Web Science Project Week 2 on Information Networks

Emiel Steegh - s1846388
Freek Nijweide - s1857746

Project Description

For this week's assignments, we will examine graph structures that are similar to the internet. We will examine how these structures are vulnerable to exploits, and how certain algorithms act upon them.

We use several packages this week, which are mostly the same as last week (except for plotly). Please make sure these are installed when trying to run our code:

- networkx
- numpy
- plotly
- pygraphviz

We decided to use plotly for this week instead of matplotlib, due to its vastly superior 3D plotting capabilities, and its interactivity. We hoped that this would increase the legibility of our results, and we think that it has done so. We came to various conclusions for assignment 3 that we could not have reached without the assistance of interactive 3D plots.

Assignment 1 (HOOD): Page ranking factors for the DuckDuckGo.

DuckDuckGo (DDG) [1] is a semantic-based search engine. This means that it stores and uses semantic information of words and websites to enhance the search query to include contextual data, as opposed to looking for keywords only. [2] For example, when you look up “Thom Palstra” on DDG you will get the Wikipedia page of the University of Twente as the 10th result, a contextual result for Thom Palstra, the Rector Magnificus at the University of Twente. This can be used to your benefit by trying to play into the intent of the searcher.

DDG gives very little information about the way they rank pages, but “Nevertheless, the best way to get good rankings (in pretty much all search engines) is to get links from high-quality sites.” [3]. This lets us know that they use some form of a link-based ranking as their main ranking force. Like most search-engines they highly value OUT-links, pointing from websites in the SCC to your website.

Prizing themselves as a ‘privacy search engine’ they do not track or record user information, so they do not use user-context for their semantic search. This means that the results of query should be the same for anyone who enters it, wherever they are. [4] When you include “near me” in a query it will use a rough location based on your IP the founder explains[5]. Because DDG does not use the user’s exact context and data for the search, they will most likely adapt the way they search to get the results they want. For finding locations they will most likely include fairly specific locations as text in their query. This logic applies broadly to how the engine is optimally used, to get better-tailored results the query will have to be more specific. So it is a good idea to let the DuckDuckBot (their web crawler) know precise information. Do not try to target a broad audience but instead define a niche and target that.

DDG states that they use over 400 sources[4], the two big names they mention are Verizon Media (used to be Yahoo) and Bing. So it seems like a good idea to optimize your page to the Yahoo and Bing standard[6], [7].

DDG blocks what they believe are spam or low-quality websites from their results.[8] So to rank better, it is advised to keep the content of your website as high quality as possible and avoid creating a site devoid of content or designed for AdSense. Users can report low-quality results they find and this may gravely impact your ranking or even get you blocked.

In summary, to optimize your page rank on DuckDuckGo: Get links from prominent sites; Think about the intent of your user and the semantics paired with it; Use specific keywords, DDG users will most likely use more detailed queries, think about what they will look for exactly (especially location-wise); Optimize for Bing and Verizon Media search engines; Keep the (content) quality of your website high and avoid spam-like practices.

Assignment 2 (FAKE): Solving the fake news problem

Fake news is a large problem in modern information technology. This is mostly seen on social media sites such as Facebook and Twitter, where users get to see a curated feed of content suited to their tastes, which might contain fake news. However, search engines such as Google can also be affected by this. A page P may achieve high search engine rankings while reporting untruthful information, with or without malicious intent.

The current solutions for this are sparse, and do not always work as intended. After the Cambridge Analytica scandal, Facebook has publicly committed to fighting fake news (although they still do not check political ads for how truthful they are[9]). Their current solution is to ask arbiters from countries around the world to verify the truthfulness of news articles.[10] In the Netherlands, this is done by NU.nl and Nieuwscheckers (an initiative by the University of Leiden)[11]. While this is preferable to the alternative, which is not checking the accuracy of news articles on their platform at all, it is quite a controversial solution.[12] The position of power granted to NU.nl (a journalism website) gives it an unfair advantage towards its competitors, because it has the capability to mark their news as fake, thus bringing down the revenue for competitors' articles.

Meanwhile, Google has also pledged to fight fake news[13], [14], although we cannot measure how much this has affected the spread of fake news (because its approach has not been as controversial as Facebook's). Twitter has not publicly made such pledges, but they have been banning more accounts recently for publishing untruthful information, and are planning to ban all political advertising.[15] Their true stance towards preventing fake news is questionable, as Twitter still allows fake news from accounts such as the US President, Donald Trump.

