Algorithms and Data Structures

Graphs

dr Szymon Murawski

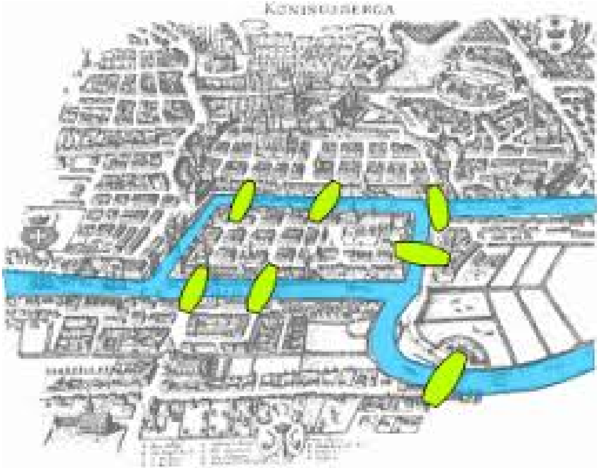
Comarch SA

May 17, 2019

Table of contents I

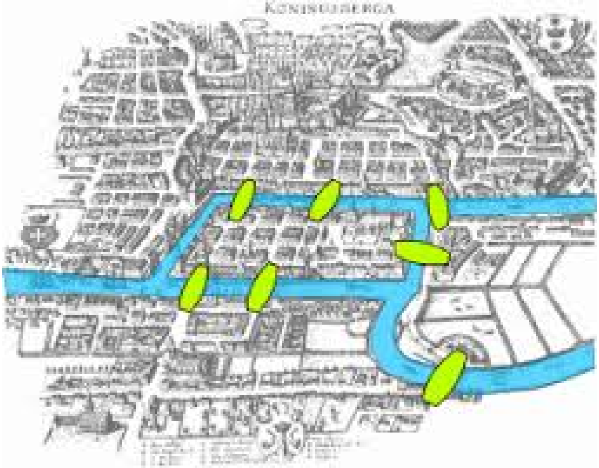
Course plan

Euler and the Seven Bridges of Königsberg



- Is there a path that goes through all the bridges only once?

Euler and the Seven Bridges of Königsberg

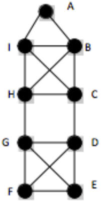


- Is there a path that goes through all the bridges only once?
- Leonhard Euler in 1736: No. By the way here's graph theory.

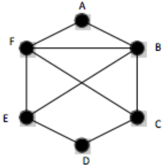
Euler's analysis

- Simplify the problem by changing land mass to vertex and bridge to edge
- Observe, that if we enter a vertex by an edge, we must exit it using another edge. Thus each vertex must have even number of edges
- There are two special vertices for which above can be broken - starting and ending vertex
- A graph then has solution to this problem if the number of vertices with odd number of edges is either 0 or 2

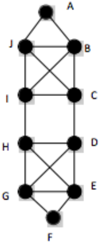
Euler's Path



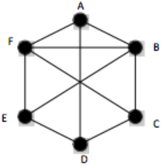
G_1



G_2



G_3



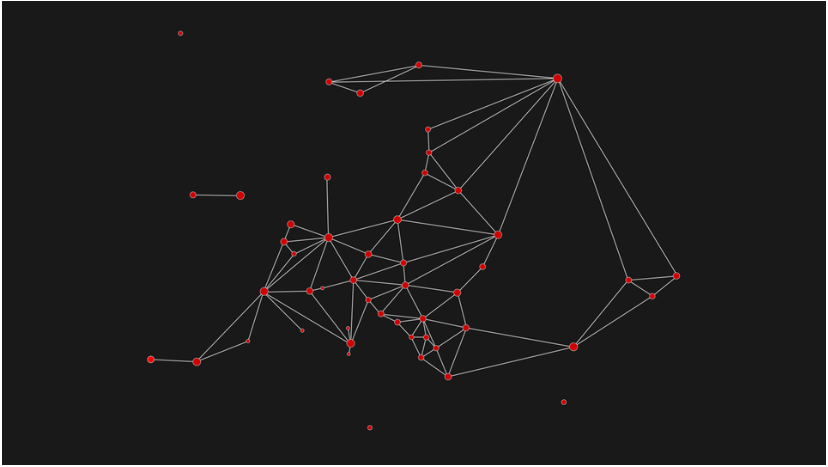
G_4

- Euler Path: a path in graph that visits each edge exactly once
- For a graph to have Euler path it must have exactly 0 or 2 vertices with odd number of edges (odd degree)
- If a graph has an Euler path it can be drawn without lifting off the pencil
- This problem was a foundation for graph theory!

