Math 124 - Programming for Mathematical Applications

UC Berkeley, Spring 2023

Homework 11

Due Wednesday, April 19

Graph functions from lecture notes

Below we copy the `Graph` type from the lecture notes, and some of the functions that we will need.

In [1]:
```
using PyPlot, SparseArrays   # Packages needed

struct Vertex
    neighbors::Vector{Int}       # Indices of neighbors of this Ve
    coordinates::Vector{Float64} # 2D coordinates of this Vertex -
    Vertex(neighbors; coordinates=[0,0]) = new(neighbors, coordina
end

function Base.show(io::IO, v::Vertex)
    print(io, "Neighbors = ", v.neighbors)
end

struct Graph
    vertices::Vector{Vertex}
end

function Base.show(io::IO, g::Graph)
    for i = 1:length(g.vertices)
        println(io, "Vertex $i, ", g.vertices[i])
    end
end

function PyPlot.plot(g::Graph; scale=1.0)
    fig, ax = subplots()
    ax.set_aspect("equal")

    xmin = minimum(v.coordinates[1] for v in g.vertices)
    xmax = maximum(v.coordinates[1] for v in g.vertices)
    ymin = minimum(v.coordinates[2] for v in g.vertices)
    ymax = maximum(v.coordinates[2] for v in g.vertices)
```

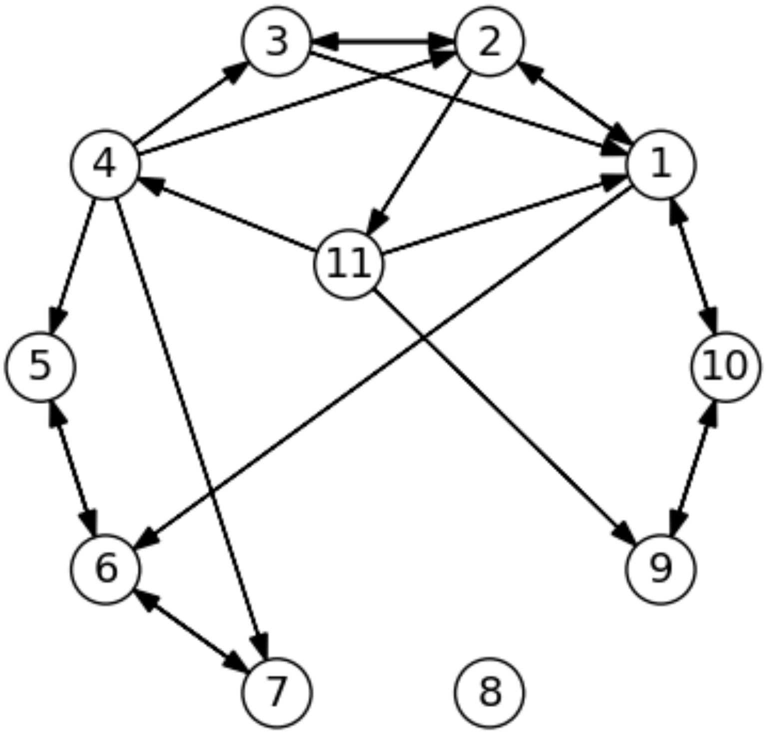
```
    ymax = maximum(v.coordinates[2] for v in g.vertices)
    sz = max(xmax-xmin, ymax-ymin)
    cr = scale*0.05sz
    hw = cr/2
    axis([xmin-2cr,xmax+2cr,ymin-2cr,ymax+2cr])
    axis("off")

    for i in 1:length(g.vertices)
        c = g.vertices[i].coordinates
        ax.add_artist(matplotlib.patches.Circle(c, cr, facecolor="
        ax.text(c[1], c[2], string(i),
                horizontalalignment="center", verticalalignment="c
        for nb in g.vertices[i].neighbors
            cnb = g.vertices[nb].coordinates
            dc = cnb .- c
            L = sqrt(sum(dc.^2))
            c1 = c .+ cr/L * dc
            c2 = cnb .- cr/L * dc
            arrow(c1[1], c1[2], c2[1]-c1[1], c2[2]-c1[2],
                  head_width=hw, length_includes_head=true, facecc
        end
    end
end

function shortest_path_bfs(g::Graph, start, finish)
    parent = zeros(Int64, length(g.vertices))
    S = [start]
    parent[start] = start
    while !isempty(S)
        ivertex = popfirst!(S)
        if ivertex == finish
            break
        end
        for nb in g.vertices[ivertex].neighbors
            if parent[nb] == 0 # Not visited yet
                parent[nb] = ivertex
                push!(S, nb)
            end
        end
    end
    # Build path
    path = Int64[]
    iv = finish
    while true
        pushfirst!(path, iv)
        if iv == start
            break
        end
        iv = parent[iv]
    end
    return path
```

```julia
end

# Example graph from lecture notes
all_neighbors = [[2,10,6], [3,1,11], [1,2], [5,3,2,7], [6],
                 [7,5], [6], Int64[], [10], [1,9], [1,4,9]]
all_coordinates = [[0.81, 0.59], [0.31, 0.95], [-0.31, 0.95], [-0.
                   [-1.0, 0.0],  [-0.81, -0.59], [-0.31, -0.95], [
                   [0.81, -0.59], [1.0, -0.0], [-0.1, 0.3]]
g = Graph([Vertex(n,coordinates=c) for (n,c) in zip(all_neighbors,
plot(g)
```



