Diskretna matematika 2

Gradiva za vaje iz diskretne matematike 2

Univerza v Ljubljani, Fakulteta za računalnišvo in informatiko

Marko Boben

June 2023

Contents

List of Figures

List of Tables

1. Introduction

1.1 What is Sage?

Algorithms in this Notes are implemented in Python programming language using SageMath (https://www.sagemath.org).

> SageMath is a free open-source mathematics software system licensed under the GPL. It builds on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more.

You can download binaries at http://www.sagemath.org/download.html for Mac, and Windows.

Note: Binaries for Windows are avaliable up to version 9.3 (late 2021). For newer versions you will need to install it in WSL. Follow the instructions athttps://doc.sagemath.org/html/en/installation/index.html.

There is also a cloud version available at https://cocalc.com/
Documentation can be found at https://doc.sagemath.org/html/en/index.html.
We will moslty use *graph theory* package https://doc.sagemath.org/html/en/reference/graphs/index.html

1.2 Some examples of Sage Graph Theory objects and methods

For representing undirected graphs we use the `Graph` class, while for representing directed graphs we use the `DiGraph` class.

1.2.1 Undirected graphs

Undirected graph is represented using `Graph` class.

```
G = Graph({0:[1,2,3], 4:[0,2], 6:[1,2,3,4,5]})
```

There are many methods to access the graph properties. For example, to get a list of vertices use `vertices` method.

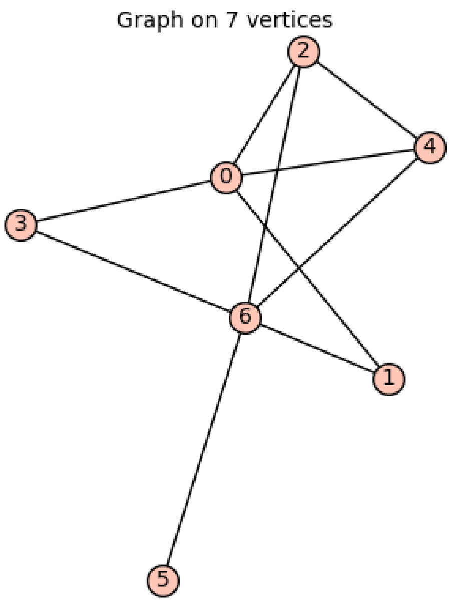
```
G.vertices()
```

```
[0,1,2,3,4,5,6]
```

To display the graph, simply execute a cell with the graph variable name.

```
G
```

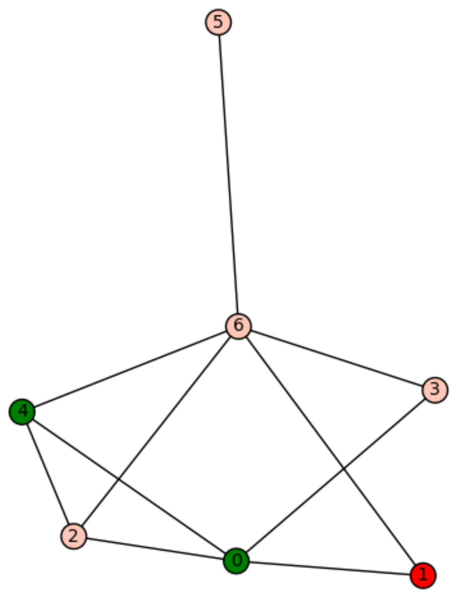
The output is a graphical representation of the graph. If we do not specify vertex coordinates (see below), Sage will use a spring embedder layout algorithm to compute the coordinates.



If a graph is too large, it will not be displayed. In this case, or if you need to specify other display options, you can use the `plot` method. There are many options for the `plot` method, see https://doc.sagemath.org/html/en/reference/plotting/sage/graphs/graph_plot.html for details.

For example. we can specify vertex colors using a dictionary, where keys are colors and values are lists of vertices.

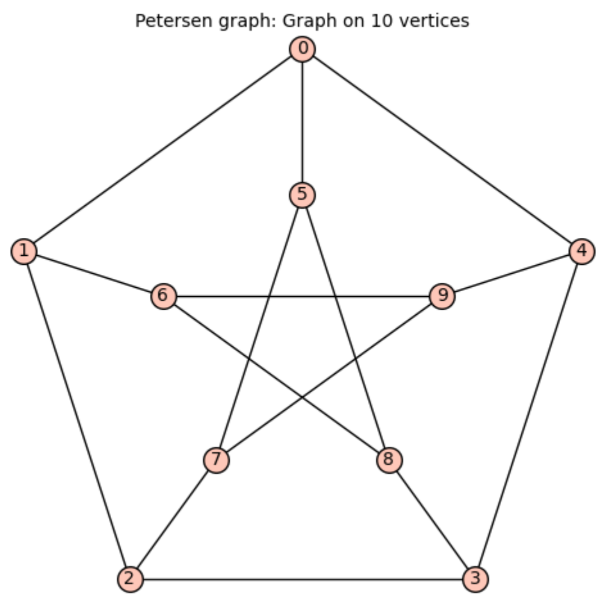
```
G.plot(vertex_colors={'red':[1],'green':[0,4]})
```



1.2.1.1 Some well-known graphs and graph families

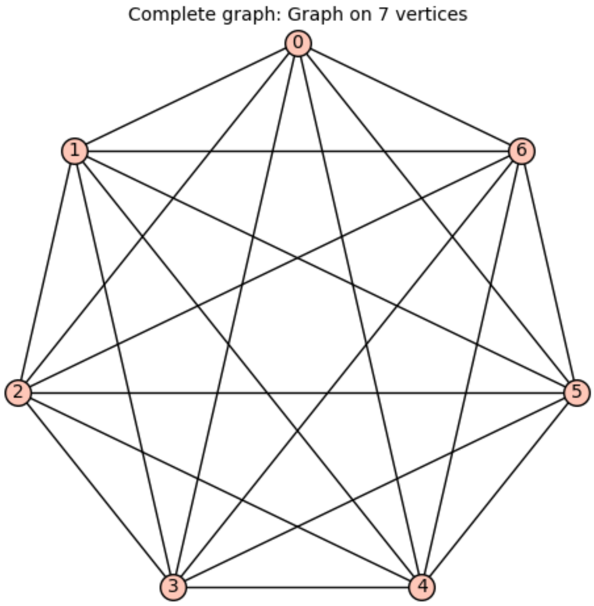
The famous Petersen graph.

```
graphs.PetersenGraph()
```



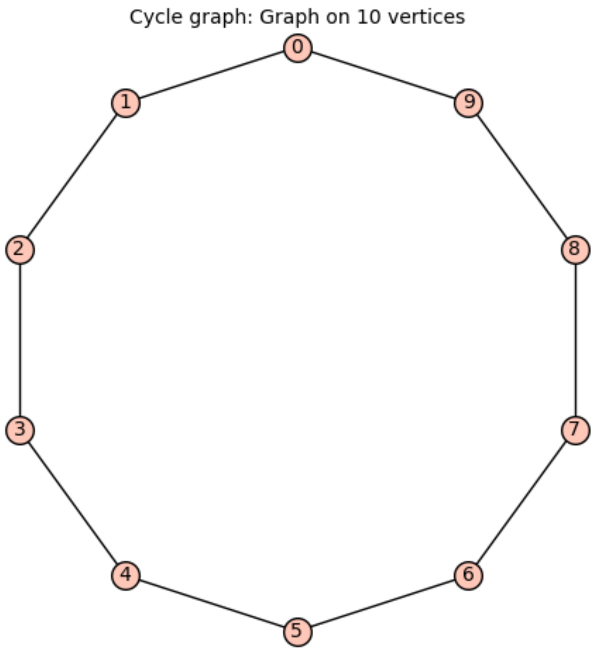
Complete graphs K_n.

```
graphs.CompleteGraph(7)
```



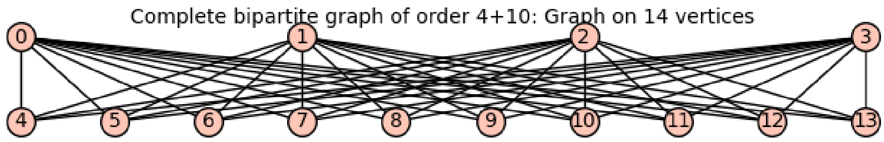
Cycle graphs C_n.

```
graphs.CycleGraph(10)
```



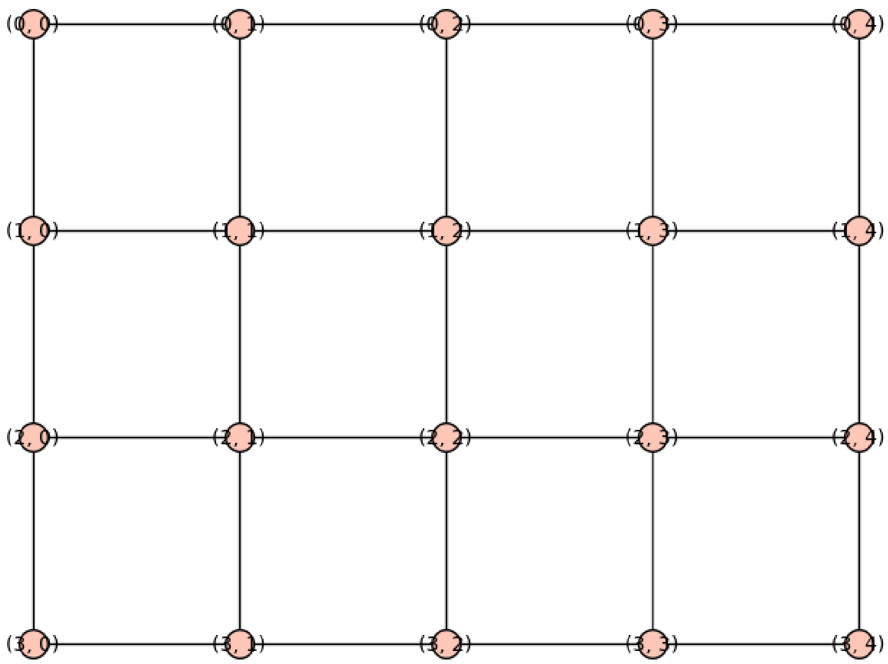
Complete bipartite graphs $K_{n,m}$.

```
graphs.CompleteBipartiteGraph(4, 10)
```



Grid graphs $G_{n,m}$.