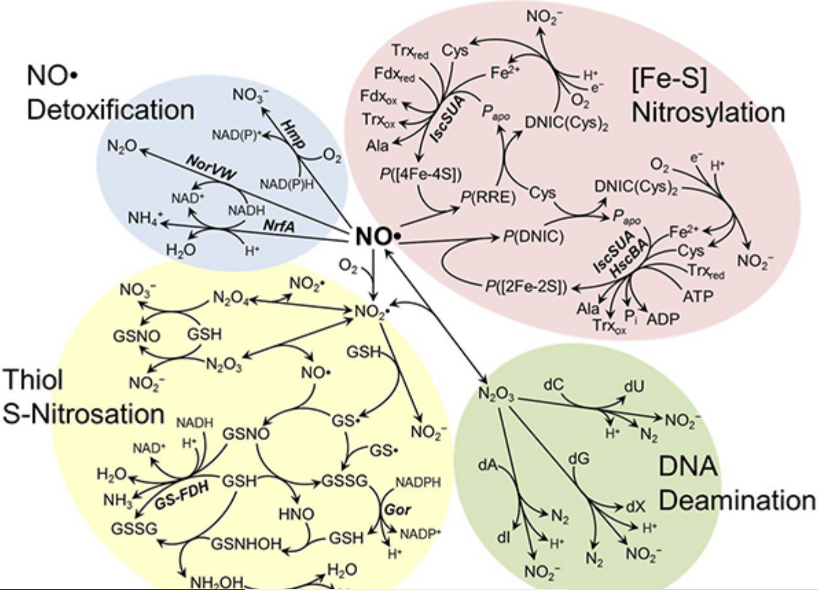
Course plan

Navigation systems

Simplifying complex relations



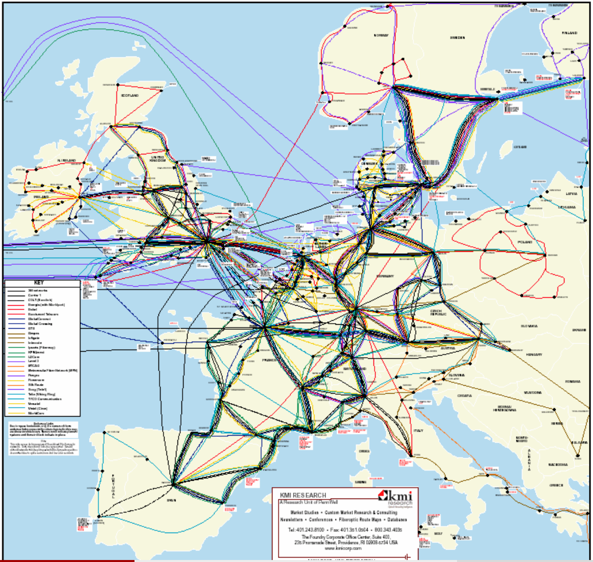
Optimizing chemical reactions



Analysis of social networks

Network analysis

PAN EUROPEAN FIBEROPTIC NETWORK ROUTES PLANNED OR IN PLACE



Course plan

Graph definition

Formal definition

Graph is an order pair $G = (V, E)$, where:

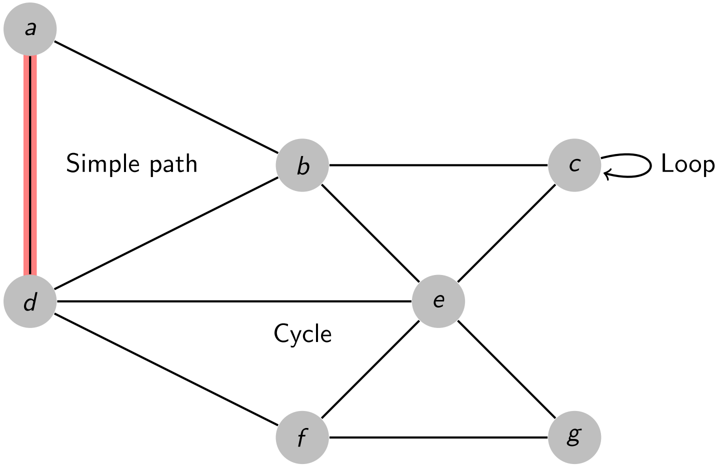
- V is a non-empty, finite set of vertices
- E is a set of edges between vertices in V

- A graph can either be directed, when every edge is traversable only in one way, or undirected.
- Edge in fact is a pair of two vertices
- In applications vertices are objects and edges are connections between objects