We propose several ideas, given unlimited budget and manpower, to solve these problems:

The first possible solution is to train an AI (neural network with several other human-defined constraints) to rank articles by their truthfulness, and display this score to the user. This score would then be displayed (in a human-readable format, so "Untrustworthy" instead of 0.1245...) next to the article link in the search engine / social media platform. This solution has several advantages:

- It is multidimensional. Many different variables can be processed by such an AI, such as the tone of the language used, the popularity of the website itself, its previous truthfulness rankings, constraints added by the developers ("Blogs are inherently untrustworthy"), user input on the search engine itself (letting them answer questions "how trustworthy would you rate this article?"), etc. Computers are perfectly capable of processing all those factors, while a human would quickly lose sight of the bigger picture when examining these differences.
- A computerized solution is much faster than a team of humans. This lets the user instantly access news that was released seconds ago, while already knowing how trustworthy the article is. A team of humans would need at least several minutes to rate the truthfulness of the article, reducing the speed of the news.
- Keeping this "score" next to the link in the search engine warns the user that an article might be untrustworthy, but does not unfairly affect the revenue stream of false positives. The article will still be at the top of the search results, it may just have a warning next to it.

There are also several problems with this idea:

- Artificial intelligences are prone to bias. If the creators use a biased dataset to train the AI, it will retain that bias. For example, if the creators are more likely to report a conservative news source as untrustworthy, the AI will also have this tendency.
- Training an AI is expensive. A large amount of computing power is needed for this. Furthermore, an incredibly large dataset is required, which needs to be ranked according to several variables by human operators. These need to be paid. Luckily, we have unlimited resources in this assignment.
- Not malleable. If a change needs to be made, it is quite hard to retrain an existing neural network, and this may be quite costly.
- This solution is prone to errors. Inconsistencies in the model may lead to false positives. If an author uses a certain vocabulary, their article may be flagged as untrustworthy. A team of humans would never do this.

The second solution is to train a large team of humans. This solution is similar to the first one. The positive aspects of this are:

- Less prone to errors (explained above).
- Criteria for trustworthiness can easily be changed over time by simply telling the employees to grade pages differently. Thus, changing this solution over time is very easy and quick

Negative aspects:

- Much more expensive, as a team of employees now needs to be paid beyond the R&D phase of the project. Humans require more time to rank an article (thus making the delivery of checked news less quick).

A third possible solution would be to train an algorithm to aggregate news from different sources, and create a summary. Positive aspects of this include:

- This process gives an all-encompassing view of actualities. This is a sum of all different opinions of journalists across the globe on this article. Therefore, this summary is the most likely version of "neutral" news one might ever hope to get. Even if it is not truly neutral, it invites the reader to think critically as completely contrasting opinions will be juxtaposed.
- Reduces the need for users to read multiple news sources, they have just one source for news. This saves quite some time for avid readers.

Problems with this solution:

- There is little incentive for journalists to produce content if this system becomes popular. Unless the creators of this program forward a share of the revenue to the journalists they aggregated content from, they will not receive any profits for their work, because people will only visit this system and not the journalists' websites.
- A lot of power is concentrated in one location if this system becomes popular. While the creators may have good intentions, future owners of this system may use it for manipulative purposes. Furthermore, by having one very popular news source which is run by an algorithm, the system will become the target of hackers and others trying to exploit it. One does not need to actually hack the system to exploit its vulnerabilities; an idea like Search Engine Optimization could be used, finding the best "trigger words" to make the system prioritize one's manipulative content.
- This program is unlikely to reach lots of people as they still have their own favorite news sources. They are not likely to switch completely to this content aggregator. Furthermore, it does not solve the problem of fake news being shared on a social media platform. Any articles shared on, for example, Facebook will not be affected by this program, even if it is made by Facebook. A proper way would have to be found to make sure this aggregate version gets seen by users who would otherwise be exposed to fake news.

We would choose to implement the first concept. It clearly has the most return on investment (where the return is measured in impact). The second option could be just as effective, but the costs to achieve this are far higher. The third option has lots of potential, but it seems hard to achieve much impact using this system. Maybe somebody with better marketing skills could come up with superior use cases.

References for assignments 1 and 2

- [1] “DuckDuckGo — Privacy, simplified,” DuckDuckGo. [Online]. Available: https://duckduckgo.com/ (https://duckduckgo.com/). [Accessed: 19-Nov-2019].
- [2] D. V. Parsania, F. Kalyani, and K. Kamani, “A Comparative Analysis: DuckDuckGo Vs. Google Search Engine,” Global Research and Development Journal for Engineering, Jan. 2017.
- [3] DuckDuckGo, “Rankings (SEO),” DuckDuckGo Help Pages. [Online]. Available: https://help.duckduckgo.com/duckduckgo-help-pages/results/rankings/ (https://help.duckduckgo.com/duckduckgo-help-pages/results/rankings/). [Accessed: 19-Nov-2019].
- [4] DuckDuckGo, “Sources,” DuckDuckGo Help Pages. [Online]. Available: https://help.duckduckgo.com/duckduckgo-help-pages/results/sources/ (https://help.duckduckgo.com/duckduckgo-help-pages/results/sources/). [Accessed: 19-Nov-2019].
- [5] “How does DuckDuckGo know where I am?,” Quora. [Online]. Available: https://www.quora.com/How-does-DuckDuckGo-know-where-I-am (https://www.quora.com/How-does-DuckDuckGo-know-where-I-am). [Accessed: 19-Nov-2019].