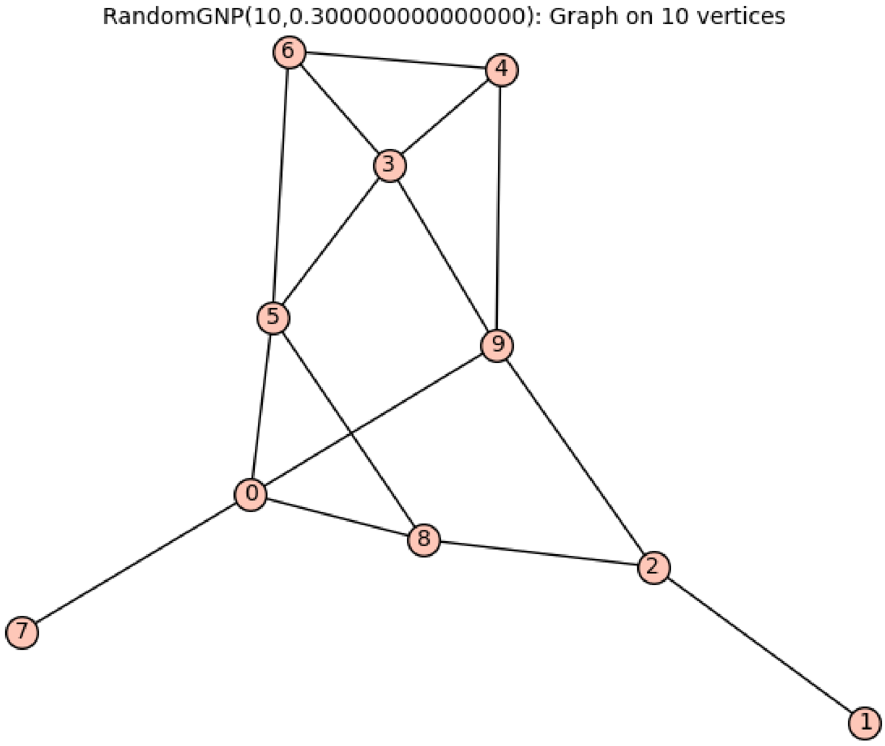
```
GG = graphs.GridGraph([4, 5])
GG.plot()
```



1.2.1.2 Randomly generated graphs

Random graph on 10 nodes. Each edge is inserted independently with probability 0.3.

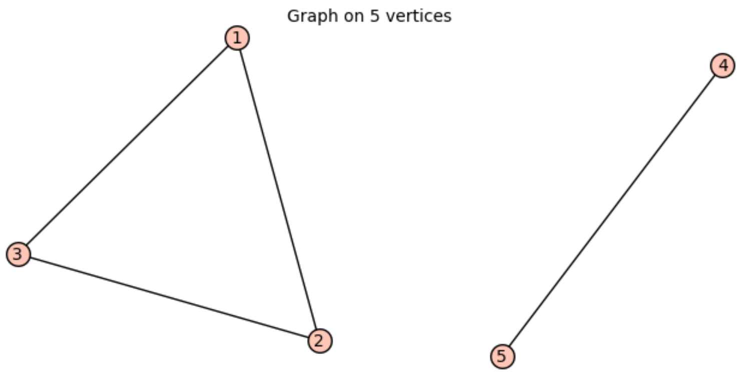
```
graphs.RandomGNP(10, 0.3)
```



1.2.1.3 Graph constructors

From a list of edges.

```
Graph([(1,2),(2,3),(3,1),(4,5)])
```

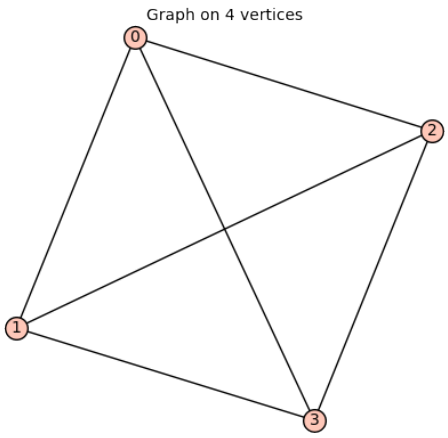


From an adjacency matrix.

```
m = matrix([[int(i != j) for i in range(4)] for j in range(4)])
m
```

```
[0 1 1 1]
[1 0 1 1]
[1 1 0 1]
[1 1 1 0]
```

```
Graph(m)
```



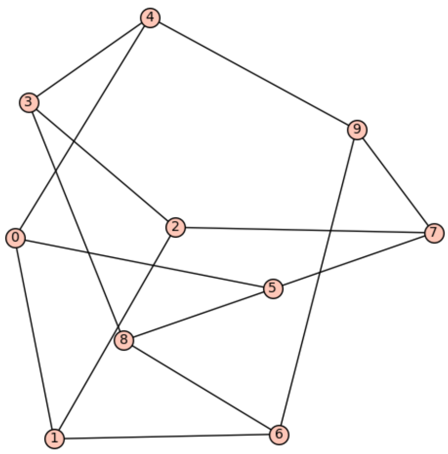
Graph to adjacency matrix.

```
M = G.adjacency_matrix()
m
```

```
[0 1 1 1 1 0 0]
[1 0 0 0 0 0 1]
[1 0 0 0 1 0 1]
[1 0 0 0 0 0 1]
[1 0 1 0 0 0 1]
[0 0 0 0 0 0 1]
[0 1 1 1 1 1 0]
```

From/to graph6 format (compressed string representation of a graph).

```
G = Graph('IheA@GUAo')
G.plot()
```



```
G.graph6_string()
```

```
'IheA@GUAo'
```

Query a graph from local database http://doc.sagemath.org/html/en/reference/graphs/sage/graphs/graph_database.html. For example to get a list of all graphs on 7 vertices with diameter 5.

```
Q = GraphQuery(display_cols=['graph6'], num_vertices=7, diameter=5)
Q.show()
```

```
Graph6
--------------------
F?`po
F?gqg
F@?]O
F@OKg
F@R@o
FA_pW
FEOhW
FGC{o
FIAHo
```

1.2.2 Basic graph manipulation

```
G = Graph({0:[1,2,3], 4:[0,2], 6:[1,2,3,4,5]});
```

Access edges, verices, neighbors, etc.

Access edges.

```
G.edges(labels=False)
```

```
[(0,1),(0,2),(0,3),(0,4),(1,6),(2,4),(2,6),(3,6),(4,6),(5,6)]
```

Note: Edges can have labels. To get a list of edges without labels, use `labels=False` option. Without this option we get

```
[(0,1,None),(0,2,None),(0,3,None),(0,4,None),(1,6,None),(2,4,None),
(2,6,None),(3,6,None),(4,6,None),(5,6,None)]
```

To check if there is an edge between two vertices use

```
G.has_edge(1,2)
```

```
False
```

Access vertices.

```
G.vertices()
```

```
[0,1,2,3,4,5,6]
```

Access neighbors of a vertex.

```
G.neighbors(0)
```

```
[1,2,3,4]
```

Degree of a vertex is a number of its neighbors

```
G.degree(0)
```

```
4
```

To list degrees of all vertices use

```
G.degree()
```

```
[4,3,3,2,3,2,5]
```

Access number of vertices, edges

```
[G.num_verts(),G.num_edges()]
```

```
[7,10]
```

Add/remove vertices, edges

Add a vertex. Note that the vertices of a graph can be any *hashable* objects, not just integers.

```
G.add_vertex('a')
```

Method `add_vertex` without arguments adds a single vertex with the smallest available label.

```
newv = G.add_vertex()
newv
```

```
7
```

```
G.vertices(sort=False)
```

```
['a',7,0,1,2,3,4,5,6]
```

Note that in certain versions of Sage sorting of vertices by some methods (e.g. `vertices`) is enabled by default and they may fail if the vertices are not comparable. To disable sorting use `sort=False` option.

To add multiple vertices use `add_vertices` method.

```
H=Graph({0:[1,2,3],4:[0,2],6:[1,2,3,4,5]})
H.add_vertices(range(10,20))
H.vertices()
```

```
[0,1,2,3,4,5,6,10,11,12,13,14,15,16,17,18,19]
```

To add one edge use `add_edge` method, to add multiple edges use `add_edges` method.

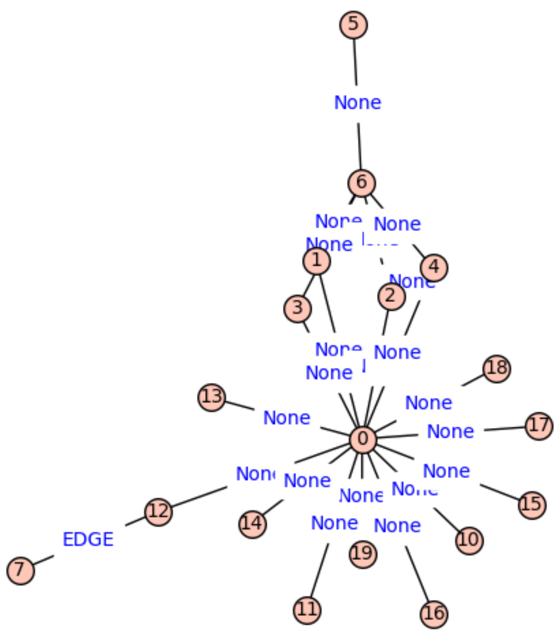
```
H.add_edges([(0, i) for i in range(10, 20)])
```

Note that edges can have labels. To add an edge with a label you need to pass a a triple $u, v, label$ as an argument to `add_edge` method.

```
H.add_edge(7,12,"EDGE")
```

To plot a graph with edge labels use `edge_labels=True` option.

```
H.plot(edge_labels=True)
```



Note that adding an existing vertex (edge) does not result in an error or a warning.