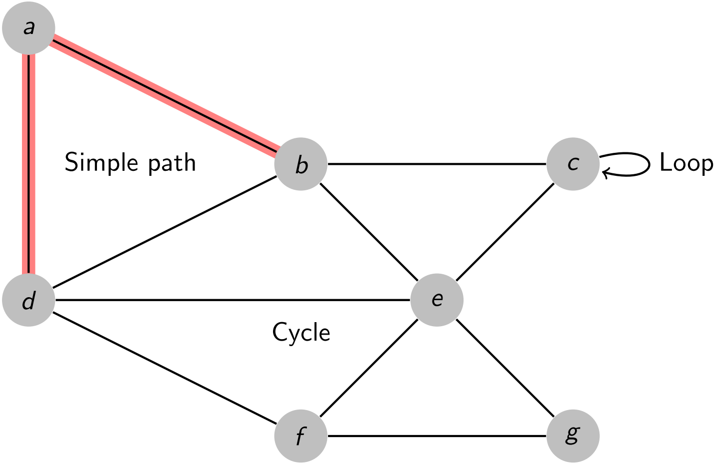
Terminology

- **Adjacent vertices** - vertices that are connected by a single edge
- **Degree of vertex** - number of edges connected to that vertex
- **Parallel edges** - edges that start at the same vertex and end at the same vertex
- **Loop** - edge that start and ends at the same vertex
- **Simple path** - any path in which all edges and vertices are distinct
- **Cycle** - path that starts and ends at the same vertex