- [6] “Bing - SEO-analyse.” [Online]. Available: https://www.bing.com/toolbox/seo-analyzer (https://www.bing.com/toolbox/seo-analyzer). [Accessed: 19-Nov-2019].
- [7] “How to Optimize for Yahoo!” [Online]. Available: http://www.searchengineguide.com/ross-dunn/how-to-optimize.php (http://www.searchengineguide.com/ross-dunn/how-to-optimize.php). [Accessed: 19-Nov-2019].
- [8] C. Mims, “The Search Engine Backlash Against ‘Content Mills,’” MIT Technology Review, 26-Jul-2010. [Online]. Available: https://www.technologyreview.com/s/419965/the-search-engine-backlash-against-content-mills/ (https://www.technologyreview.com/s/419965/the-search-engine-backlash-against-content-mills/). [Accessed: 19-Nov-2019].
- [9] “Concerns about Facebook’s political ad policy brought to Zuckerberg's dinner table,” CNN Politics. [Online]. Available: https://edition.cnn.com/2019/11/05/politics/facebook-mark-zuckerberg-political-ads/index.html (https://edition.cnn.com/2019/11/05/politics/facebook-mark-zuckerberg-political-ads/index.html). [Accessed: 19-Nov-2019].
- [10] “Working to Stop Misinformation and False News,” Working to Stop Misinformation and False News | Facebook Media. [Online]. Available: https://www.facebook.com/facebookmedia/blog/working-to-stop-misinformation-and-false-news (https://www.facebook.com/facebookmedia/blog/working-to-stop-misinformation-and-false-news). [Accessed: 22-Nov-2019].
- [11] “Facebook gaat ook in Nederland nepnieuws aanpakken.” [Online]. Available: https://nos.nl/artikel/2160892-facebook-gaat-ook-in-nederland-nepnieuws-aanpakken.html (https://nos.nl/artikel/2160892-facebook-gaat-ook-in-nederland-nepnieuws-aanpakken.html). [Accessed: 22-Nov-2019].
- [12] R. Kist, “‘Concurrenten de maat nemen is ongemakkelijk,’” NRC, 12-Sep-2019. [Online]. Available: https://www.nrc.nl/nieuws/2019/09/12/concurrenten-de-maat-nemen-is-ongemakkelijk-a3973207 (https://www.nrc.nl/nieuws/2019/09/12/concurrenten-de-maat-nemen-is-ongemakkelijk-a3973207). [Accessed: 22-Nov-2019].
- [13] K. Roose, “Google Pledges $300 Million to Clean Up False News,” 20-Mar-2018. [Online]. Available: https://www.nytimes.com/2018/03/20/business/media/google-false-news.html (https://www.nytimes.com/2018/03/20/business/media/google-false-news.html). [Accessed: 22-Nov-2019].
- [14] “Google News Initiative – Google News Initiative,” Google News Initiative. [Online]. Available: https://newsinitiative.withgoogle.com/ (https://newsinitiative.withgoogle.com/). [Accessed: 22-Nov-2019].
- [15] BBC News, “Twitter to ban all political advertising,” BBC News, 31-Oct-2019. [Online]. Available: https://www.bbc.com/news/world-us-canada-50243306 (https://www.bbc.com/news/world-us-canada-50243306). [Accessed: 22-Nov-2019].

Import and Settings for assignments 3 and 4

In [1]:

```
# The following includes are needed to work with graphs and display solutions.
from __future__ import division
import networkx as nx

from IPython.display import SVG
from IPython.display import HTML
from IPython.display import display
import StringIO
from networkx.drawing.nx_pydot import read_dot
from networkx.drawing.nx_pydot import from_pydot
from networkx.drawing.nx_agraph import to_agraph
import pydot

import numpy as np
import random

# from utils import *
# from graphs import *

import plotly.graph_objects as go


print("imports done")
```

```
imports done
```

Functions from last week

In [2]:

```
# All these definitions are copied from last week's canvas files, utils.py and graphs.p
y
def fromDot(s):
  P_list = pydot.graph_from_dot_data(s)
  return from_pydot(P_list[0])

def draw(G, mapping=None, emapping=None):
    '''draw graph with node mapping and emapping'''
    A=to_agraph(G)
    A.graph_attr['overlap']='False'
    if mapping:
        if isinstance(mapping, dict):
            mapping = nM(mapping)
        for n in A.nodes():
            mapping(n, A.get_node(n))
    if emapping:
        if isinstance(emapping, dict):
            emapping = eM(emapping)
        for e in A.edges():
            emapping(e, A.get_edge(e[0],e[1]))
    A.layout()
    output = StringIO.StringIO()
    A.draw(output, format='svg')
    return SVG(data=output.getvalue())
```