To delete a vertex or and edge use `delete_vertex` and `delete_edge` methods, respectively. For example:

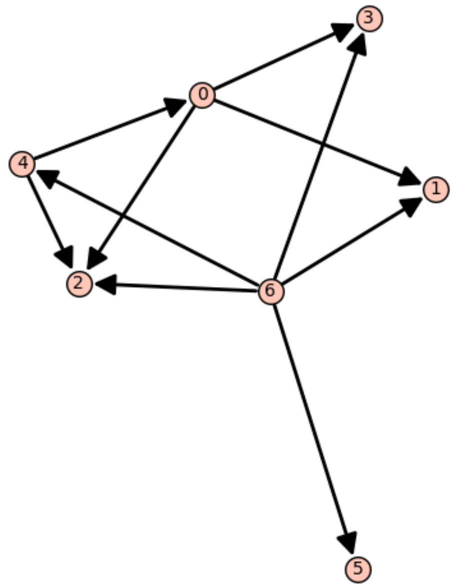
```
H.delete_vertex(7)
H.delete_edge(0,10)
```

Note that deleting a non-existing vertex results in an error while deleting a non-existing edge does not.

1.2.3 Directed graphs

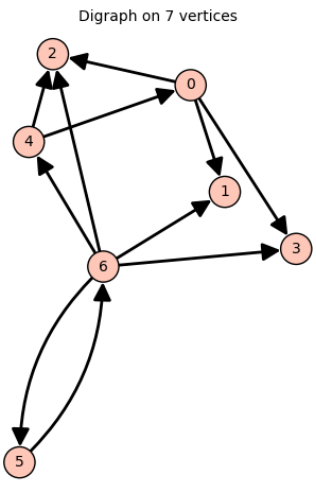
Directed graph is represented using `DiGraph` class.

```
D = DiGraph({0:[1,2,3],4:[0,2],6:[1,2,3,4,5]})
D.plot()
```



Most of the methods for `Graph` class have their counterparts for `DiGraph` class. For example, to add an edge use `add_edge` method.

```
D.add_edge(5,6)
D.plot()
```



Specific methods for `DiGraph` class include `in_degree` and `out_degree` methods to get in-degree and out-degree of a vertex, respectively. Similarly, in addition to `neighbors` there are `in_neighbors` and `out_neighbors` methods.

```
[D.in_degree(0),D.out_degree(0),degree(0)]
```

```
[1,3,4]
```

```
[D.in_neighbors(0),D.out_neighbors(0),D.neighbors(0)]
```

```
[[4],[1,2,3],[1,2,3,4]]
```

To check connectivity of a directed graph use `is_strongly_connected` method.

```
[D.is_connected(),D.is_strongly_connected()]
```

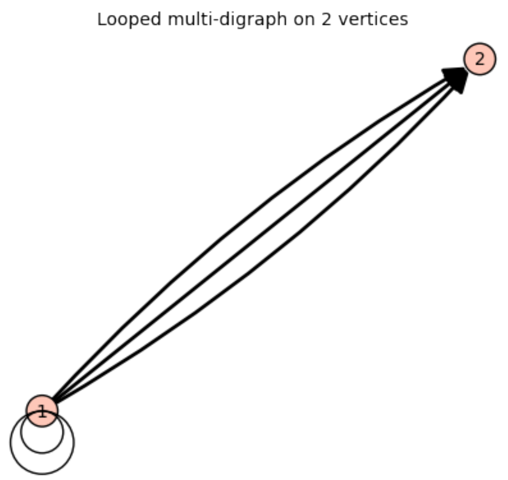
```
[True,False]
```

To convert a directed graph to an undirected graph use `to_undirected` (or `to_simple`) method.

Even more general graphs

To allow multiple edges and/or loops use options `multiedges=True` and `loops=True` to the `DiGraph` constructor. For example, consider the following graph.

```
MG = DiGraph({},multiedges=True,loops=True)
MG.add_vertices([1,2])
MG.add_edges([(1,2),(1,2),(1, 2),(1,1),(1,1)])
MG
```



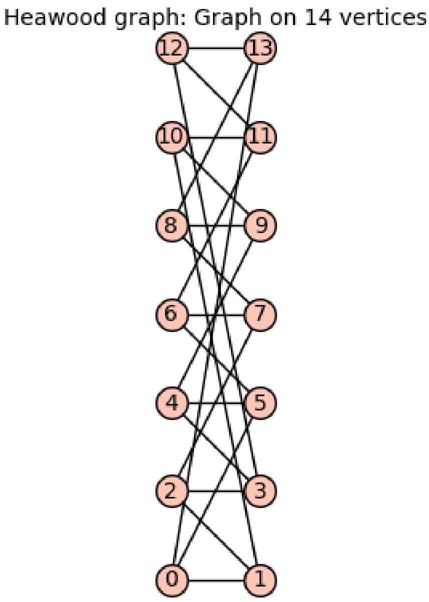
1.2.4 Exercises

Exercise 1.1 Write a function `remove_max_vertex(G)` which removes a vertex with the largest degree from undirected graph G (any of them, if there are more than one with the largest degree). ■

Exercise 1.2 Write a function `plot_bipartite` which plots a bipartite graph in a way that vertices of each bipartition are arranged on two parallel lines. ■

For example:

```
HGR=graphs.HeawoodGraph()
plot_bipartite(HGR)
```

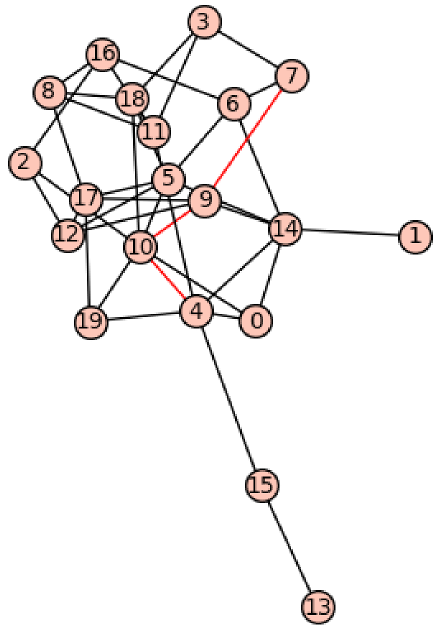


Exercise 1.3 Write a function `set_random_edge_labels(G,a,b)` which sets edge labels of G to random integers from interval $[a,b]$.

Write a function `mark_shortest_path(G,a,b)` which calculates a shortest path between the vertices a and b in the weighted graph G and colors it with red color. (For calculating shortest paths use built-in function `shortest_path`. ■

Example:

```
X=graphs.RandomGNP(20,0.2)
set_random_edge_labels(X,1,10)
mark_shortest_path(X,4,7)
```


3
16
7
8
18
6
11
2
5
17
9
14
1
12
10
4
0
19
15
13

2. Depth-first search and Breadth-first search

2.1 Depth-first search (DFS)

Write a Depth-first search (DFS) implementation using Sage Graph representation

- Write a recursive implementation of the depth-first search.
- Add computation of discovery and finishing times to the implementation.

(See Handouts on Course Homepage for pseudocode)

```
def DFS_recursive(G, r):
    """
    Perform DFS from root r. Result is a dictionary mapping a vertex v to
    its predecessor in DFS tree (root is mapped to None).
    """
    prev = {}
    prev[r] = None
    DFS_recursive_call(G, r, prev)
    return prev

def DFS_recursive_call(G, v, prev):
    for u in G.neighbors(v):
        if u not in prev:
            prev[u] = v
            DFS_recursive_call(G, u, prev)
```

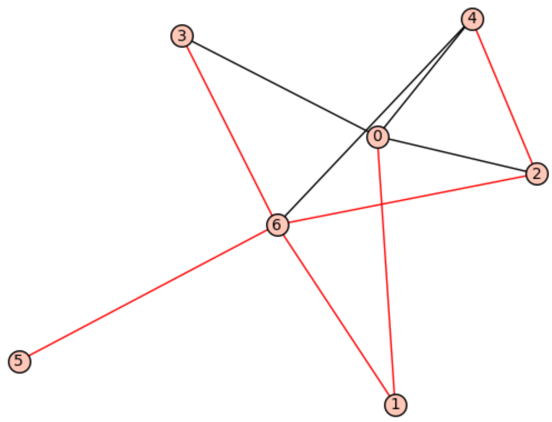
Examples

```
G = Graph({0:[1,2,3], 4:[0,2], 6:[1,2,3,4,5]})

dfs_dict = DFS_recursive(G, 0)
dfs_dict
```

```
{0: None, 1: 0, 6: 1, 2: 6, 4: 2, 3: 6, 5: 6}
```

```
G.plot(edge_colors={'red': [(u, v) for (u, v) in dfs_dict.items() if v
!= None]})
```




```
H = graphs.Grid2dGraph(3, 3)
DFS_recursive(H, (0, 0))
```

```
{(0, 0): None,
 (0, 1): (0, 0),
 (0, 2): (0, 1),
 (1, 2): (0, 2),
 (1, 1): (1, 2),
 (1, 0): (1, 1),
 (2, 0): (1, 0),
 (2, 1): (2, 0),
 (2, 2): (2, 1)}
```

2.2 DFS with start (discovery) time and end (finishing) time

```
def DFS_with_times(G, r):
    """
    Perform DFS from root r. Result is a triple of three dictionaries:
    - dictionary mapping a vertex v to its predecessor in DFS tree
      (root is mapped to None).
    - dictionary mapping a vertex to its start time
    - dictionary mapping a vertex to its end time
    """
    global time
    time = 0
    prev = {}
    start = {}
    end = {}
    prev[r] = None
    DFS_with_times_call(G, r, prev, start, end)
    return (prev, start, end)

def DFS_with_times_call(G, v, prev, start, end):
    global time
    time += 1;
    start[v] = time;
    for u in G.neighbors(v):
        if u not in prev:
            prev[u] = v
```