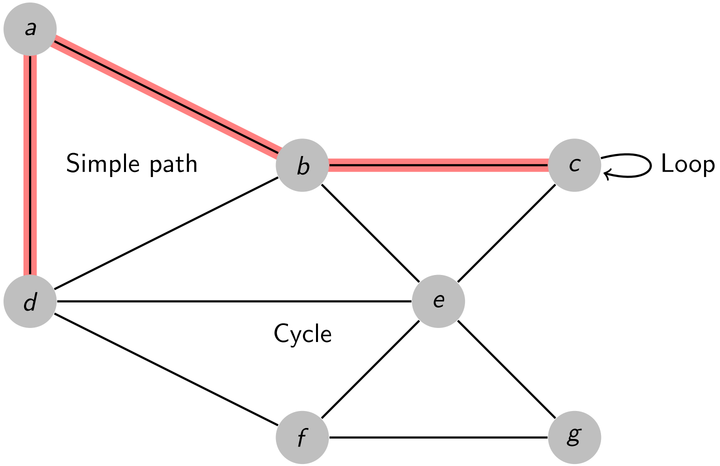
Graph example



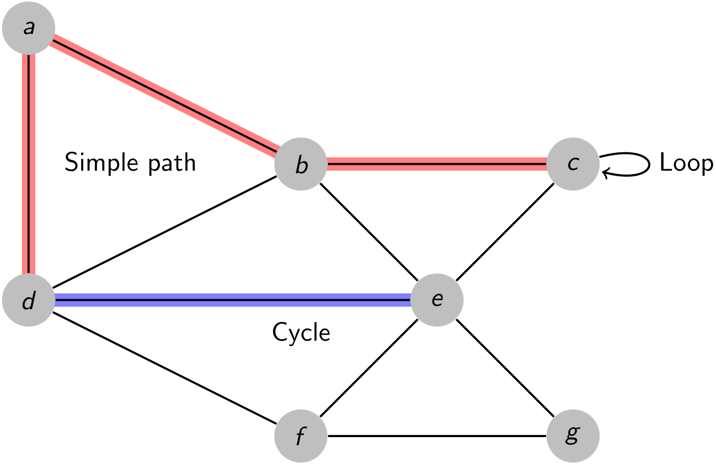
Graph example



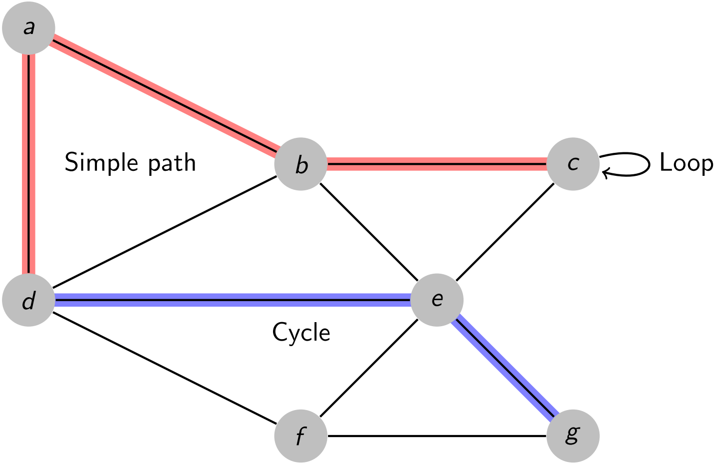
Graph example



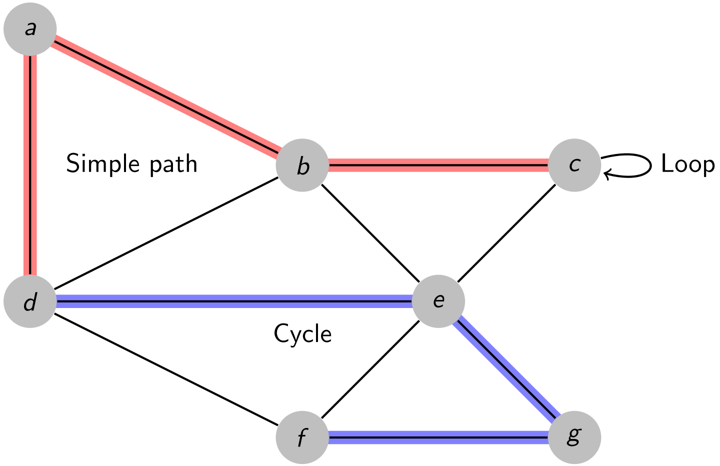
Graph example



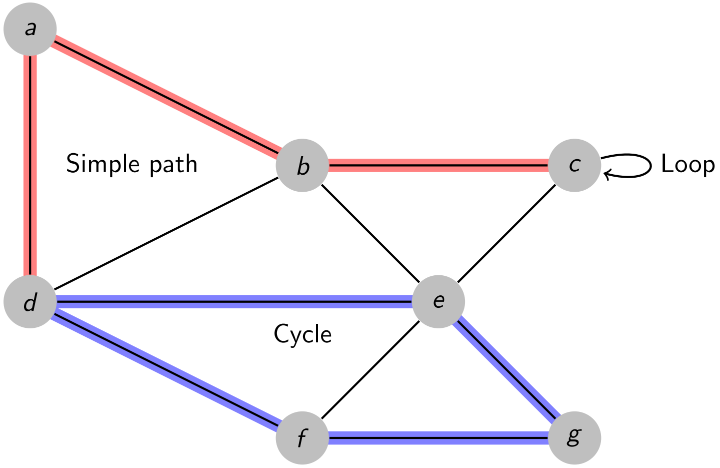
Graph example



Graph example

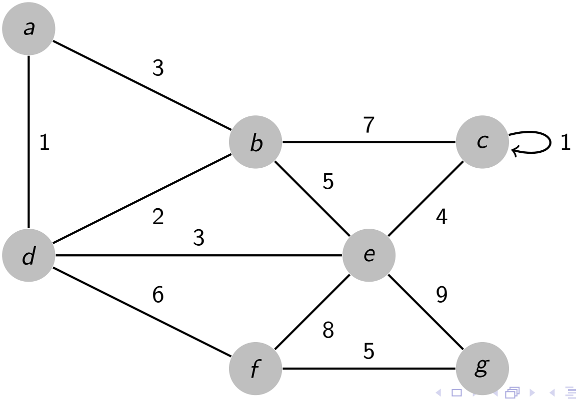


Graph example



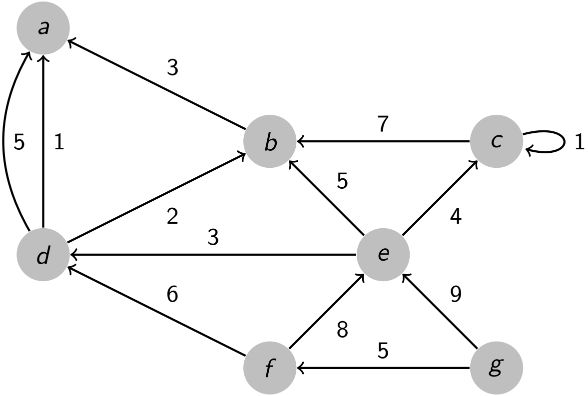
Edge weight

- Each edge might have weight associated to it.
- Weight is a cost of following specific edge
- For graphs without weight we might assume all edges have the same weight of 1 - it will simplify many algorithms



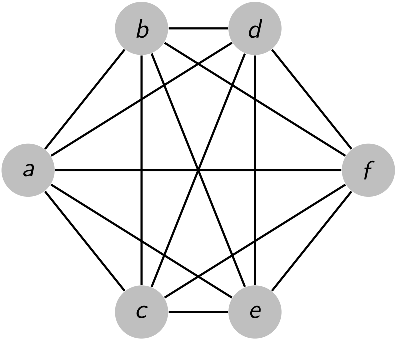
Directed graph (digraph)

- Each edge can be traversed only in one direction
- There might be multiple edges between same vertices, each with different weight



Complete graph

- In complete graph every vertex is connected to each other
- If the graph is unweighted it's called clique, else it's called tournament



Course plan

Graph representation

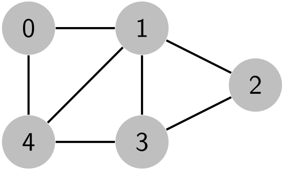
Adjacency matrix

- Adjacency matrix A is a 2D grid of size $V \times V$, where V is number of nodes in graph
- Value $A[i, j]$ indicates an edge between node i and j
- If $A[i, j] == 0$ then edge does not exists, otherwise it exists with a weight $A[i, j]$
- If in weighted graph edge with weight 0 can exists, different value must be used to indicate absence of edge
- For undirected graphs adjacency matrix is symmetrical

