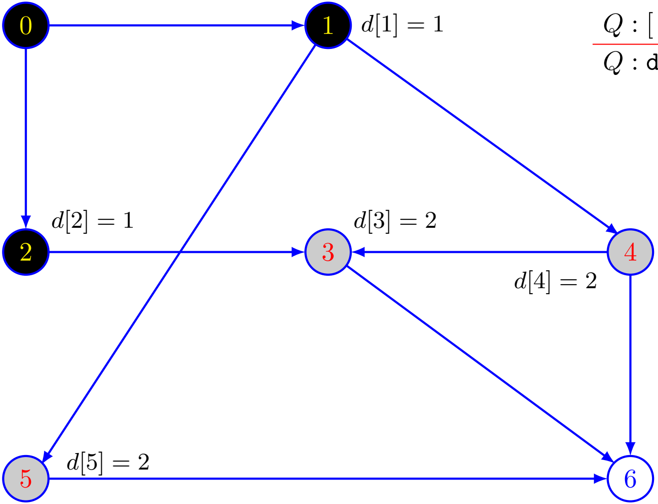
$Q : [\,3\;5\;4\;2\,]$

$v = 3 \Leftarrow \{3\}$

$\text{pred}[3] = 2$

$d[3] = d[2] + 1 = 2$

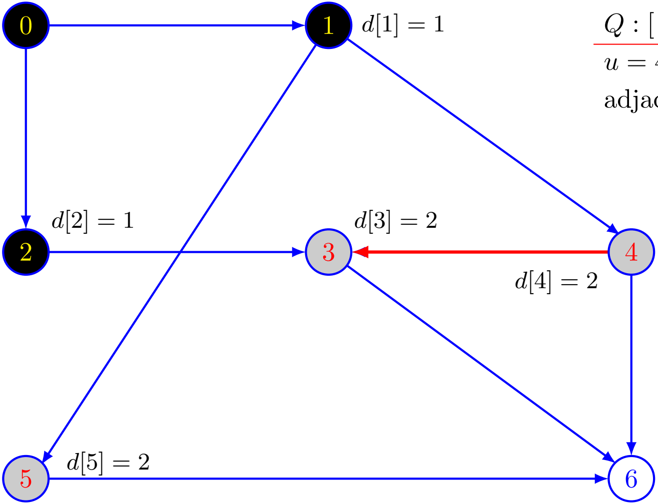
Towards Finding Girth of a Graph G: BFSvisit(0)



Q : [3 5 4]

Q : delete()

Towards Finding Girth of a Graph G: BFSvisit(0)



$Q : [\,3\ 5\ 4\,]$

$u = 4 \leftarrow Q.\texttt{peek}()$

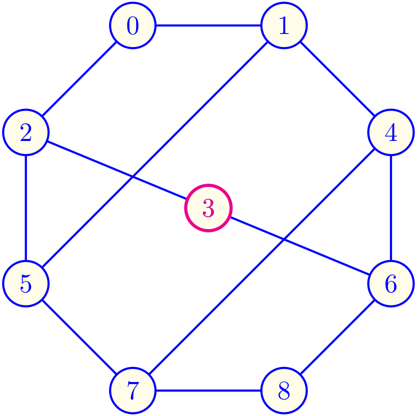
$\text{adjacency}(4) = \{\mathbf{3}, 6\}$

As the vertex 3 is GREY, complete the current level and stop.

Vertex 0 is the common ancestor: so $(0, 1, 4, 3, 2, 0)$ is a cycle of length $d[4] + d[3] + 1 = 5$.