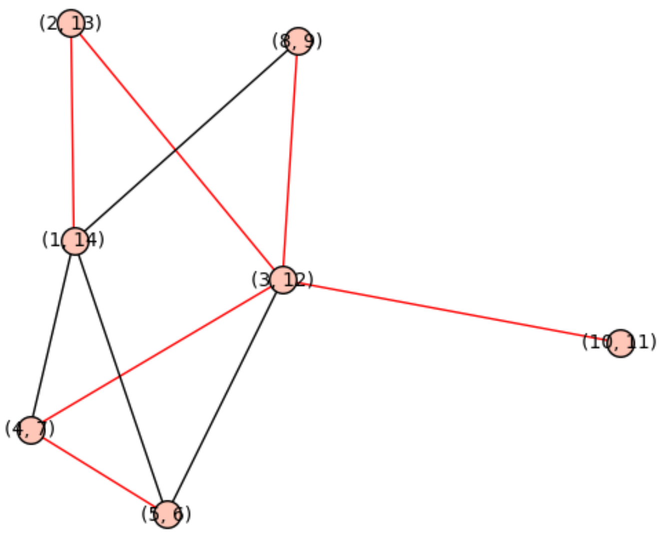
```
            DFS_with_times_call(G, u, prev, start, end)
    time += 1;
    end[v] = time;
```

Examples

```
G = Graph({0:[1,2,3], 4:[0,2], 6:[1,2,3,4,5]})
(prev, disc, finish) = DFS_with_times(G, 0)
(prev, disc, finish)
```

```
({0: None, 1: 0, 2: 6, 3: 6, 4: 2, 5: 6, 6: 1},
 {0: 1, 1: 2, 2: 4, 3: 8, 4: 5, 5: 10, 6: 3},
 {0: 14, 1: 13, 2: 7, 3: 9, 4: 6, 5: 11, 6: 12})
```

```
G.relabel(dict([(v, (disc[v], finish[v])) for v in G.vertices()]))
G.plot(edge_colors={'red': [((disc[u],finish[u]), (disc[v],finish[v]))
  for (u, v) in prev.items() if v != None]})
```



2.3 Breadth-first search (BFS)

Write a Breadth-first search (BFS) implementation using Sage Graph representation.

```
import queue
def BFS(G, r):
    """
    Perform BFS from root r. Result is a dictionary mapping a vertex v
    to its predecessor in BFS tree (root is mapped to None).
    """
    prev = {}
    prev[r] = None
    q = queue.Queue()
    q.put(r)
    while not q.empty():
        v = q.get()
```

```
        for u in G.neighbors(v):
            if u not in prev:
                prev[u] = v
                q.put(u)
    return prev
```

Example

```
BFS(H, (0, 0))
```

```
{(0, 0): None,
 (0, 1): (0, 0),
 (1, 0): (0, 0),
 (0, 2): (0, 1),
 (1, 1): (0, 1),
 (2, 0): (1, 0),
 (1, 2): (0, 2),
 (2, 1): (1, 1),
 (2, 2): (1, 2)}
```

2.4 Topological sorting

- Use DFS with discovery and finishing times to implement topological sorting of a DAG (directed acyclic) graph
- Help professor Bumstead to dress himself in the correct order. Order of putting his garments is given by the digraph below

```
T = DiGraph({'undershorts': ['shoes','pants'],'pants':['shoes','belt'],
'belt':['jacket'],'shirt':['belt','tie'],'tie':['jacket'],'socks':['
shoes'],'watch':[]})
```

```
def DFS_DiGraph(G):
    """
    Implement (recursive) DFS on a digraph to create a
    "forest of DFS trees"

    Use G.neighbors_out(v) to get "out" neighbors of vertex v
    """
    global time
    time = 0
    prev = {}
    start = {}
    end = {}
    for v in G.vertices(sort=False):
        if v not in prev:
            prev[v] = None
            DFS_DiGraph_call(G, v, prev, start, end)
    return (prev, start, end)

def DFS_DiGraph_call(G, v, prev, start, end):
    global time
    time += 1;
    start[v] = time;
    for u in G.neighbor_out_iterator(v):
        if u not in prev:
            prev[u] = v
            DFS_DiGraph_call(G, u, prev, start, end)
```

```
    time += 1
    end[v] = time
```

```
DFS_DiGraph(T)
```

```
({'belt': None,
  'jacket': 'belt',
  'tie': None,
  'watch': None,
  'shoes': None,
  'socks': None,
  'pants': None,
  'undershorts': None,
  'shirt': None},
 {'belt': 1,
  'jacket': 2,
  'tie': 5,
  'watch': 7,
  'shoes': 9,
  'socks': 11,
  'pants': 13,
  'undershorts': 15,
  'shirt': 17},
 {'jacket': 3,
  'belt': 4,
  'tie': 6,
  'watch': 8,
  'shoes': 10,
  'socks': 12,
  'pants': 14,
  'undershorts': 16,
  'shirt': 18})
```

```
def topological_sort(G):
    """
    Performs topological sort on a DAG (directed acyclic graph) G
    (calculate finishing times and sort vertices by them in
    descending order)
    """
    (_, _, finish) = DFS_DiGraph(T)
    return sorted(finish.items(), key=lambda x: -x[1])
```

```
topological_sort(T)
```

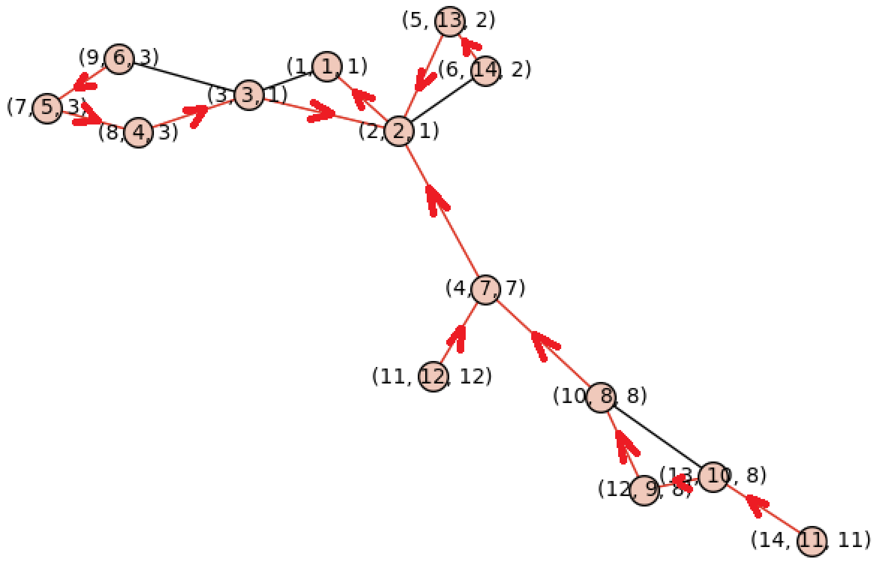
```
[('shirt', 18),
 ('undershorts', 16),
 ('pants', 14),
 ('socks', 12),
 ('shoes', 10),
 ('watch', 8),
 ('tie', 6),
 ('belt', 4),
 ('jacket', 3)]
```

3. Low value and 2-connected components

3.1 Low value

For a vertex v, $\mathrm{low}(v)$ is the smallest discovery time, $\mathrm{disc}(x)$, over all vertices which can be reached from v using tree edges (away from root) – red edges – and at most one back edge – black edge.

In the example below labels of vertices are (vertex name, discovery time, low value) and arrows indicate parent of a vertex (prev).



$\mathrm{low}(2)$ is 1 since we can reach the root (with discovery time 1) using red edge $(2,3)$ and black edge $(3,1)$.

$\mathrm{low}(8)$ is 3 since the vertex with the smallest discovery time we can reach in the prescribed way is 3: edges are $(8,7),(7,9),(9,3)$ and 3 has discovery time 3.

$\mathrm{low}(10)$ is 8 (its discovery time) since we can not reach any vertex with smaller discovery time using the tree edges "below" 10.

Use a recursive implementation of the depth-first search given in the previous chapter to compute the low value of each vertex in a graph.

```
def DFS_low(G, r):
    """
    Calculate DFS with root r, discovery time, low values.
    """
    global time
    time = 0
    prev = {}
    disc = {}
    low = {}
    prev[r] = None
    DFS_low_call(G, r, prev, disc, low)
    return (prev, disc, low)

def DFS_low_call(G, v, prev, disc, low):
    global time
    time += 1;
    disc[v] = time;
    low[v] = time;
    for u in G.neighbors(v):
        if u not in prev:
            prev[u] = v
            DFS_low_call(G, u, prev, disc, low)
    for u in G.neighbors(v):
        if prev[u] == v:
            # edge (vertex) in "subtree"
            low[v] = min(low[v], low[u])
        elif u != prev[v]:
            # "back edge" and not a tree edge
            low[v] = min(low[v], disc[u])
```