Problem 1 - Graph to adjacency matrix

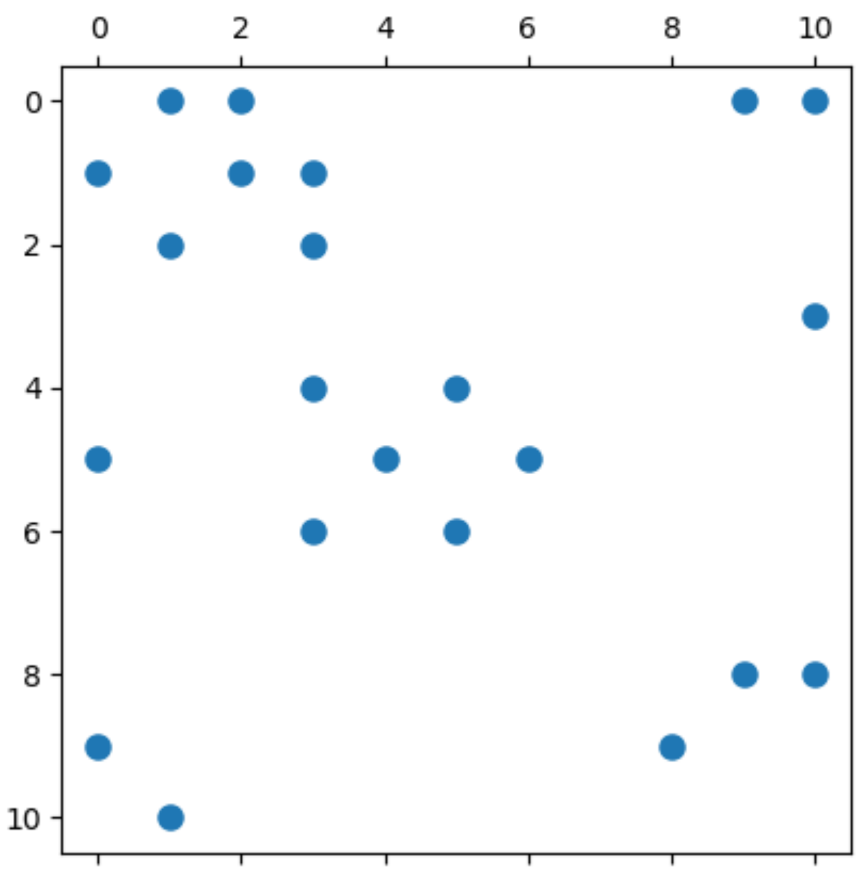
Write a function `convert2adjmatrix(g::Graph)` which creates a sparse adjacency matrix for the graph `g`. That is, a matrix A of size $|V|$-by-$|V|$ where $|V|$ is the number of vertices, and $A_{ij} = 1$ if there is an edge from vertex i to vertex j (otherwise zero).

Make sure you do not insert elements into an existing sparse matrix. Instead, create vectors of row / column indices, and call `sparse` once to create the matrix.

In [2]:
```julia
function convert2adjmatrix(g::Graph)
    rows = Int64[]
    cols = Int64[]
    v = length(g.vertices)
    for i = 1:v
        for a in g.vertices[i].neighbors
            push!(cols, i)
            push!(rows, a)
        end
    end
    c = length(cols)
    sparse(rows, cols, ones(Int64, c), v, v)
end
```

Out[2]: convert2adjmatrix (generic function with 1 method)

In [3]:
```julia
# For testing
A = convert2adjmatrix(g)
spy(A, marker=".", markersize=16);
```



Problem 2 - Spanning tree

One application of the DFS algorithm is to generate a spanning tree for a graph, that is, an acyclic graph for the nodes reachable from a given starting node. Write a function

```
function spanning_tree(g::Graph, start)
```

which returns a new graph `gtree` with the same vertices as `g`, but only the edges that are traversed by the DFS method starting from vertex `start`. That is, an edge from `ivertex` to a neighbor `nb` is only included in `gtree` if the DFS method visits `nb`.

Hint: First initialize `gtree` to a graph with no edges (but the same `coordinates` as `g`, for plotting). Then you run the DFS method and add edges to `gtree`.