Function definitions for use in both assignments

In [3]:

```
#General code definitions
def generate_M_and_v(graph):
    # Generates M (adjacency matrix) and v (list of nodes) for any graph
    M = nx.to_numpy_matrix(graph) #type: np.matrix
    v = list(graph.nodes)
    return M,v

def pagerank(graph,beta):
    # Our implementation of pagerank
    M,nodes = generate_M_and_v(graph)
    for i in range(len(nodes)):
        node=nodes[i]
        if graph.out_degree(node) > 0:
            # Divide the numbers in the adjacency matrix by their out-degree, the first
step of turning this into a transition matrix
            M[i] /= graph.out_degree(node)
    M=M.T # We were working with adjacency matrix. The transition matrix is a transpose
of this, where each
    #       number is divided by the out-degree of the node the edge is coming from
    original_v=np.ones(len(M),dtype=float)/len(M) # The vector v, for pagerank, is of l
ength n with each element = 1/n
    v = np.copy(original_v)
    change_was_made = True
    while change_was_made:
        # do v' = beta*M*v + (1-beta)*original_v until the result no longer changes
        previous_v = np.copy(v)
        first_term = beta* (np.array(np.dot(M,v)).flatten())
        second_term = np.dot((1-beta),original_v)
        v = first_term + second_term
        change_was_made = ((abs(v-previous_v).max() ) > 0.00000000000001) #This 0.00...
number was experimentally chosen. Smaller values seemed not to converge, for some graph
s

    return v

def wrap_pagerank_in_dict(graph,beta=0.85):
    # Wraps the pagerank function's output in a dict
    values = pagerank(graph,beta)
    keys = list(graph.nodes())
    return dict(zip(keys,values))

def order_nodes_by_rank(nodes, rank):
    # rank is a list of ranking numbers, where the order corresponds to the order of th
e list "nodes"
    # an example using pagerank: rank[i] is the pagerank of the node at nodes[i]
    order = np.argsort(rank)[::-1] #sort the nodes by rank number in descending order.
 order[i] = index of node i in this sorted list
    sorted_nodes = [nodes[i] for i in order] #Human readable sorted list of nodes
    ordinal_rank = [list(order).index(i) for i in range(len(order))] # Ordinal rank as
 specified in exercise 4
    return sorted_nodes,ordinal_rank
```

Graph Definitions

Necessary for generating the graphs we will use for testing.

The first graph was merely created to test our ideas in a fixed setting.

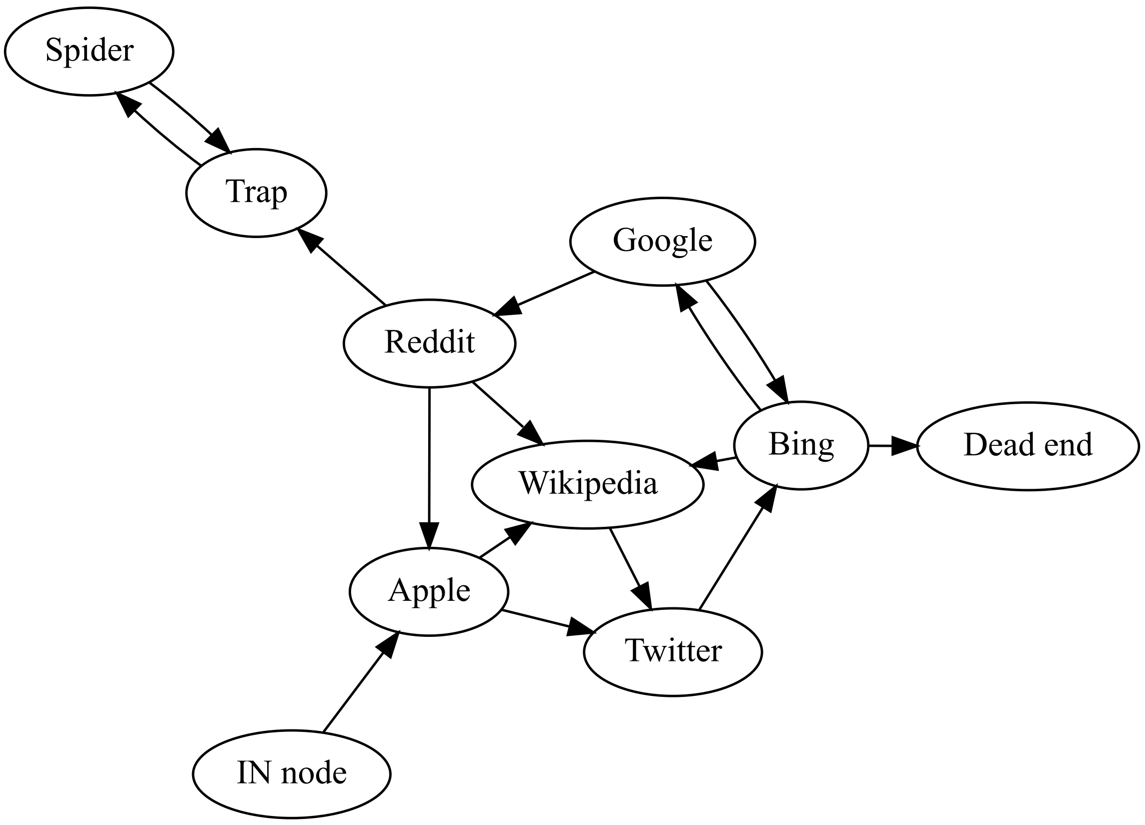
The graphs after that are copied from the text of assignment 4 on Canvas.