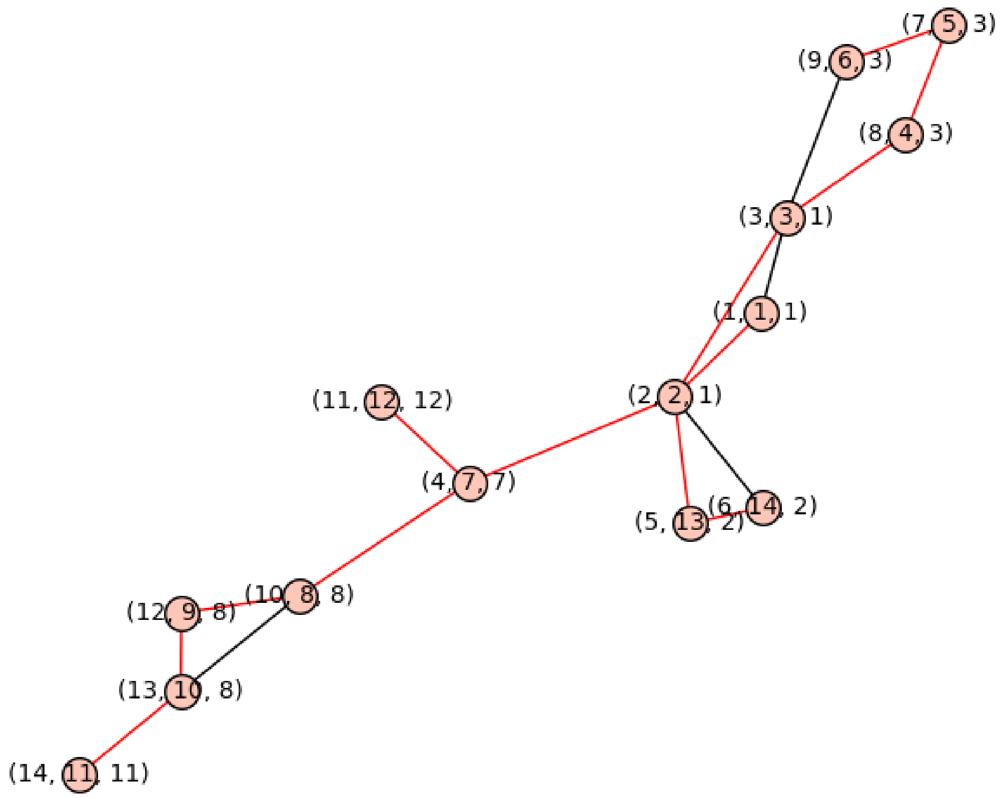
Example

```
G = Graph({1:[2,3], 2:[3,4,5,6], 3:[8,9], 4:[10,11], 5:[6], 7:[8,9],
10:[12,13], 12:[13], 13:[14]})
(prev, disc, low) = DFS_low(G, 1)
low
```

```
{1: 1,
 2: 1,
 3: 1,
 8: 3,
 7: 3,
 9: 3,
 4: 7,
 10: 8,
 12: 8,
 13: 8,
 14: 11,
 11: 12,
 5: 2,
 6: 2}
```

Relabel vertices with triples (vertex label, discovery time, low value) and color tree edges red

```
G1 = G.relabel(dict([(v, (v, disc[v], low[v])) for
    v in G.vertices(sort=False)]), inplace=False)
G1.plot(edge_colors={'red': [((u, disc[u], low[u]), (v, disc[v],
    low[v])) for (u, v) in prev.items() if v != None]})
```



3.2 Cutvertices

We can get cutvertices using the following Theorem:

Theorem 3.1 Let G be connected, undirected, simple, let r be the root of its DFS tree T:

- r is a cutvertex if it is incident with at least 2 tree edges
- nonroot vertex v is a cutvertex if v has a son y so that $\text{low}(y) \geq \text{disc}(v)$

In the example above, cutvertices are $2, 3, 4, 10, 13$. For example, 10 is a cutvertex, since its son in the tree has low value 8 which is $\geq$ than discovery time of 10, which is 8.

Also, 4 is a cutvertex since its sons (11 and 10) have low values ≥ 7 $(7 = \text{disc}(4))$.

The root 1 is not a cutvertex since it is incident with only one tree edge.

```
def cutvertices(G):
    """
    Retuns an array of cutvertices of a connected graph G.
    """
    root = G.vertices(sort=False)[0]      # assume G is connected
    (prev, start, low) = DFS_low(G, root)
    result = []
    rootn = 0
    for v in G.vertices(sort=False):
        for u in G.neighbors(v):
            if v != root:
                if v == prev[u] and low[u] >= start[v]:
                    result.append(v)
                    break
```

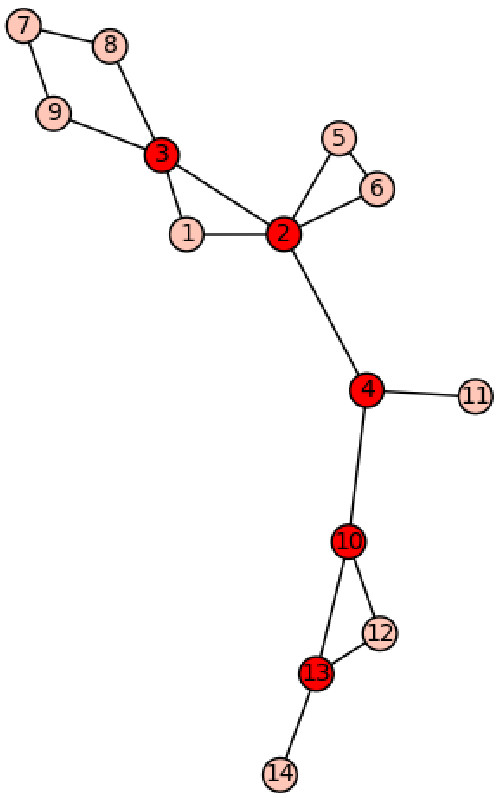
```
            elif v == prev[u]:
                rootn += 1
    if rootn >= 2:
        result.append(root)
    return result
```

Example

```
cutvertices(G)
```

```
[2, 3, 4, 13, 10]
```

```
plot(G, vertex_colors={'red': cutvertices(G)})
```



3.3 2-connected components

Write a function partition(G) which partitions edges of G into blocks (2-connected components).

Output should be a dictionary which maps an edge of the graph into a number which represents a block. In the example above, vertices $1,2,3$ (edges $(1,2),(1,3),(2,3)$) create a block. Therefore the resulting dictionary should map the pairs $(1,2),(1,3),(2,3)$ into the same number, say 1.

```
def partition(G):
    """
    Partitions of edges of a connected graph G into blocks.
```

```
    Returns a dictionary mapping each edge to the block (number) it belongs
     to.
    """
    global blocknum
    root = G.vertices(sort=False)[0]      # assume G is connected
    (prev, start, low) = DFS_low(G, root)
    blocknum = 0
    blocks = {}
    partition_call(G, root, prev, start, low, blocks, 0)
    return blocks

def partition_call(G, v, prev, start, low, blocks, blockn):
    global blocknum
    for u in G.neighbors(v):
        if v == prev[u]: # forward tree edge
            if low[u] >= start[v]: # cut vertex, start a new block
                blocknum += 1
                blocks[(v, u)] = blocknum
                partition_call(G, u, prev, start, low, blocks, blocknum)
            else: # stay in the same block
                blocks[(v, u)] = blockn
                partition_call(G, u, prev, start, low, blocks, blockn)
        elif start[u] < start[v] and u != prev[v]: # back edge not in tree
            blocks[(u, v)] = blockn
```

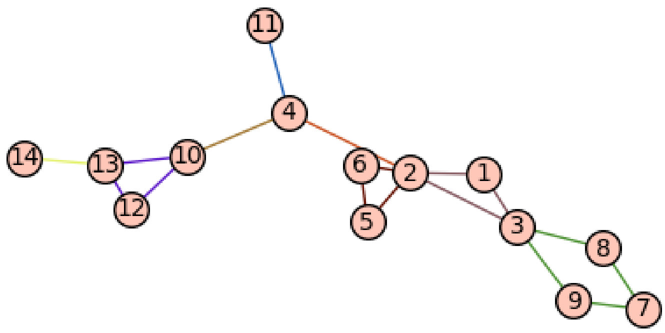
Example

```
    partition(G)
```

```
    {(10, 4): 1,
     (4, 2): 2,
     (2, 1): 3,
     (1, 3): 3,
     (2, 3): 3,
     (3, 8): 4,
     (8, 7): 4,
     (7, 9): 4,
     (3, 9): 4,
     (2, 5): 5,
     (5, 6): 5,
     (2, 6): 5,
     (4, 11): 6,
     (10, 12): 7,
     (12, 13): 7,
     (10, 13): 7,
     (13, 14): 8}
```

```
import random
def edge_colors(part):
    blocks = set(part.values())
    colors = [(random.random(), random.random(), random.random()) for b in
   blocks]
    colorblocks = [[edge for edge in part.keys() if part[edge] == b] for b
   in blocks]
    return dict(zip(colors, colorblocks))
```

```
    G.plot(edge_colors=edge_colors(partition(G)))
```



11
4
14
13
10
12
6
2
1
5
3
8
9
7

4. Shortest Hamiltonian cycle (Travelling salesman problem)

The travelling salesman problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

We will assume that there are roads (edges) between all cities (complete graph) and that the distances are Euclidean distances (Euclidean TSP).

We will write an approximation algorithm for TSP, which will be based on the minimum spanning tree (MST) algorithm.

4.1 Approximation

Implement the following 2-approximation algorithm (that means that the length of our solution will be better than 2 times the length of an optimal solution).

1. Find minimal spanning tree of our graph (use built-in Sage function `min_spanning_tree`).
2. Run DFS on this tree.
3. Take vertices in the order of increasing (DFS) start time.

```
def TSP_approximation(G):
    """
    Returns Hamiltonian cycle (travelling salesman circuit) using a 2-
   approximation algorithm.
    """
    mst = G.min_spanning_tree(by_weight=True)
    T = Graph(mst) # graph (tree) from edges

    # DFS with times
    r = T.vertices(sort=False)[0]
    _, start, _ = DFS_with_times(T, r)
    sort_start = sorted(list(start.items()), key=lambda p: p[1])
    cycle = []
    length = 0
    sort_start.append(sort_start[0])
    for i in range(1, len(sort_start)):
        u = sort_start[i - 1][0]
        v = sort_start[i][0]
        cycle.append((u, v))
        length += G.edge_label(u, v)
    return cycle, length
```

Example

Create the test example, a complete graph with 10 vertices with given vertex coordinates.

```
def distance(a, b):
    """
    Return Euclidean distance between a = (ax, ay) and b = (bx, by)
    """
    ax, ay = a
    bx, by = b
    return math.sqrt((bx - ax)**2 + (by - ay)**2)

def set_euclidean_distances(G):
    """
    Set Euclidean distances as edge weights (labels)
    """
    pos = G.get_pos()
    for (u, v) in G.edges(sort=False, labels=False):
        G.set_edge_label(u, v, distance(pos[u], pos[v]))

H = graphs.CompleteGraph(10)
H.set_pos({0: [8, 1], 1: [0, 8], 5: [1, 0], 2: [5, 3], 3: [1.5, 7], 4: [2,
    4],
  6: [6, 2], 7: [3, 1], 8: [2, 2], 9: [3, 3]})
set_euclidean_distances(H)
```

```
    cycle, length = TSP_approximation(H)
```

```
    cycle, length
```

```
   ([(0, 6),
     (6, 2),
     (2, 9),
     (9, 4),
     (4, 3),
     (3, 1),
     (1, 8),
     (8, 5),
     (5, 7),
     (7, 0)], 27.70534328046342)
```

Compare with the optimal solution, computed using the built-in Sage function `traveling_salesman_problem`.

```
    def cycle_length(cycle, G):
        length = 0
        for u, v in cycle:
            length += G.edge_label(u, v)
        return length

    opt_cycle = H.traveling_salesman_problem(use_edge_labels=True)
    opt_length = cycle_length(opt_cycle.edges(sort=False, labels=False),
      opt_cycle)
    opt_length
```

```
    27.540829118717557
```

4.2 Iterative improvement

4.2.1 2-changes on intersecting segments

Improve a solution iteratively using 2-changes on *intersecting* segments.