Adjacency list

- Adjacency list A is an array of lists
- Element $A[i]$ keeps a list of all vertices connected to vertex i
- For undirected graph list element is a pointer to vertex. For directed graph list element is a pair of pointer to vertex and weight associated with this edge

Undirected graph representation



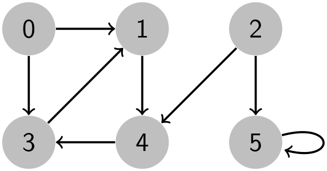
Adjacency list

0 | $\rightarrow 1 \rightarrow 4$
1 | $\rightarrow 0 \rightarrow 2 \rightarrow 3 \rightarrow 4$
2 | $\rightarrow 1 \rightarrow 3$
3 | $\rightarrow 1 \rightarrow 2 \rightarrow 4$
4 | $\rightarrow 0 \rightarrow 1 \rightarrow 3$

Adjacency matrix

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Directed graph representation



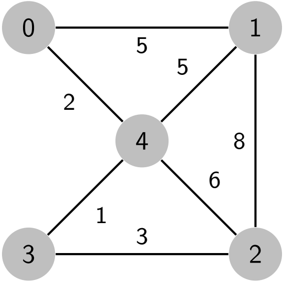
Adjacency list

0	$\rightarrow 1 \rightarrow 3$
1	$\rightarrow 1$
2	$\rightarrow 4 \rightarrow 5$
3	$\rightarrow 1$
4	$\rightarrow 3$
5	$\rightarrow 5$

Adjacency matrix

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

Undirected weighted graph representation



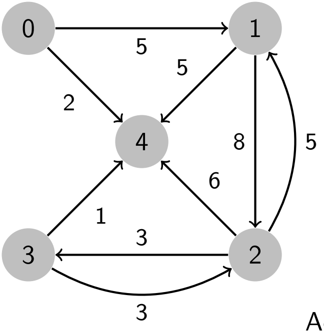
Adjacency list

0	$\rightarrow$ (1,5) $\rightarrow$ (4,2)
1	$\rightarrow$ (0,5) $\rightarrow$ (2,8) $\rightarrow$ (4,5)
2	$\rightarrow$ (1,8) $\rightarrow$ (3,3) $\rightarrow$ (4,6)
3	$\rightarrow$ (2,3) $\rightarrow$ (4,1)
4	$\rightarrow$ (0,2) $\rightarrow$ (1,5) $\rightarrow$ (2,6) $\rightarrow$ (3,1)

Adjacency matrix

	0	1	2	3	4
0	0	5	0	0	2
1	5	0	8	0	5
2	0	8	0	3	6
3	0	0	3	0	1
4	2	5	6	1	0

Directed weighted graph representation



Adjacency list

0	→ (1,5) → (4,2)
1	→ (2,8) → (4,5)
2	→ (1,5) → (3,3) → (4,6)
3	→ (2,3) → (4,1)
4	

Adjacency matrix

	0	1	2	3	4
0	0	5	0	0	2
1	0	0	8	0	5
2	0	5	0	3	6
3	0	0	3	0	1
4	0	0	0	0	0

Adjacency list vs matrix

Complexity

	Adjacency list	Adjacency matrix
Space	$O(V+E)$	$O(V^2)$
areAdjacent(i,j)	$O(min(deg(i), deg(j))$	$O(1)$
addVertex()	$O(1)$	$O(V^2)$
addEdge(i,j)	$O(1)$	$O(1)$
removeVertex(i)	$O(deg(i))$	$O(V^2)$
removeEdge(i,j)	$O(1)$	$O(1)$

- For sparse graphs (graphs with small number of edges) adjacency matrix is better solution.
- For dense graphs adjacency list
- Adjacency matrix cannot be used, if there might be multiple edges between same vertices

Course plan

Graph interface

- Depending on the problem, there might be additional attributes assigned to edges/vertices, or maybe additional behaviour regarding moving through graph is added.
- Because of that there is no generic graph class in C#
- Luckily writing our own graph implementation is fairly simple
- Implementation will differ depending on whether adjacency matrix or list is used
- Most implementations will provide following functions:
  - `getVertices()` - returns a list of all vertices in graph
  - `getNeighbours(vertex)` - returns a list of all neighbours of specified vertex
  - `areAdjacent(vertex1, vertex2)` - checks whether two nodes are neighbours, might also return weight of the edge
  - `addVertex(data)` - adds new vertex to the graph
  - `addEdge(vertex1, vertex2, weight)` - adds new edge to the graph
  - `removeVertex(label)` - removes vertex from the graph
  - `removeEdge(vertex1, vertex2)` - removes edge between specified vertices