Towards Finding the Girth of a Graph

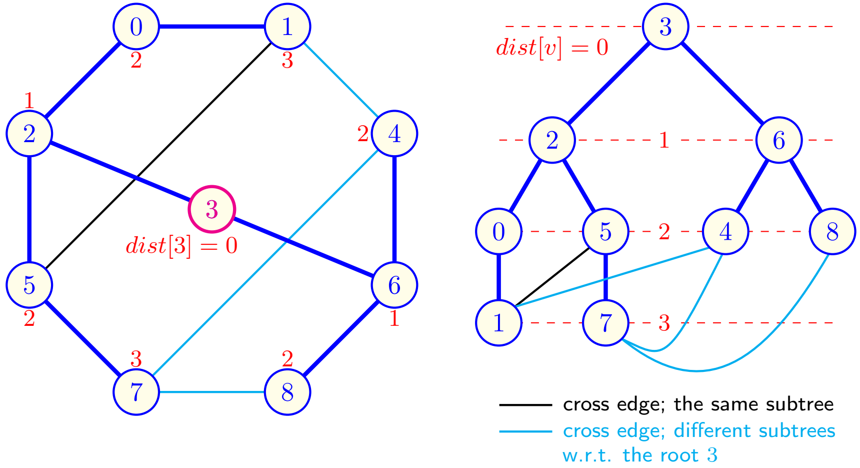
BFS from the node 3



Finding the shortest cycle containing the node 3 by BFS.

Towards Finding the Girth of a Graph

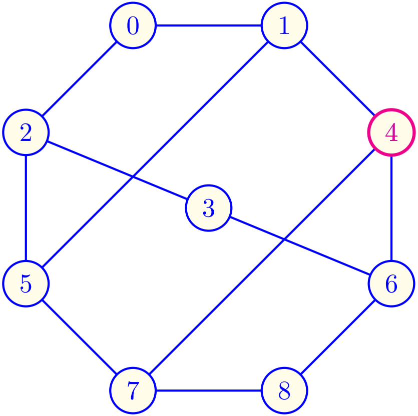
BFS from the node 3



The shortest cycle containing the node 3 is of length 6 $(= 3 + 2 + 1)$.

Towards Finding the Girth of a Graph

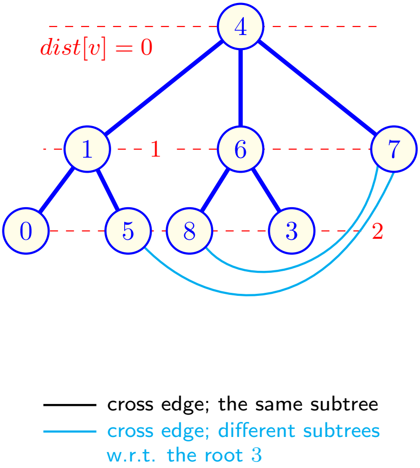
BFS from the node 4



Finding the shortest cycle containing the node 4 by BFS.

Towards Finding the Girth of a Graph

BFS from the node 3



The shortest cycle containing the node 4 is of length 4 $(= 1 + 2 + 1)$.

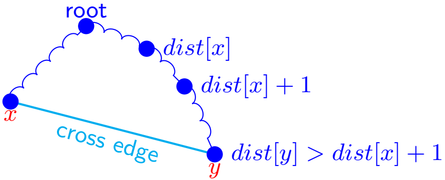
Towards Finding the Girth: Cycle Length

Cross edge (x, y):

- If $dist[x] = dist[y]$, then the cycle has the odd length: $dist[x] + dist[y] + 1 = 2 \cdot dist[x] + 1$.
- If $dist[y] = dist[x] + 1$, then the cycle has the even length: $dist[x]+dist[y]+1 = dist[x]+(dist[x]+1)+1 = 2 \cdot dist[x]+2$.

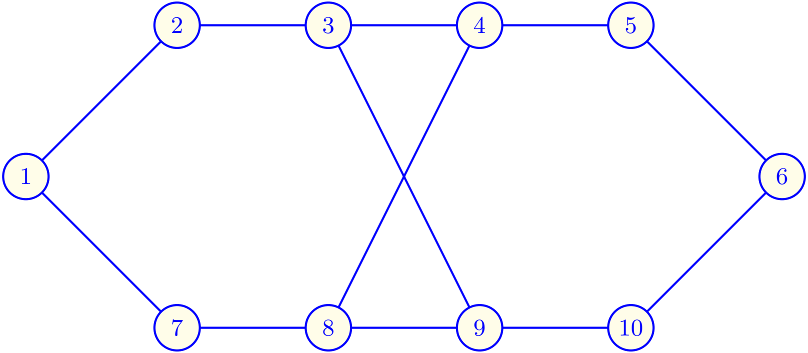
Let (x, y) be a found by BFS cross edge of an undirected graph. Then either $dist[x] = dist[y]$ or $|dist[x] - dist[y]| = 1$.

Sketch of the proof: Suppose that $|dist[x] - dist[y]| > 1$.



Then there is a contradiction: the shortest path from the root to y will contain the cross edge, so that greater than $dist[x] + 1$ values of $dist[y]$ exceed the actual distance from the root to y.