A *2-change* is a transformation of a cycle by removing two non-consecutive edges and adding two other edges such that the resulting graph is still a cycle.

```
def iterate_2_changes_intersecting(cycle, G, n=1000):
    """
    Iterate by eliminating intersections by 2-changes. Make at most n
   iterations
    """
    for k in range(n):
        inter = find_intersection(cycle, G)
        if inter == None:
            return cycle
        i, j = inter
        cycle = perform_2_change(cycle, i, j)
    return cycle
```

(See the code for `find_intersection` and `perform_2_change` at the end of this chapter.)

Example

```
    cycle = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8)
   , (8, 9),
     (9, 0)]
    new_cycle = iterate_2_changes_intersecting(cycle, H, 10)
    (cycle_length(new_cycle, H), cycle_length(cycle, H))
```

```
    (30.367268577721603, 46.94180782091779)
```

4.2.2 2-changes on random edges

Improve a solution iteratively using 2-changes on *random* non-adjacent cycle edges.

```
def iterate_2_changes(cycle, G, n):
    min_length = cycle_length(cycle, G)
    min_cycle = cycle
    for k in range(n):
        i = randint(0, len(min_cycle) - 1)
        add = randint(2, len(min_cycle) - 2)
        j = (i + add) % len(min_cycle)
        new_cycle = perform_2_change(min_cycle, i, j)
        new_length = cycle_length(new_cycle, G)
        if new_length < min_length:
            min_length = new_length
            min_cycle = new_cycle
    return min_cycle
```

Example

```
    new_cycle = iterate_2_changes(cycle, H, 50000)
    (cycle_length(new_cycle, H), cycle_length(cycle, H))
```

```
(27.669330960262805, 46.94180782091779)
```

```
(cycle_length(new_cycle, H), opt_length)
```

```
(27.669330960262805, 27.540829118717557)
```

4.2.3 Code of auxiliary functions

```
def perform_2_change(cycle, i, j):
    """
    Perform a 2-change on a (hamiltonian) cycle for edges with
    (non-consecutive) indices i and j. Cycle is a list of edges
    """
    if i > j:
        i, j = j, i
    e1 = cycle[i]
    e2 = cycle[j]
    v1, u1 = e1
    v2, u2 = e2
    result = []
    revert = False
    for k in range(i):
        result.append(cycle[k])
    result.append((v1, v2))
    for k in reversed(range(i + 1, j)):
        result.append(tuple(reversed(cycle[k])))
    result.append((u1, u2))
    for k in range(j + 1, len(cycle)):
        result.append(cycle[k])
    return result
```

Intersection of two segments.

```
def find_intersection(cycle, G):
    """
    Find indices of two non-consecutive cycle edges which intersect and
   None if there are none
    """
    pos = G.get_pos()
    for i in range(len(cycle)):
        ei = cycle[i]
        l = len(cycle) if i > 0 else len(cycle) - 1
        for j in range(i + 2, l):
            ej = cycle[j]
            if do_intersect(pos[ei[0]], pos[ei[1]], pos[ej[0]], pos[ej[1]])
   :
                return (i, j)
    return None

def on_segment(p, q, r):
    if ((q[0] <= max(p[0], r[0])) and (q[0] >= min(p[0], r[0])) and
        (q[1] <= max(p[1], r[1])) and (q[1] >= min(p[1], r[1]))):
        return True
    return False

def orientation(p, q, r):
    val = (float(q[1] - p[1]) * (r[0] - q[0])) - (float(q[0] - p[0]) * (r
   [1] - q[1]))
```

```python
    if (val > 0):
        return 1
    elif (val < 0):
        return 2
    else:
        return 0

# Check if two segments intersect
# https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
def do_intersect(p1, q1, p2, q2):
    o1 = orientation(p1, q1, p2)
    o2 = orientation(p1, q1, q2)
    o3 = orientation(p2, q2, p1)
    o4 = orientation(p2, q2, q1)

    if ((o1 != o2) and (o3 != o4)):
        return True
    if ((o1 == 0) and on_segment(p1, p2, q1)):
        return True
    if ((o2 == 0) and on_segment(p1, q2, q1)):
        return True
    if ((o3 == 0) and on_segment(p2, p1, q2)):
        return True
    if ((o4 == 0) and on_segment(p2, q1, q2)):
        return True
    return False
```

5. Graph Drawing

In this chapter, we will write *iterative* methods for drawing graphs. General idea is to:

1. Start with a random drawing of a graph
2. Iteratively improve the drawing

5.1 Method 1: Mass center

Write the following functions:

1. `move_vertex_c(G, v, pos)`
   Where G is graph, v is a vertex in G and pos is a dictionary of positions for each vertex. It should move position of v to the mass center of its neighbors, i.e., $\mathrm{pos}(v) = 1/|N(v)|\sum_{u\in N(v)} \mathrm{pos}(u)$.
2. `draw_graph_c(G, F, iters)`
   Where G is graph, F is a list of fixed vertices, iters is a number of iterations. Function should
   a. Draw positions of vertices of F on a circle (with radius 1, i.e., set positions of F to vertices of a regular polygon).
   b. Other vertices, $V(G)\setminus F$, set to random positions in square $[-0.5,0.5]\times[-0.5,0.5]$.
   c. Use function `move_vertex_c` to change the position of each vertex $V(G)\setminus F$.
   d. Repeat Step 3 iters times.

```python
def move_vertex_c(G, v, pos):
    """
    Move vertex v to the mass center of its neighbors.
    """
    sx = 0
    sy = 0
    N = 0
    for u in G.neighbors(v):
        x,y = pos[u]
        sx += x
        sy += y
        N += 1
    if N > 0:
        pos[v] = (sx/N, sy/N)
```

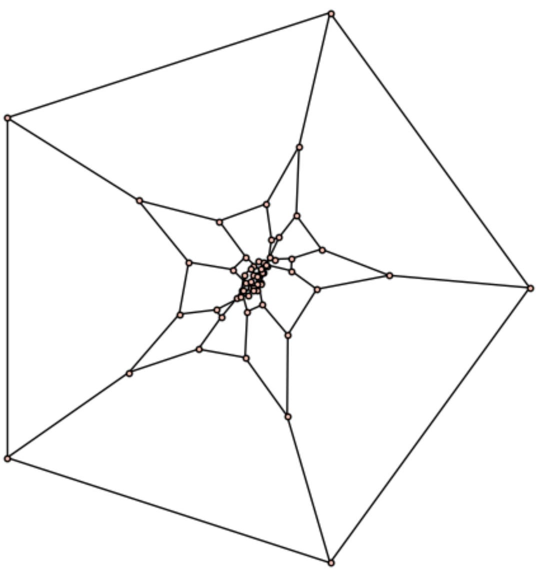
```python
def draw_graph_c(G, F, iters):
    """
```

```
    Draw graph G with fixed vertices F using mass center method.
    """
    pos = {}
    for i in range(len(F)):
        pos[F[i]] = (cos(2*i*math.pi/len(F)), sin(2*i*math.pi/len(F)))
    vert = [v for v in G.vertices(sort=False) if v not in F]
    for v in vert:
        pos[v] = (random() - 0.5, random() - 0.5)
    for i in range(iters):
        for v in vert:
            move_vertex_c(G, v, pos)
    G.set_pos(pos)
    return G.plot(vertex_labels = False, vertex_size = 10)
```

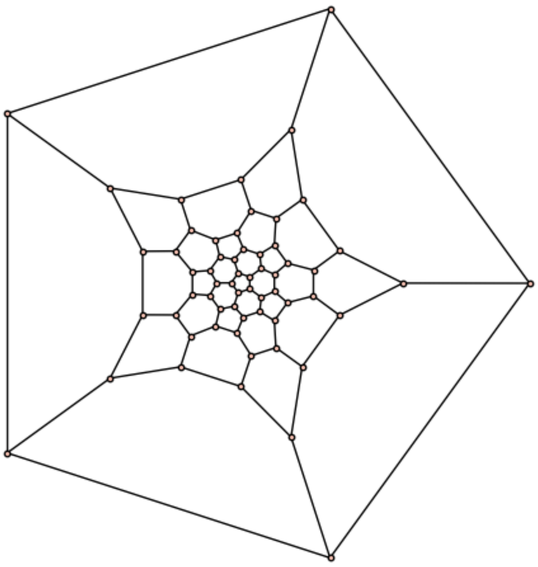
Example:

```
def find_cycle(G0):
    """
    An ad-hoc function to find some cycle in a graph, provided that G is 2-
   connected
    """
    G = G0.copy()
    e = G.edges(sort=False)[0]
    G.delete_edge(e)
    return G.shortest_path(e[0], e[1])
```

```
    G = graphs.BuckyBall()
    draw_graph_c(G, find_cycle(G), 5)
```



```
    G = graphs.BuckyBall()
    draw_graph_c(G, find_cycle(G), 100)
```



5.2 Method 2: Move vertices using force

Write the following functions:

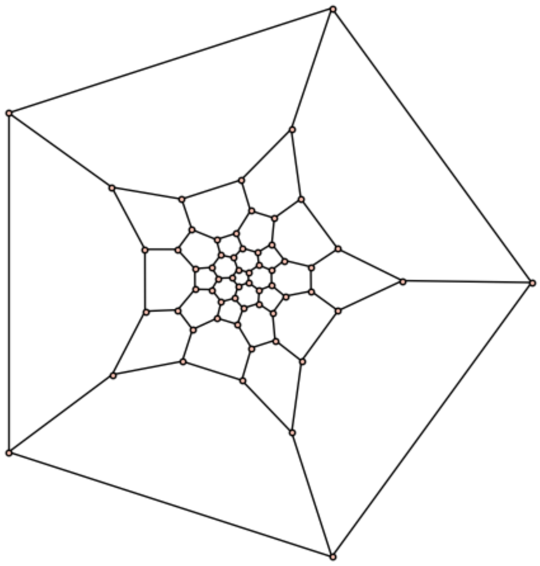
1. `move_vertex_f(G, v, pos, k)`
   Where G is graph, v is a vertex in G, pos is a dictionary of positions for each vertex and k is a constant. Similar to `move_vertex_c`, just use "forces" to move vertex v. Each edge vu (u is a neighbor of v) acts like a "spring" and acts with force $\vec{F} = k\,\vec{\delta}$ where $\vec{\delta} = \vec{u} - \vec{v}$ (Hooke's law) and k is characteristic of the spring (razteznostni koeficient in Slovene). That is: $\mathrm{pos}(v) = \mathrm{pos}(v) + \sum_{u \in N(v)} k(\mathrm{pos}(u) - \mathrm{pos}(v))$.
2. `draw_graph_f(G, F, k, iters)`
   which acts in the same way as `draw_graph_c`, but it uses the function `move_vertex_f` instead of `move_vertex_c`.

```
def move_vertex_f(G, v, pos, k):
    """
    Move vertex v using force method.
    """
    vx, vy = pos[v]
    fx, fy = pos[v]
    for u in G.neighbors(v):
        x,y = pos[u]
        dx = x - vx
        dy = y - vy
        fx += dx * k
        fy += dy * k
    pos[v] = (fx, fy)


def draw_graph_f(G, F, k, iters):
    pos = {}
    for i in range(len(F)):
        pos[F[i]] = (cos(2*i*math.pi/len(F)), sin(2*i*math.pi/len(F)))
    vert = [v for v in G.vertices(sort=False) if v not in F]
    for v in vert:
```

```
        pos[v] = (random() - 0.5, random() - 0.5)
    for i in range(iters):
        for v in vert:
            move_vertex_f(G, v, pos, k)
    G.set_pos(pos)
    return G.plot(vertex_labels = False, vertex_size = 10)
```

```
G = graphs.BuckyBall()
draw_graph_f(G, find_cycle(G), 0.1, 100)
```



5.3 Method 3: Spring embedder

For both methods above we required a cycle to be selected before fixing its coordinates. But this is not "practical". Can we do without this? Without fixing some vertices, and using only the (attractive) force method, the vertices of the graph will eventually move to a single point. So we need to add *repulsive* forces.

1. `move_vertex_se(G, v, pos, k, e)`
   Similar to `move_vertex_f`, each edge vu acts like a "spring" and acts with force $\vec{F} = k\,\vec{\delta}$ where $\vec{\delta} = \vec{u} - \vec{v}$. Additionally: vertices also act in a repulsive way with force $\vec{R} = -e\,\vec{\delta}/|\vec{\delta}|^2$ for all $u \neq v$. With the repulsive force we do not allow two vertices to be too close, since the force is inversely proportional to the square of the distance between them!
2. `draw_graph_se(G, k, e, iters)`
   Similar to `draw_graph_f`, just use `move_vertex_se` instead of `move_vertex_f`. Note that there are no fixed vertices. Initially, for each vertex, choose a random position in the square $[-0.5, 0.5] \times [-0.5, 0.5]$.

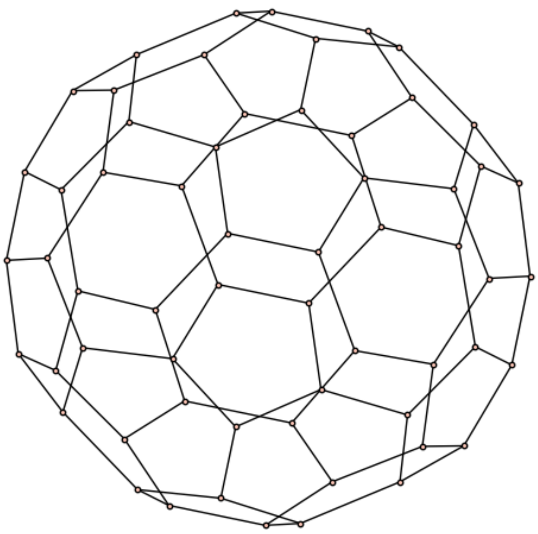
```
def move_vertex_se(G, v, pos, k, e):
    vx, vy = pos[v]
    fx, fy = pos[v]
    for u in G.neighbors(v):
        x,y = pos[u]
        dx = x - vx
```

```
        dy = y - vy
        fx += dx * k
        fy += dy * k
    for u in G.vertices(sort=False):
        if v == u:
            continue
        x, y = pos[u]
        dx = x - vx
        dy = y - vy
        r2 = dx*dx + dy*dy
        fx += -e*dx/r2
        fy += -e*dy/r2
    pos[v] = (fx, fy)


def draw_graph_se(G, k, e, iters):
    pos = {}
    for v in G.vertices(sort=False):
        pos[v] = (random() - 0.5, random() - 0.5)
    for i in range(iters):
        for v in G.vertices(sort=False):
            move_vertex_se(G, v, pos, k, e)
    G.set_pos(pos)
    return G.plot(vertex_labels = False, vertex_size = 10)
```

For the graphs below, try to find k and e such that the result will be "nice".

```
    G = graphs.BuckyBall()
    draw_graph_se(G, ?, ?, 100)
```



More examples

```
    draw_graph_se(graphs.Grid2dGraph(10, 10), ?, ?, 100)

    draw_graph_se(graphs.CycleGraph(10), ?, ?, 100)

    C10 = graphs.CycleGraph(10)
    C4 = graphs.CycleGraph(4)
    draw_graph_se(C10.cartesian_product(C4), ?, ?, 100)
```

```
draw_graph_se(graphs.RandomTree(100), ?, ?, 100)

draw_graph_se(Graph('ShCHGD@?K?_@?@?C_GGG@??cG?G?GK_?C'), ?, ?, 100)
```

6. 3-coloring planar graphs without short cycles

6.1 Introduction

The chromatic number $\chi(G)$ of a graph G is the smallest number of colors that suffice to color the vertices of G such that no two adjacent vertices have the same color.

The well known Four Color Theorem states that for every planar graph is $\chi G \leq 4$.

It is NP-hard to decide if $\chi(G) \leq 3$ if G is planar, but:

Theorem 6.1 Let G be a planar graph without cycles of lengths $4, \ldots, 11$. Then $\chi(G) \leq 3$.

6.2 Discharging method

Discharging method idea (see "Discharging method, by M. Salavatipour for more details).

If the theorem is not true and G is a smallest counterexample, then there is:

1. no vertex of degree ≤ 2 and
2. no cutvertex.

If we apply the following discharging method:

1. assign a charge of $deg(v) - 6$ units to each vertex v of G and of $2|f| - 6$ to each face f of G and
2. the rule for discharging is: each non-triangle face sends 3/2 units to each of its vertices

then we come to a contradiction with the initial total charge of -12 and the final charge ≥ 0. Thus, there is either a vertex of degree ≤ 2 or a cutvertex in such graphs.

This gives us an algorithm to color such graphs with 3 colors:

1. If we find a vertex of degree ≤ 2 we can remove it, recursively color the rest of the graph and color the removed vertex with the color missing in its two neighbors.
2. If we find a cutvertex, we split the graph into two (or more) blocks, recursively color the blocks, make sure that the removed vertex gets the same color in all blocks and color the removed vertex with that color.

6.3 Exercises

1. Write a function `initial_charge(G)` which returns dicitionary with initial charges of vertices and faces.
2. Write a function `discharge(G, c0)` which returns dictionary with charges after discharging was applied to the initial charges `c0` (result of `initial_charge(G)`).

3. Write a function `plot_charge(G, c)` which plots vertices with green color if they have non-negative charge and with red color if they are negatively charged (`c` is result of the function `plot_charge(G, c)`).
4. Write a function `three_color(G)` which implements the algorithm for three coloring of G described above.

You can use Sage built-in function `blocks_and_cut_vertices` to find cutvertices and blocks.