Course plan

Vertex class in C#

```
class Vertex<T>{
  T data;
  string label;
  List<Vertex<T>> neighbours;

  public Vertex(T data){
    this.data = data;
  }
  public Vertex(T data, List<Vertex<T>> neighbours)
  {
    this.data = data;
    this.neighbours = neighbours;
  }
  public int GetDegree() {
    return neighbours.Count;
  }
  public List<Vertex<T>> GetNeighbours() {
    return neighbours;
  }
```

Vertex class in C# continued

```
  public void AddEdge(Vertex<T> vertex){
    neighbours.Add(vertex);
  }
  public void RemoveEdge(Vertex<T> vertex){
    neighbours.Remove(vertex);
  }
  public bool HasNeighbour(Vertex<T> vertex){
    return neighbours.Contains(vertex);
  }
}
```

Graph class in C#

```
class Graph<T>{
  List<Vertex<T>> vertices;

  public Graph(){
    this.vertices = new List<Vertex<T>>();
  }
  public Graph(List<Vertex<T>> vertices){
    this.vertices = vertices;
  }
  public List<Vertex<T>> GetVertices() {
    return vertices;
  }
  public int GetSize() {
    return vertices.Count;
  }
  public List<Vertex<T>> GetNeighbours(Vertex<T> vertex){
    return vertex.GetNeighbours();
  }
```

Graph class in C# continued

```
    public void AddVertex(Vertex<T> vertex){
      vertices.Add(vertex);
    }
    public void RemoveVertex(Vertex<T> vertex){
      vertices.Remove(vertex);
    }
    public void AddEdge(Vertex<T> vertex1, Vertex<T> vertex2){
      vertex1.AddEdge(vertex2);
      vertex2.AddEdge(vertex1);
    }
    public void RemoveEdge(Vertex<T> vertex1, Vertex<T> vertex2){
      vertex1.RemoveEdge(vertex2);
      vertex2.RemoveEdge(vertex1);
    }
    public bool AreAdjacent(Vertex<T> vertex1, Vertex<T> vertex2){
      return vertex1.HasNeighbour(vertex2);
    }
}
```

Course plan

Vertex class in C#

```
class Vertex<T>{
  int index;
  T data;

  public Vertex(int index, T data) {
    this.index = index;
    this.data = data;
  }
  public int GetIndex() {
    return index;
  }
}
```

Graph class in C#

```
class Graph<T> {
  List<Vertex<T>> vertices;
  int iterator;
  int maxNumberOfVertices;
  bool[,] adjacencyMatrix;

  public Graph(int maxNumberOfVertices) {
    this.vertices = new List<Vertex<T>>();
    this.iterator = 0;
    this.maxNumberOfVertices = maxNumberOfVertices;
    this.adjacencyMatrix = new bool[maxNumberOfVertices,
    maxNumberOfVertices];
  }
  public List<Vertex<T>> GetVertices() {
    return vertices;
  }
  public int GetSize() {
    return vertices.Count;
  }
```

Graph class in C# continued

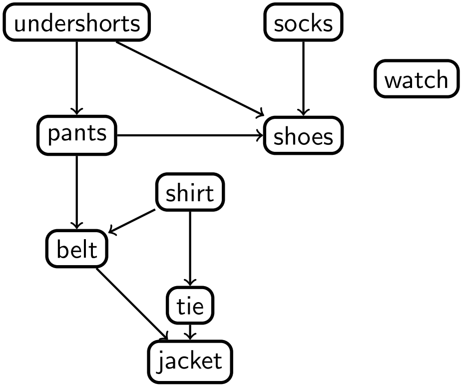
```
  public List<Vertex<T>> GetNeighbours(Vertex<T> vertex) {
    List<Vertex<T>> neighbours = new List<Vertex<T>>();
    for (int i = 0; i < maxNumberOfVertices; i++) {
      if (adjacencyMatrix[i, vertex.GetIndex()]) {
        Vertex<T> neighbour = neighbours.Find(ver => ver.GetIndex
    () == i);
      }
    }
    return neighbours;
  }
  public void AddVertex(T data) {
    vertices.Add(new Vertex<T>(iterator, data));
    iterator++;
  }
  public void RemoveVertex(Vertex<T> vertex){
    int vertexIndex = vertex.GetIndex();
    for (int i = 0; i < maxNumberOfVertices; i++) {
      adjacencyMatrix[i, vertexIndex] = false;
      adjacencyMatrix[vertexIndex, i] = false;
    }
    vertices.Remove(vertex);
  }
```

Graph class in C# continued

```
    public void AddEdge(Vertex<T> vertex1, Vertex<T> vertex2){
      adjacencyMatrix[vertex1.GetIndex(), vertex2.GetIndex()] =
      true;
      adjacencyMatrix[vertex2.GetIndex(), vertex1.GetIndex()] =
      true;
    }
    public void RemoveEdge(Vertex<T> vertex1, Vertex<T> vertex2)
    {
      adjacencyMatrix[vertex1.GetIndex(), vertex2.GetIndex()] =
      false;
      adjacencyMatrix[vertex2.GetIndex(), vertex1.GetIndex()] =
      false;
    }
    public bool AreAdjacent(Vertex<T> vertex1, Vertex<T> vertex2) {
      return adjacencyMatrix[vertex1.GetIndex(), vertex2.GetIndex()
      ];
    }
}
```

Course plan

Putting on clothes



What is the order in which clothes should be put on?