Towards Finding the Girth of a Graph



The shortest cycle containing the vertex 1 has length of 6.

- Just the same: for the vertex 2, 5, 6, 7, and 10.

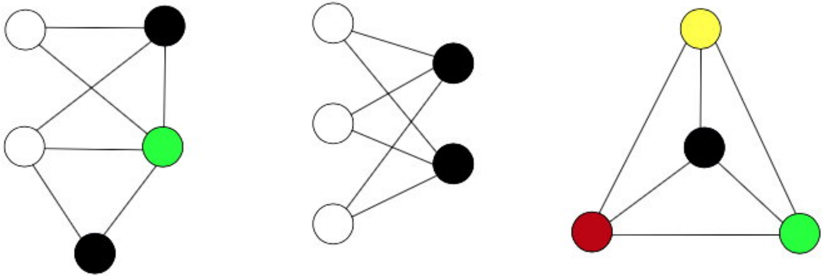
The shortest cycle containing the vertex 3 has length of 4.

- Just the same: for the vertex 4, 8, and 9.

The girth of this graph is 4.

k-colourable and Bipartite Graphs

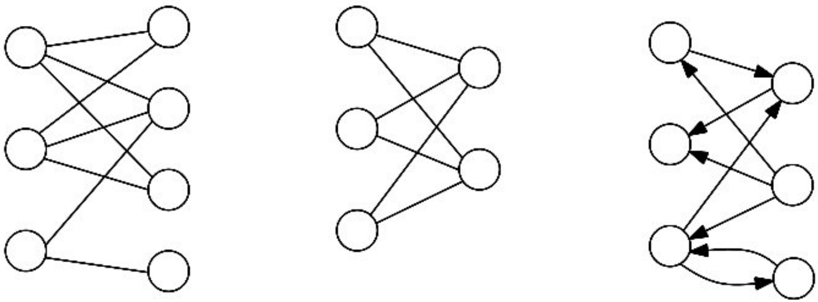
A graph $G = (V, E)$ is k-colourable, where k is a positive integer, if $V(G)$ can be partitioned into k nonempty disjoint subsets, such that each edge in $E(G)$ joins two vertices in different subsets (colours).



A 2-colourable graph is called a bipartite graph.

Bipartite Graphs (Digraphs)

A graph (digraph) $G = (V, E)$ is **bipartite** if $V(G)$ can be partitioned into two nonempty disjoint subsets, $\{V_0, V_1\}$, such that each edge in $E(G)$ has one endpoint in V_0 and one in V_1.



Equivalence of Bipartite and 2-colourable Graphs

Theorem 5.29: The following conditions are equivalent:

- A graph G is bipartite.
- A graph G has a 2-coloring.
- A graph G contains no odd length cycles.

Proof: Bipartition subsets (V_0, V_1) allow for 2-colouring, and vice versa.

- A cycle must have even length, since its start and end vertices have the same colour. □

A version of BFS can check if a graph is bipartite (2-colourable):

- If each vertex at a BFS level i can take the same colour $i \bmod 2$, then each edge is between the vertices of different colours.
- Otherwise, there are adjacent vertices at the same level and odd-length cycles.

Equivalence of Bipartite and 2-colourable Graphs

Theorem 5.29: The following conditions are equivalent:

- A graph G is bipartite.
- A graph G has a 2-coloring.
- A graph G contains no odd length cycles.

Proof: Bipartition subsets (V_0, V_1) allow for 2-colouring, and vice versa.

- A cycle must have even length, since its start and end vertices have the same colour. $\square$

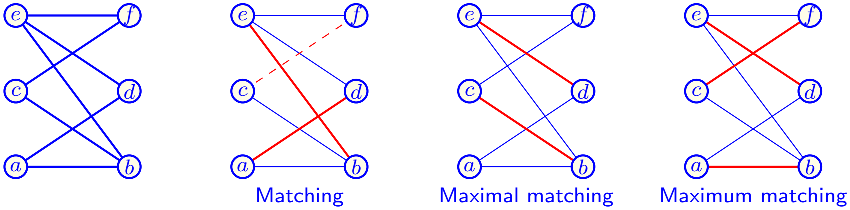
A version of BFS can check if a graph is bipartite (2-colourable):

- If each vertex at a BFS level i can take the same colour $i \bmod 2$, then each edge is between the vertices of different colours.
- Otherwise, there are adjacent vertices at the same level and odd-length cycles.