6.4 Solutions

```
    def faces(G):
    """
    Return faces (as "tuples" of vertices) of a planar graph G.
    """
    G.is_planar(set_embedding=True)
    F = G.faces()
    F = [tuple(x for (x, y) in f) for f in F]
    return F

def initial_charge(G):
    """
    Return a dictionary of charges for each vertex and face
    """
    F = faces(G)
    c = {}
    for v in G.vertices():
        c[v] = G.degree(v) - 6
    for f in F:
        c[f] = 2 * len(f) - 6
    return c

def discharge(G, c0):
    """
    Return a dictionary of charges for each vertex and face after
   discharging initial charges c0
    """
    c = c0.copy()
    F = faces(G)
    for f in F:
        if len(f) > 3:
            for v in f:
                c[v] += 3/2
                c[f] -= 3/2
    return c

def plot_colored_charges(G, c):
    """
    Plot negatively charged vertices of G with red and non-negatively
   charged vertices of G with green;
    according to charges given by the dictionary c
    """
    v_pos = [v for v in G.vertices() if c[v] >= 0]
    v_neg = [v for v in G.vertices() if c[v] < 0]
    return G.plot(vertex_colors = {'green': v_pos, 'red': v_neg},
   vertex_size=20, vertex_labels=False)
```

```
    def three_color(G):
    '''
    Return 3 coloring of planar graph G without cycles of length 4, ...,
   11.
```

```
    Coloring is represented as a dicitionary mapping a vertex to one of the
     colors 0, 1, 2.
    '''
    if G.num_verts() == 1:
        return {G.vertices()[0]: 0}

    G = G.copy()

    # find a cutvertex
    blocks, c_vertices = G.blocks_and_cut_vertices()
    if len(c_vertices) > 0:
        cutv = c_vertices[0]
        nbs = G.neighbors(cutv)
        result = dict()
        G.delete_vertex(cutv)
        for C in G.connected_components_subgraphs():
            # color subgraphs such that cutv has color 0
            c = three_color_cv(C, cutv, nbs)
            for v, color in c.items():
                result[v] = color
        return result

    # find a vertex of degree <= 2
    v = min(G.vertices(), key=lambda v: G.degree(v))
    if G.degree(v) <= 2:
        nbs = G.neighbors(v)
        G.delete_vertex(v)
        c = three_color(G)
        freec = list(set([0, 1, 2]) - set([c[u] for u in nbs]))
        c[v] = freec[0]
        return c
    raise Exception('No substructure')

# color G such that cutv has color 0
def three_color_cv(G, cutv, cutvnbs):
    gverts = set(G.vertices())
    G = G.copy()
    G.add_vertex(cutv)
    for u in cutvnbs:
        if u in gverts:
            G.add_edge(cutv, u)
    c = three_color(G)
    cvc = c[cutv] # color of cut vertex, we will change colors such that
   cvc will be 0
    if cvc == 0:  # ok, cut vertex has color 0
        return c
    cr = dict()
    # switch colors 0 and cvc
    for v, col in c.items():
        if col == cvc:  # color cvc -> color 0
            cr[v] = 0
        elif col == 0: # color 0 -> color cvc
            cr[v] = cvc
        else:
            cr[v] = col
    return cr
```

Unnumbered Section

Unnumbered Subsection

Unnumbered Subsubsection

7. In-text Element Examples

7.1 Referencing Publications

This statement requires citation [1]; this one is more specific [2, page 162].

7.2 Link Examples

This is a URL link: LaTeX Templates. This is an email link: example@example.com. This is a monospaced URL link: `https://www.LaTeXTemplates.com`.

7.3 Lists

Lists are useful to present information in a concise and/or ordered way.

7.3.1 Numbered List

1. First numbered item
    a. First indented numbered item
    b. Second indented numbered item
        i. First second-level indented numbered item
2. Second numbered item
3. Third numbered item

7.3.2 Bullet Point List

- First bullet point item
    – First indented bullet point item
    – Second indented bullet point item
        ◦ First second-level indented bullet point item
- Second bullet point item
- Third bullet point item

7.3.3 Descriptions and Definitions

Name Description
Word Definition
Comment Elaboration

7.4 International Support

àáâäãåèéêëìíîïòóôöõøùúûüÿýñçčšž
ÀÁÂÄÃÅÈÉÊËÌÍÎÏÒÓÔÖÕØÙÚÛÜŸÝÑ
ßÇŒÆČŠŽ

7.5 Ligatures

fi fj fl ffl ffi Ty Ty

Part Two Title

8. Mathematics

8.1 Theorems

8.1.1 Several equations

This is a theorem consisting of several equations.

Theorem 8.1 — Name of the theorem. In $E = \mathbb{R}^n$ all norms are equivalent. It has the properties:

$$\big| ||\mathbf{x}|| - ||\mathbf{y}|| \big| \leq ||\mathbf{x} - \mathbf{y}|| \tag{8.1}$$

$$||\sum_{i=1}^{n} \mathbf{x}_i|| \leq \sum_{i=1}^{n} ||\mathbf{x}_i|| \quad \text{where } n \text{ is a finite integer} \tag{8.2}$$

8.1.2 Single Line

This is a theorem consisting of just one line.

Theorem 8.2 A set $\mathscr{D}(G)$ in dense in $L^2(G)$, $|\cdot|_0$.

8.2 Definitions

A definition can be mathematical or it could define a concept.

Definition 8.1 — Definition name. Given a vector space E, a norm on E is an application, denoted $||\cdot||$, E in $\mathbb{R}^+ = [0, +\infty[$ such that:

$$||\mathbf{x}|| = 0 \Rightarrow \mathbf{x} = \mathbf{0} \tag{8.3}$$

$$||\lambda \mathbf{x}|| = |\lambda| \cdot ||\mathbf{x}|| \tag{8.4}$$

$$||\mathbf{x} + \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}|| \tag{8.5}$$

8.3 Notations

■ **Notation 8.1** Given an open subset G of $\mathbb{R}^n$, the set of functions φ are:

1. Bounded support G;
2. Infinitely differentiable;

a vector space is denoted by $\mathscr{D}(G)$.

8.4 Remarks

This is an example of a remark.

R The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field $\mathbb{K}=\mathbb{R}$, however, established properties are easily extended to $\mathbb{K}=\mathbb{C}$.

8.5 Corollaries

Corollary 8.1 — Corollary name. The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field $\mathbb{K}=\mathbb{R}$, however, established properties are easily extended to $\mathbb{K}=\mathbb{C}$.

8.6 Propositions

8.6.1 Several equations

Proposition 8.1 — Proposition name. It has the properties:

$$\big| ||\mathbf{x}|| - ||\mathbf{y}|| \big| \leq ||\mathbf{x}-\mathbf{y}|| \tag{8.6}$$

$$||\sum_{i=1}^{n} \mathbf{x}_i|| \leq \sum_{i=1}^{n} ||\mathbf{x}_i|| \quad \text{where } n \text{ is a finite integer} \tag{8.7}$$

8.6.2 Single Line

Proposition 8.2 Let $f,g \in L^2(G)$; if $\forall \varphi \in \mathscr{D}(G)$, $(f,\varphi)_0 = (g,\varphi)_0$ then $f = g$.

8.7 Examples

8.7.1 Equation Example

■ **Example 8.1** Let $G = \{x \in \mathbb{R}^2 : |x| < 3\}$ and denoted by: $x^0 = (1,1)$; consider the function:

$$f(x) = \begin{cases} e^{|x|} & \text{si } |x-x^0| \leq 1/2 \\ 0 & \text{si } |x-x^0| > 1/2 \end{cases} \tag{8.8}$$

The function f has bounded support, we can take $A = \{x \in \mathbb{R}^2 : |x-x^0| \leq 1/2+\varepsilon\}$ for all $\varepsilon \in]0;5/2-\sqrt{2}[$. ■

8.7.2 Text Example

■ **Example 8.2 — Example name.** Aliquam arcu turpis, ultrices sed luctus ac, vehicula id metus. Morbi eu feugiat velit, et tempus augue. Proin ac mattis tortor. Donec tincidunt, ante rhoncus luctus semper, arcu lorem lobortis justo, nec convallis ante quam quis lectus. Aenean tincidunt sodales massa, et hendrerit tellus mattis ac. Sed non pretium nibh. Donec cursus maximus luctus. Vivamus lobortis eros et massa porta porttitor. ■

8.8 Exercises

Exercise 8.1 This is a good place to ask a question to test learning progress or further cement ideas into students' minds. ■

8.9 Problems

Problem 8.1 What is the average airspeed velocity of an unladen swallow?

8.10 Vocabulary

Define a word to improve a students' vocabulary.

- **Vocabulary 8.1 — Word.** Definition of word.

9. Presenting Information and Results with a Long Chapter Title

9.1 Table

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie. Sed dictum nisl non aliquet porttitor. Etiam vulputate arcu dignissim, finibus sem et, viverra nisl. Aenean luctus congue massa, ut laoreet metus ornare in. Nunc fermentum nisi imperdiet lectus tincidunt vestibulum at ac elit. Nulla mattis nisl eu malesuada suscipit.

Treatments	**Response 1**	**Response 2**
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

Table 9.1: Table caption.

Referencing Table 9.1 in-text using its label.

9.2 Figure

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie. Sed dictum nisl non aliquet porttitor. Etiam vulputate arcu dignissim, finibus sem et, viverra nisl. Aenean luctus congue massa, ut laoreet metus ornare in. Nunc fermentum nisi imperdiet lectus tincidunt vestibulum at ac elit. Nulla mattis nisl eu malesuada suscipit.

Figure 9.1: Figure caption.

Referencing Figure 9.1 in-text using its label.

Treatments	**Response 1**	**Response 2**
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

Table 9.2: Floating table.

Figure 9.2: Floating figure.

Bibliography

Articles

[1] A. B. Jones and J. M. Smith. “Article Title”. In: *Journal title* 13.52 (Mar. 2022), pages 123–456. DOI: 10.1038/s41586-021-03616-x (cited on page 53).

Books

[2] J. M. Smith and A. B. Jones. *Book Title*. 7th. Publisher, 2021 (cited on page 53).

Index

A. Appendix Chapter Title

A.1 Appendix Section Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam auctor mi risus, quis tempor libero hendrerit at. Duis hendrerit placerat quam et semper. Nam ultricies metus vehicula arcu viverra, vel ullamcorper justo elementum. Pellentesque vel mi ac lectus cursus posuere et nec ex. Fusce quis mauris egestas lacus commodo venenatis. Ut at arcu lectus. Donec et urna nunc. Morbi eu nisl cursus sapien eleifend tincidunt quis quis est. Donec ut orci ex. Praesent ligula enim, ullamcorper non lorem a, ultrices volutpat dolor. Nullam at imperdiet urna. Pellentesque nec velit eget est euismod pretium.

B. Appendix Chapter Title

B.1 Appendix Section Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam auctor mi risus, quis tempor libero hendrerit at. Duis hendrerit placerat quam et semper. Nam ultricies metus vehicula arcu viverra, vel ullamcorper justo elementum. Pellentesque vel mi ac lectus cursus posuere et nec ex. Fusce quis mauris egestas lacus commodo venenatis. Ut at arcu lectus. Donec et urna nunc. Morbi eu nisl cursus sapien eleifend tincidunt quis quis est. Donec ut orci ex. Praesent ligula enim, ullamcorper non lorem a, ultrices volutpat dolor. Nullam at imperdiet urna. Pellentesque nec velit eget est euismod pretium.