Topological sort

- Procedure for transforming a graph into linear list of vertices is called topological sorting
- Topological sorting is also called topological ordering
- It has many applications, as many processes can be described in form of graphs, but in reality only one task can be performed at time (i.e. car production line, build order in software)
- Topological ordering can only be applied to directed acyclic graph
- Every directed graph has at least one topological ordering

Definition

Topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

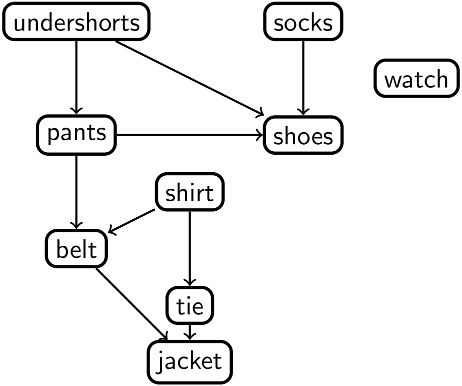
Kahn's algorithm for topological ordering

- In Kahn's algorithm we select a vertex with no incoming edges, remove it and it's edges from the graph and add it to output set S
- Set S can either be implemented as a stack or queue, depending on what order we prefer
- To do that we need to calculate in-degree of every vertex in a graph
- Complexity of this algorithm is $O(V + E)$
- Another algorithm to produce topological ordering uses DFS (Depth-First Search)
- Neat feature of this algorithm is the ability to spot cycles

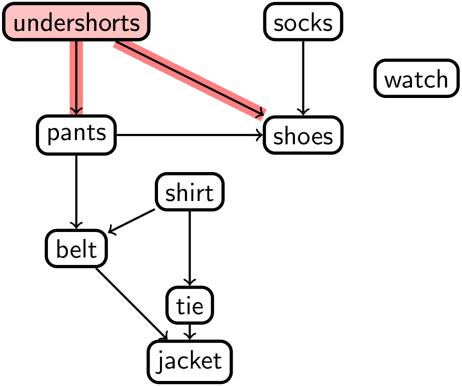
Kahn's algorithm pseudocode

```
TopologicalSort(Graph){
  //calculating in degree
  for each vertex in Graph
    vertex.inDegree = 0;
  for each edge in Graph
    edge.destination.inDegree++;
  //topological sort
  new Queue // queue for vertices with inDegree == 0
  new Set // output set
  for each vertex in Graph
    if vertex.inDegree == 0
      Queue.enqueue(vertex)
  while Queue is not empty
    vertexSource = Queue.dequeue
    Set.push_back(vertex)
    for each edge in Graph where edge.source == vertexSource
      edge.destination.inDegree--;
      if edge.destination.inDegree == 0
        Queue.enqueue(edge.destination)
      remove edge from Graph
  if Graph has edges
    print "Graph is cyclic, topological ordering is not possible"
  else
    return Set
}
```