In [4]:

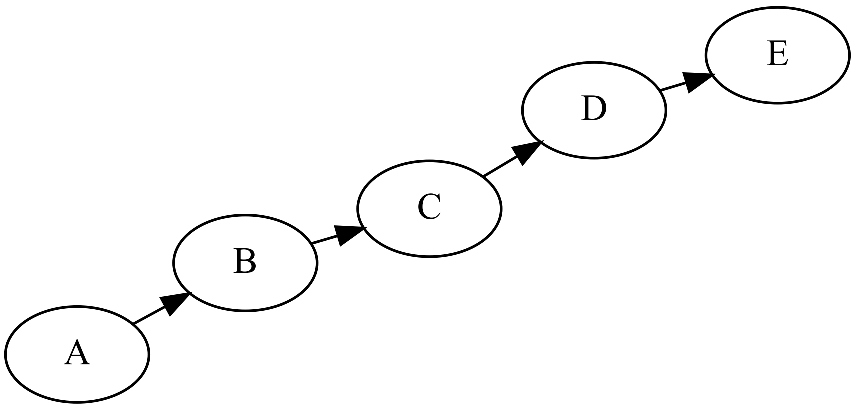
```
def generate_network(n_count, out_d_count, in_d_count):
    '''
    Generates a network around T (Target) with
    n inaccesible pages for T
    out_d_count amount of out degree per node
    in_d_count amount in in degree per node
    '''

    G = nx.DiGraph()

    for n in range(0, n_count):
        new_node = "n[{}]".format(n)
        G.add_node(new_node)

        nodes_in_network = list(G.nodes())
        nodes_in_network.remove(new_node)

        current_node_count = len(nodes_in_network)

        if current_node_count < out_d_count:
            nodes_out_index = random.sample(range(current_node_count), current_node_cou
nt)
        else:
            nodes_out_index = random.sample(range(current_node_count), out_d_count)

        if current_node_count < in_d_count:
            nodes_in_index = random.sample(range(current_node_count), current_node_coun
t)
        else:
            nodes_in_index = random.sample(range(current_node_count), in_d_count)

        for index in nodes_out_index:
            G.add_edge(new_node, nodes_in_network[index])

        for index in nodes_in_index:
            G.add_edge(nodes_in_network[index], new_node)

    return G

def generate_test_web():
    # Generate a graph we will use in the exercise 4
    graph = nx.DiGraph()

    graph.add_edge('Google', 'Bing')
    graph.add_edge('Google', 'Reddit')
    graph.add_edge('Reddit', 'Trap')
    graph.add_edge('Spider','Trap')
    graph.add_edge('Trap','Spider')
    graph.add_edge('Reddit', 'Apple')
    graph.add_edge('Reddit', 'Wikipedia')
    graph.add_edge('Apple', 'Twitter')
    graph.add_edge('Twitter', 'Bing')
    graph.add_edge('Wikipedia', 'Twitter')
    graph.add_edge('Bing', 'Wikipedia')
    graph.add_edge('Apple', 'Wikipedia')
    graph.add_edge('Bing','Google')
    graph.add_edge('Bing','Dead end')
    graph.add_edge('IN node','Apple')
    graph.name = "Big web graph (for testing)"
```

```python
    return graph

# The following graphs correspond to the examples in the assignment 4 text on Canvas

def gen_arrow():
    G = fromDot('''
    strict digraph A {
    A -> B -> C -> D -> E;
    }''')
    G.name = "Arrow graph"
    return nx.DiGraph(G)

def gen_inward():
    G = fromDot('''
    strict digraph A {
    {B C D E F G } -> A;
    }''')
    G.name = "Inward graph"
    return nx.DiGraph(G)

def gen_lasso():
    G = fromDot('''
    strict digraph A {
    A -> B -> C -> D -> E -> A;
    A -> E -> D -> C -> B -> A;
    A -> F -> G;
    }''')
    G.name = "Lasso graph"
    return nx.DiGraph(G)

def gen_grid():
    G = fromDot('''
    strict digraph A {
    A -> {B D};
    B -> {C E};

    C -> {F};
    D -> {E G};
    E -> {F H};
    F -> {I};
    G -> {H};
    H -> {I};
    I -> {};
    }''')
    G.name = "Grid graph"
    return nx.DiGraph(G)


graphs = [
generate_test_web(),
gen_arrow(),
gen_inward(),
gen_lasso(),
gen_grid()
]

for graph in graphs:
    print graph.name
    display(draw(graph))
```