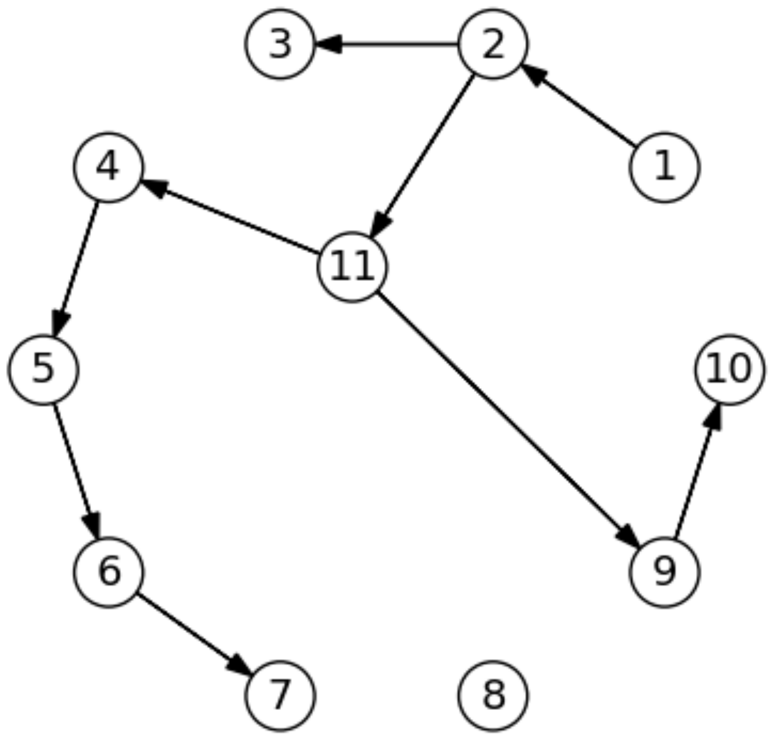
In [4]:

```julia
function spanning_tree(g::Graph, start)
    v = length(g.vertices)
    gtree = Graph([ Vertex(Int64[], coordinates=g.vertices[i].coor
    visited = falses(v)
    function visit(current_p, g::Graph)
        visited[current_p] = true
        for a in g.vertices[current_p].neighbors
            if !visited[a]
                push!(gtree.vertices[current_p].neighbors, a)
                visit(a, g::Graph)
            end
        end
    end
    visit(start, g)
    return gtree
end
```

Out[4]:

```
spanning_tree (generic function with 1 method)
```

In [5]:
```
# For testing
gtree = spanning_tree(g,1)
plot(gtree)
```



Problem 3 - Word ladder

A *word ladder* is a sequence of words, beginning at `first_word` and ending at `last_word`, such that

1. Only one letter is changed in each step
2. Each word exists in a given word list

Write a function

```
function word_ladder(first_word, last_word)
```

which returns the *shortest* such sequence of words. You can assume that the sequence exists, if it is not unique you can return any shortest sequence, and that the lengths of the two given words are equal. Use the same word list as in homework 8 (that is, https://github.com/BenLauwens/ThinkJulia.jl/blob/master/data/words.txt (https://github.com/BenLauwens/ThinkJulia.jl/blob/master/data/words.txt)).

Algorithm:

1. Read the word list into an array of string, but only keep the ones of the same length as the given two words.
2. Create a graph where these words are vertices, and an edge between words i, j if the corresponding words only differ in one character.
3. Run the `shortest_path_bfs` function on the graph, with start and finish indices corresponding to the given two words. Use the path to output the sequence of words.

Example:

```
word_ladder("fool", "sage")
```

```
7-element Array{String,1}:
 "fool"
 "foil"
 "fail"
 "fall"
 "sall"
 "sale"
 "sage"
```

In [57]:
```
function word_ladder(first_word, last_word)
    v = length(first_word)
    kept_words = String[]

```

```
    for w in readlines("words.txt")
        if length(w) == v
            push!(kept_words, w)
        end
    end

    z = length(kept_words)
    g = Graph([ Vertex(Int64[]) for a = 1:z ])
    for i = 1:z
        for j = 1:z
            if kept_words[i] != kept_words[j]
                count = 0
                for k = 1:v
                    if kept_words[i][k] != kept_words[j][k]
                        count += 1
                    end
                end

                if count == 1
                    push!(g.vertices[i].neighbors, j)
                end
            end
        end
    end

    function shortest_path_bfs(start, stop, g::Graph, kept_words)
        parent = zeros(Int64, length(kept_words))
        index = findfirst(start .== kept_words)
        Queue = [index]
        ivertex = 0

        while true
            ivertex = popfirst!(Queue)

            if stop == kept_words[ivertex]
                break
            end

            for a in g.vertices[ivertex].neighbors
                if parent[a] == 0
                    push!(Queue, a)
                    parent[a] = ivertex
                end
            end
        end

        path = Int64[]
        id = ivertex
        while true
            pushfirst!(path, id)
```

```
                if id == index
                    break
                end

                id = parent[id]
            end

            return [ kept_words[path[i]] for i = 1:length(path) ]
        end

    shortest_path_bfs(first_word, last_word, g, kept_words)
end
```

Out[57]: word_ladder (generic function with 1 method)

In [58]:

```
# For testing
word_ladder("fool", "sage")
```

Out[58]:

```
7-element Vector{String}:
 "fool"
 "foil"
 "fail"
 "fall"
 "sall"
 "sale"
 "sage"
```