Maximal and Maximum Matchings in Graphs

A **matching** in a graph is a set of pairwise non-adjacent edges.

- Each vertex can be in at most one edge of the matching.



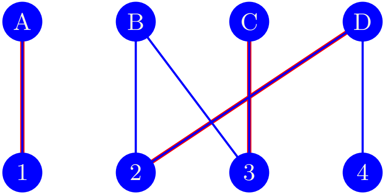
- A **maximal matching** is a matching, which is not a proper subset of any other matching.
- A **maximum matching** is one with the largest possible number of edges (over all possible matchings).

Maximal vs. Maximum Matchings in a Bipartite Graph G

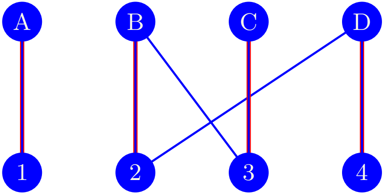
Simple greedy search for a maximal matching:

- Iterating over all edges $e \in E(G)$.
- Adding each edge to a maximal matching M if it is non-adjacent to anything already in M.

A maximal matching may have fewer edges than a more desirable maximum matching.

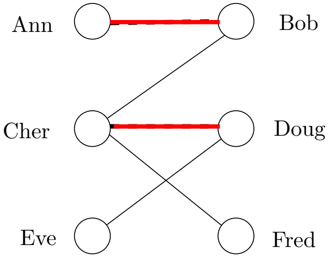


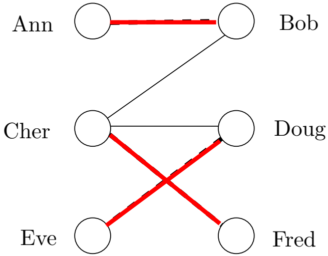
Maximal matching
$M = \{(1, \mathrm{A});\ (2, \mathrm{D});\ (3, \mathrm{C})\}$



Maximum matching
$M = \{(1, \mathrm{A});\ (2, \mathrm{B});\ (3, \mathrm{C});\ (4, \mathrm{D})\}$

Maximal vs. Maximum Matchings in a Bipartite Graph G

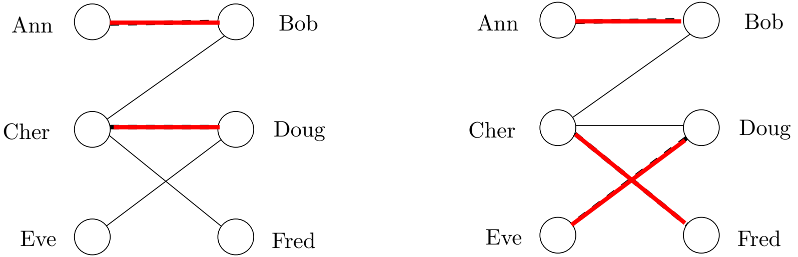




Maximal (left) and **maximum** (right) matching M in G.

- Given a matching M, an **alternating path** is a path in which the edges of the path alternate from being in the matching and not: e.g., Ann–Bob–Cher–Doug–Eve (left).
- An **augmenting path** is an alternating path that starts from and ends on unmatched vertices: e.g., Eve–Doug–Cher–Fred (left).

Improving Maximal Matching M in a Bipartite Graph G



Maximal (left) and **maximum** (right) matching M in G.

- There is always one more non-matching edge than matching edge in an augmenting path: e.g., Eve–Doug–Cher–Fred (left).
- Thus find an augmenting path, remove from M its matching edges (e.g., Doug–Cher), and add to M its non-matching edges (e.g., Fred–Cher and Doug–Eve).
- If there is no augmenting path, M is a maximum matching.

Examples of Maximum Matchings

Example 1: **Worker-Assignment Problem**

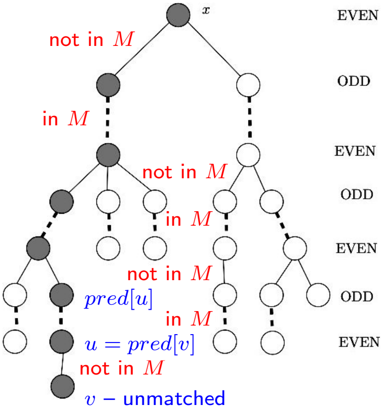
- **Given**: A set of *workers*; a set of *tasks* to be assigned.
- **Constraints**:
  - Each worker is able to perform a subset of the tasks.
  - Each worker can do at most one task at a time.
- **Goal**: Assign (match) as many workers as possible to as many of the tasks.