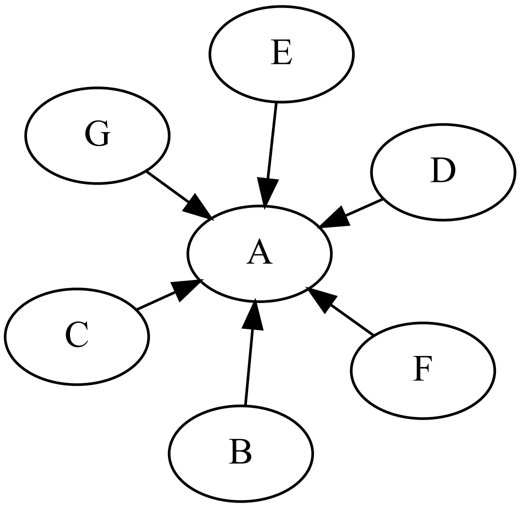
```
Big web graph (for testing)
```



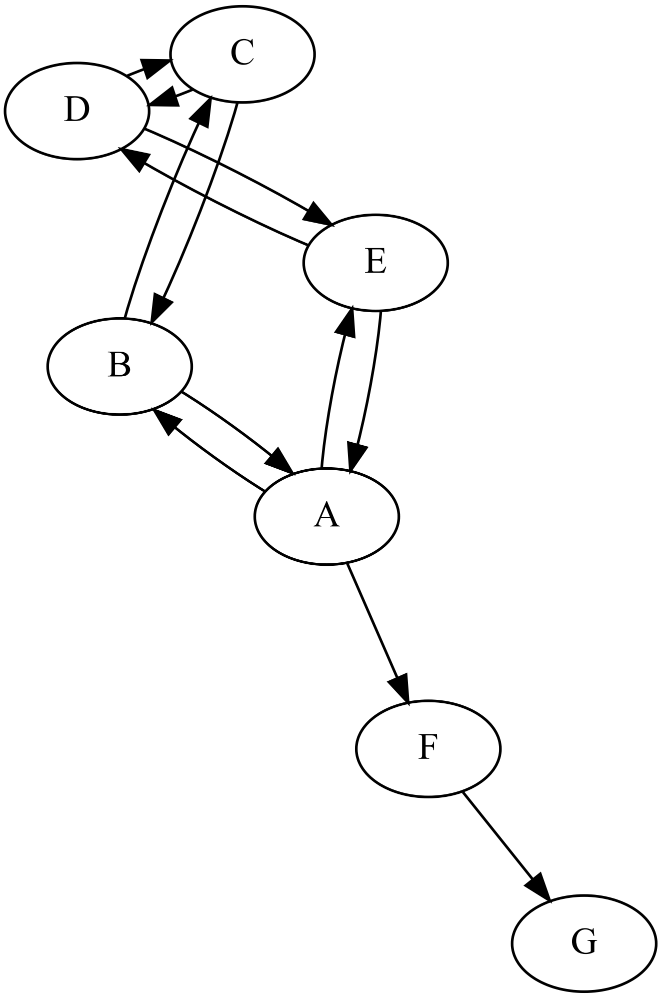
```
Arrow graph
```



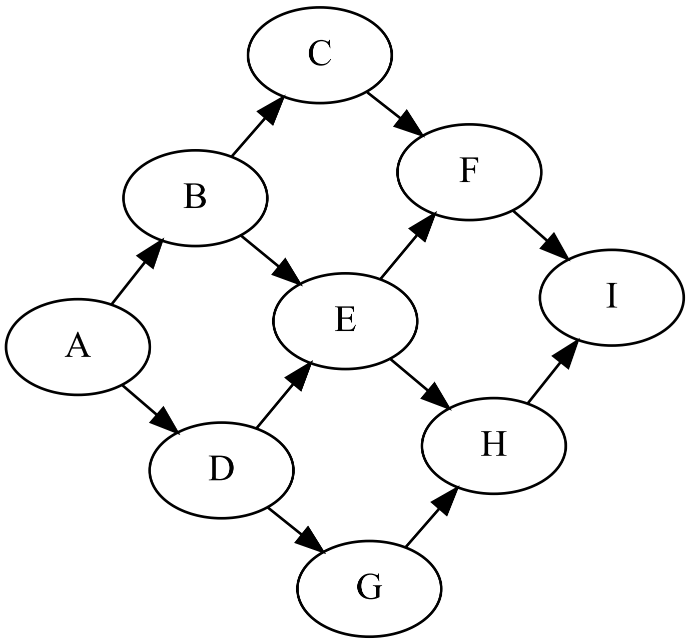
```
Inward graph
```



Lasso graph



Grid graph



Assignment 3 (SPAM): Growing a spam farm

In this exercise, we use three dependent variables to plot our data. We want to know the effect of these variables on our target page, T.