Topological ordering example

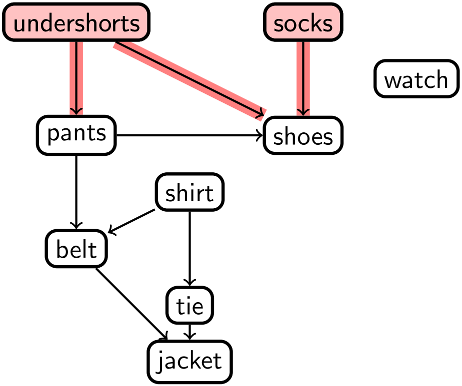


Topological ordering example

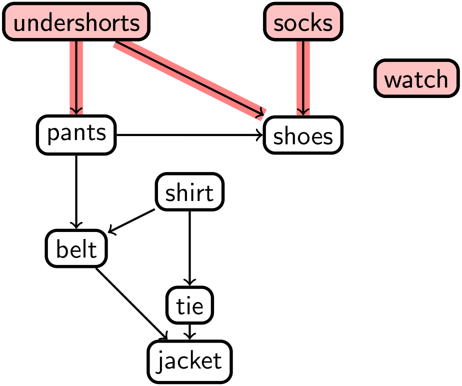


undershorts

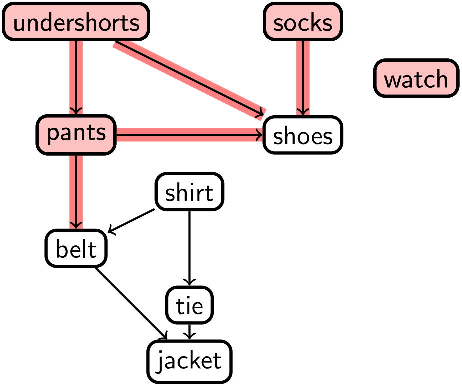
Topological ordering example

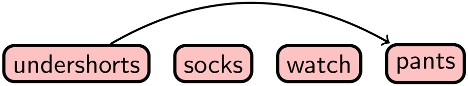


Topological ordering example

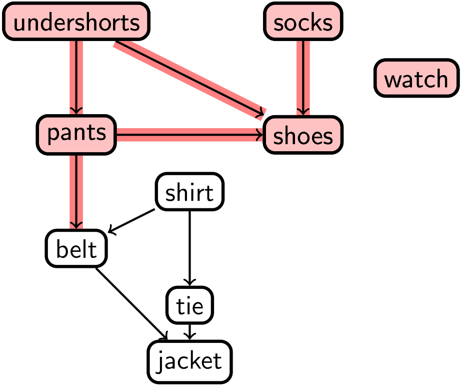


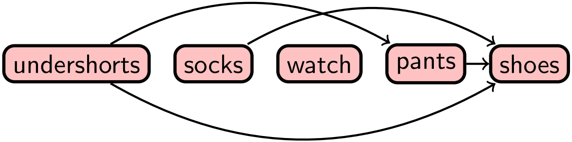
Topological ordering example



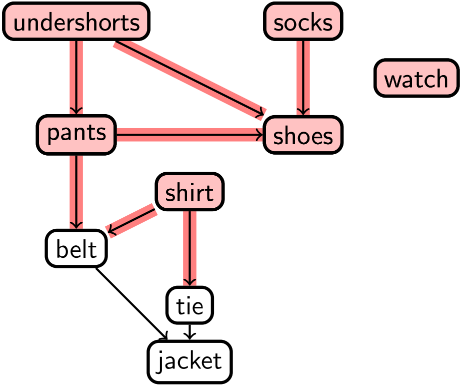


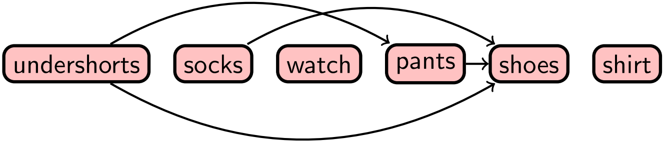
Topological ordering example



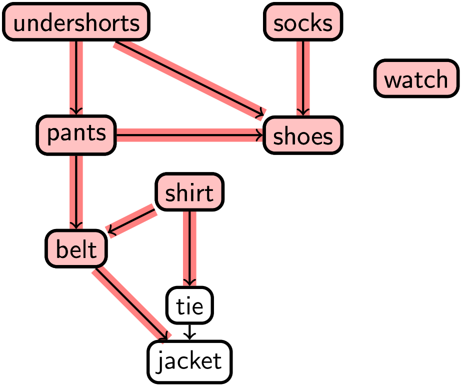


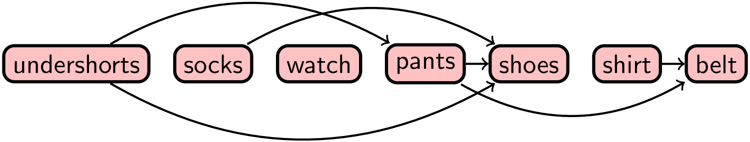
Topological ordering example



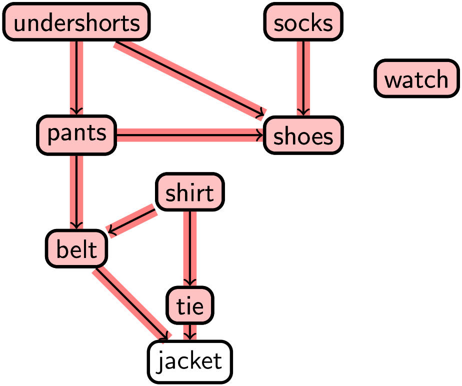


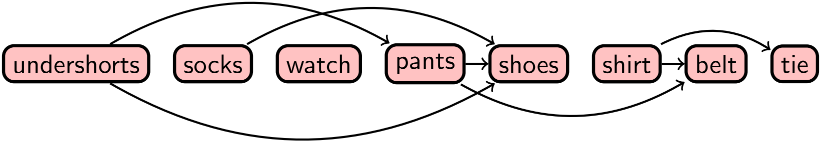
Topological ordering example





Topological ordering example





Topological ordering example