Example 2: **Marriage Problem**

- **Given**: A set of men and women (as vertices).
- **Constraints**: Edges between compatible relationships.
- **Goal**: Marry as many couples as possible.
  - It is the same as finding a maximum matching in the relationship graph.

Finding Maximum Matchings in Bipartite Graphs

Theorem 5.34: There exists a polynomial-time algorithm to find a maximum matching in a bipartite graph.

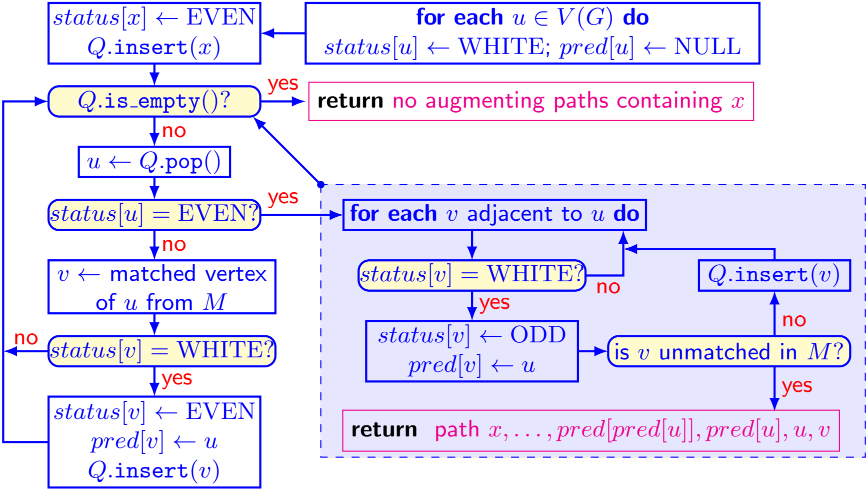


Basic idea of augmenting a given matching M:

1. Start from an unmatched vertex x.
2. Build a tree (via BFS) of alternating paths away from x.
3. If another unmatched vertex is reached, an augmenting path is found.
4. If all vertices are visited, then no augmenting path exists starting at x.

Finding an Augmenting Path: Bipartite graph G; matching M

Unmatched vertex x; queue Q; arrays $pred[0..n-1]$; $status[0..n-1]$

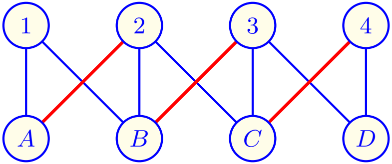


Finding a Maximum Matching: Total Running Time

One invocation of `findAugmentingPath` (Slide 35) can be carried out in time $\mathrm{O}(m)$ for adjacency list representation.

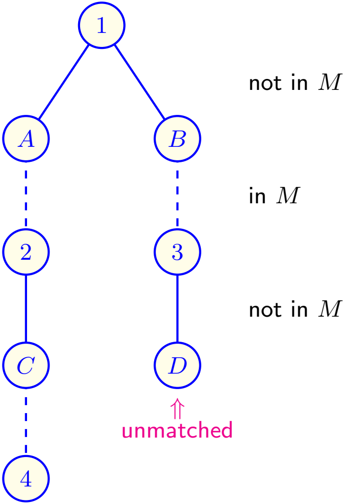
- After an augmenting path is found, the best matching increases by one.
- A maximum matching is bounded by $\left\lfloor \frac{n}{2} \right\rfloor$.
- Thus only at most $\mathrm{O}(n)$ augmenting paths have to be found.
- Potentially, `findAugmentingPath` should be called once for each unmatched vertex, which is bounded by $\mathrm{O}(n)$.
- Because the process has to be repeated for each modified matching, the total running time to find a maximum matching is at most $\mathrm{O}(n^2 m)$.
- The above algorithm can be improved to $\mathrm{O}(mn)$ by traversing and computing an "alternating path forest".
- Further improvements lead to time $\mathrm{O}(m\sqrt{n})$.