The first is P, the amount of accessible/incoming pages as defined in assignment 3. The second is f, the amount of supporting pages as defined in assignment 3. The third is the "quality" of the pages P. We define this as the percentile of these pages, when ordering by PageRank.

We then calculate the PageRank for these pages again, after connecting T to the pages P. We generate three plots from this. For these plots, and their explanations, regard the cells below.

We will deliver a short mathematical proof for why an increasing r (as defined in the exercise text), and an increasing f, would increase the ranking of T.

$$n = total\ amount\ of\ nodes$$

$$F = The\ set\ of\ supporting\ pages$$

$$T = the\ PageRank\ of\ target\ node\ T$$

$$r = \beta \cdot \sum_p r_p$$

$$T = r + \sum_{f \in F}(net\ PageRank\ received\ from\ f)$$

$$net\ PageRank\ received\ from\ f = (1-\beta) \cdot \frac{1}{n} \cdot n = 1 - \beta$$

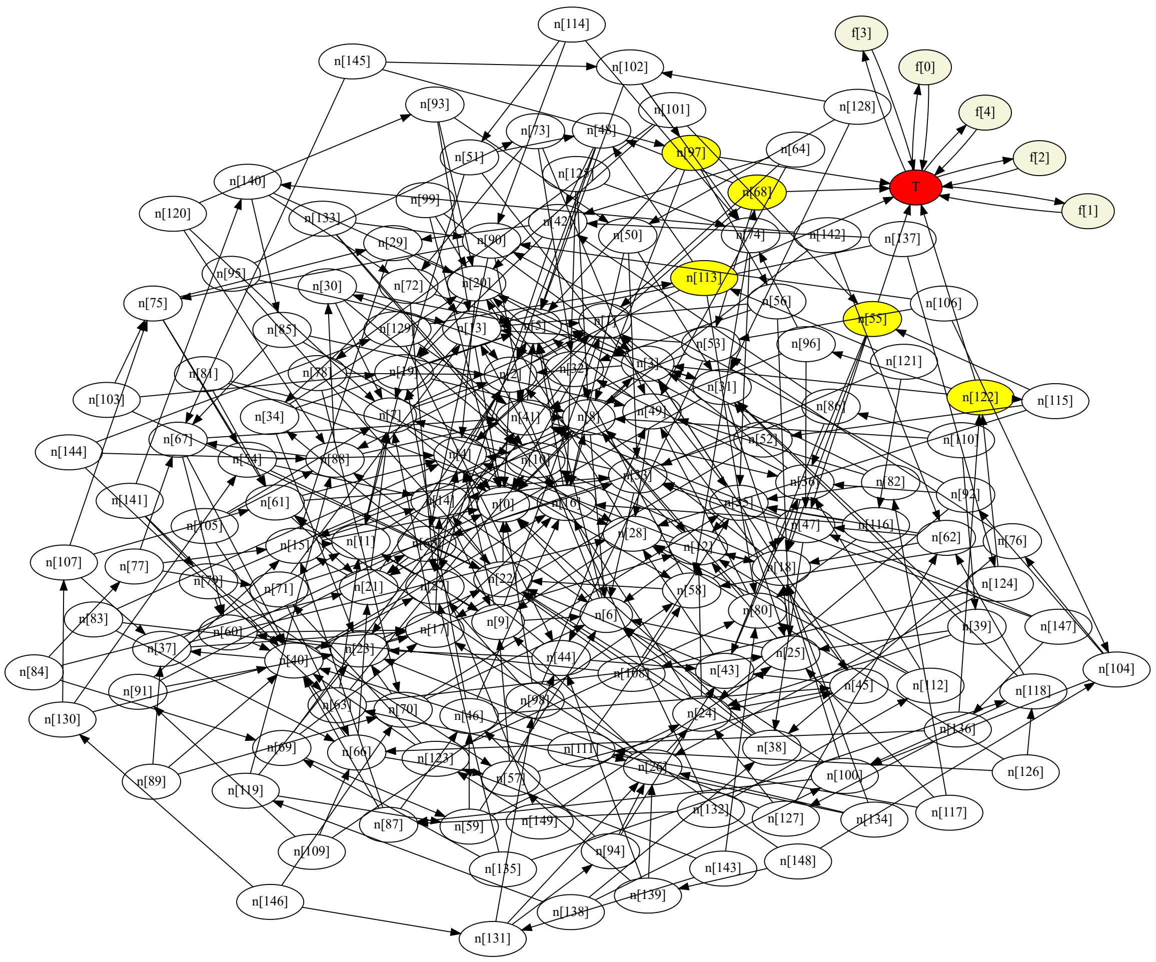
$$T = r + (1-\beta) \cdot f$$

Obviously, if r increases, T increases as well. The same holds for f.