Finding an Augmenting Parh: An Example

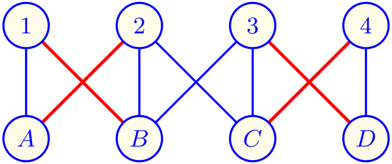


$M = \{(2, A);\ (3, B);\ ((4, C)\} \Rightarrow$

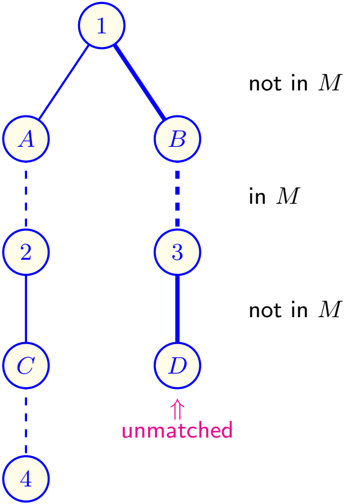
BFS from the unmatched vertex 1



Finding an Augmenting Path: An Example



$M = \{(2,A);\ (3,B);\ ((4,C)\} \Rightarrow$



Augmenting path P found:

$$M' = (M - \{(3,B)\} \bigcup \{(1,B),(3,D)\}$$

$$|M'| = 4 > |M| = 3$$

$M' = \{(1,B);\ (2,A);\ (3,D);\ (4,C)\}$

Weighted (Di)graphs

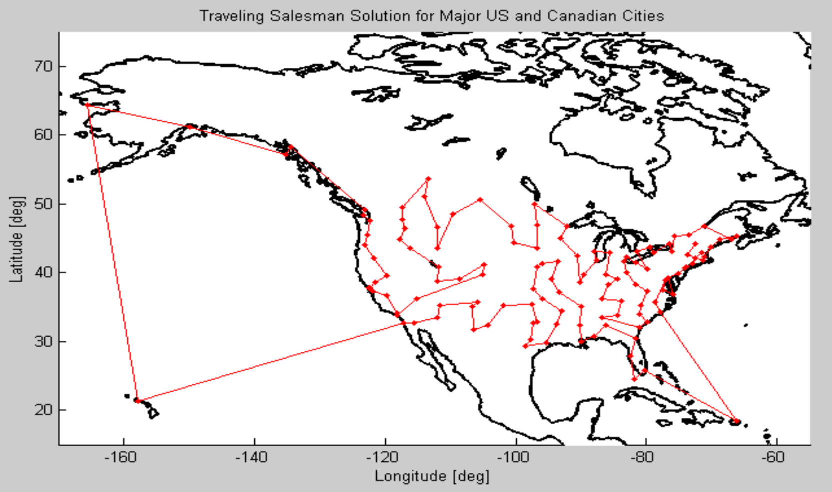
Very common in applications; also called "networks".

- Optimisation on networks is important in operations research, signal processing, navigation etc.
- Each arc carries a real number, or "weight" , which is usually positive and represents cost, distance, or time (can be $+\infty$).
- Representation: weighted adjacency matrix or double adjacency list.

Standard optimisation problems:

- Finding a minimum- or maximum-weight path between given nodes (covered in COMPSCI 220).
- Minimum or maximum spanning tree (COMPSCI 220, 225).
- Optimal cycle or tour (e.g., a computationally hard travelling salesman problem (TSP), matching, flow etc.

Travelling Salesman: An Approximate Solution



http://blogs.mathworks.com/pick/2011/10/14/traveling-salesman-problem-genetic-algorithm/

Weighted (Di)graphs: Definitions

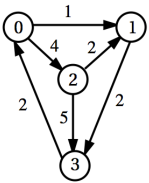
A **weighted digraph** is a pair (G, c) where G is a digraph and c is a **cost function**, associating a real number to each arc of G.

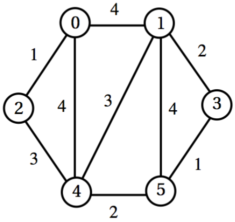
- $c(u, v)$ is interpreted as the cost of using arc (u, v).
- An ordinary digraph is a weighted digraph with the unit cost of each arc.

A **weighted graph** is a symmetric digraph where each pair of antiparallel arcs has the same weight.

- Computer representations: special conventions if there is no arc between u and v.
- An entry of `null` or 0 in a weighted adjacency matrix if the arc does not exist.
  - This entry is equal to ∞ for primitive data types.

Computer Representations of Weighted Digraphs



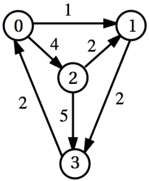


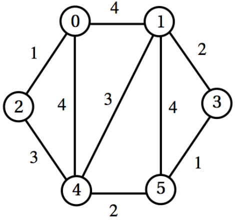
Cost matrices:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 4 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 2 & 0 & 5 \\ 2 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 4 & 1 & 0 & 4 & 0 \\ 4 & 0 & 0 & 2 & 3 & 4 \\ 1 & 0 & 0 & 0 & 3 & 0 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 4 & 3 & 3 & 0 & 0 & 2 \\ 0 & 4 & 0 & 1 & 2 & 0 \end{bmatrix} \end{array}$$

Computer Representations of Weighted Digraphs





Weighted Adjacency Lists:

0 :	1	1	2	4
1 :	3	2		
2 :	1	2	3	5
3 :	0	2		

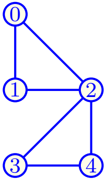
0 :	1	4	2	1	4	4		
1 :	0	4	3	2	4	3	5	4
2 :	0	1	4	3				
3 :	1	2	5	1				
4 :	0	4	1	3	2	3	5	2
5 :	1	4	3	1	4	2		

Diameter of a Strongly Connected Digraph

Definition 6.3

The **diameter** of a strongly connected digraph G is the maximum of distances $d(u,v)$ over all nodes $u,v \in V(G)$.

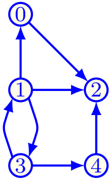
- If a digraph is not strongly connected, the diameter is undefined.
- Two "reasonable" definitions: $+\infty$ or n since no path in G can have length more than $n-1$.
- **Distance matrix** – $d(u,v)]$; $u,v \in V(G)$; by running `BFSvisit` from each node in turn (running time $\Theta(n^2+nm)$).



$$[d(u,v)]_{u,v \in V(G)} = \begin{bmatrix} 0 & 1 & 1 & 2 & 2 \\ 1 & 0 & 1 & 2 & 2 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 2 & 1 & 0 & 1 \\ 2 & 2 & 1 & 1 & 0 \end{bmatrix}$$

diameter $= 2$

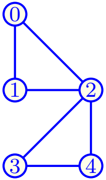
radius $= 1$



diameter $= \infty$

$$\begin{bmatrix} 0 & \infty & 1 & \infty & \infty \\ 1 & 0 & 1 & 1 & 2 \\ \infty & \infty & 0 & \infty & \infty \\ 2 & 1 & 2 & 0 & 1 \\ \infty & \infty & 1 & \infty & 0 \end{bmatrix}$$

Distance Matrix: Eccentricity of a Node



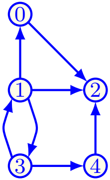
$$\overbrace{\begin{bmatrix} 0 & 1 & 1 & 2 & 2 \\ 1 & 0 & 1 & 2 & 2 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 2 & 1 & 0 & 1 \\ 2 & 2 & 1 & 1 & 0 \end{bmatrix}}^{[d(u,v)]}$$

Eccentricity of a node $u \in V(G)$: $\mathrm{ec}[u] = \max_{v \in V} d(u,v)$:

u	0	1	2	3	4
$\mathrm{ec}[u]$	2	2	1	2	2

Diameter of G: $\max_{u \in V} \mathrm{ec}[u] = 2$.

Radius of G: $\min_{u \in V} \mathrm{ec}[u] = 1$.



$$\overbrace{\begin{bmatrix} 0 & \infty & 1 & \infty & \infty \\ 1 & 0 & 1 & 1 & 2 \\ \infty & \infty & 0 & \infty & \infty \\ 2 & 1 & 2 & 0 & 1 \\ \infty & \infty & 1 & \infty & 0 \end{bmatrix}}^{[d(u,v)]}$$

Eccentricity of a node $u \in V(G)$: $\mathrm{ec}[u] = \max_{v \in V} d(u,v)$:

u	0	1	2	3	4
$\mathrm{ec}[u]$	∞	2	∞	2	∞

Diameter of G: $\max_{u \in V} \mathrm{ec}[u] = \infty$.

Radius of G: $\min_{u \in V} \mathrm{ec}[u] = 2$.

Distances in weighted (di)graphs (G, c): more complex computations.