In [5]:

```
def return_T_rank(G_o, P_count, quality, f_count, ordered_nodes, drawing):
    '''
    Returns the pagerank of T, the percentile of it's pagerank, and the adjusted versio
n thereof (when leaving supporting pages out of it)
    '''

    G = G_o.copy() #a duplicate of the graph to work on

    nodes_count = len(G.nodes())

    P_T_connections = [] # a list that will conatain the n pages that will become acces
sible to T

    index = int(round(quality*nodes_count)) #starting point for grabbing web pages

    for P in range(P_count): #grab the first page and go decrement to the next lower qu
ality pages
        P_T_connections.append(ordered_nodes[index-P-1])

    attrs = {} # a dict for drawing a network with colors to signify interesting nodes
    attrs['T'] = {'style' : 'filled', 'fillcolor' : 'red'}

    #add T, this is neccessary to prevent out no T in the network if P = 0 && f = 0
    G.add_node('T')

    #connect T to all P
    for P in P_T_connections:
        G.add_edge(P, 'T')
        attrs[P] = {'style' : 'filled', 'fillcolor' : 'yellow'}

    #connect T and it's supporting pages
    for f in range(int(f_count)):
        new_f = "f[{}]".format(f)
        G.add_edge('T', new_f)
        G.add_edge(new_f, 'T')
        attrs[new_f] = {'style' : 'filled', 'fillcolor' : '#f5f5dc'}

    #draw and use the color attributes only if desired, gives some visual insight into
 the network but makes many repeat computations very slow
    if drawing:
        nx.set_node_attributes(G, attrs)
        display(draw(G))

    #calculate the pagerank - in general definitions, theoretical explanation in assign
ment 4
    #and calculate the percentile of page T's pagerank
    pr = pagerank(G, 0.85)
    T_pagerank = pr[list(G.nodes).index('T')]
    pagerank_order, ordinal = order_nodes_by_rank(list(G.nodes),pr)
    T_pagerank_index = pagerank_order.index('T')
    reported_percentage = 100 - (T_pagerank_index / len(pr) * 100)
    #the adjusted percentile removes the supporting pages from the equation
    adjusted_percentage = 100 - (T_pagerank_index / (len(pr)-f_count) * 100)

    return T_pagerank, reported_percentage, adjusted_percentage

def plot_assignment3_results(ls_data_in, title_in, q_range,):
    '''
    takes a list of 2Darrays to plot them on top of each other, a slider lets you selec
```

```
t which array from the list you see
    '''

    #there are three types of plots to be made: pagerank, percentile & adjusted percent
ile
    #this adjusts the title to the relevant type
    l_title = '{} of T based on f and r'.format(title_in)

    fig = go.Figure()

    itter = 0 #for the naming of the traces
    #loop over the array list and plot each as an invisble surface graph
    for data in ls_data_in:
        fig.add_trace(go.Surface(
                        visible=False,
                        name="P {}".format(itter),
                        z=data, y=100*(1-q_range)))
        itter += 1

    #add axis labels, title and size here
    fig.update_layout(title=l_title,
                    autosize=True,
                    scene = dict(
                    xaxis_title='amount of supporting pages',
                    yaxis_title='percentile of pagerank gained from accessible pages',
                    zaxis_title=title_in))

    #set something to be visble
    fig.data[0].visible = True

    #code for sliders from slider example:
    #       https://plot.ly/python/sliders/
    #thanks! :)
    steps = []
    for i in range(len(fig.data)):
        step = dict(
            method="restyle",
            args=["visible", [False] * len(fig.data)],
        )
        step["args"][1][i] = True  # Toggle i'th trace to "visible"
        steps.append(step)

    sliders = [dict(
        active=10,
        currentvalue={"prefix": "Amount of accessible pages: "},
        pad={"t": 50},
        steps=steps
    )]

    fig.update_layout(
        sliders=sliders
    )
    #end of copied slider code

    fig.show()

def wrapper(Graph, P_range, q_range, f_range):
    '''
    Takes the desired ranges of P q and f to graph and itterates through them all
    The final function that calls the others
    '''
    #a bit of user-error prevention, one cannot have more pages P than fit in a step of
quality
    nodes_count = len(G.nodes())
    max_P_count = int(nodes_count/len(q_range))
    P_count = max(P_range)
    if P_count > max_P_count:
        print "The number of accessible pages (P) cannot be larger than {} for the curr
ent quality range (q)".format(max_P_count)
        return None

    #with error prevention out of the way it is time to start
    ranked_nodes = pagerank(G, .85) #calculate the pagerank of the net without T here t
o save computational time, pass it for every calculation
    ordered_nodes, ordinal_rank = order_nodes_by_rank(list(G.nodes),ranked_nodes)

    #lists to be filled wto send to the plotting function
    ls_results_pagerank = []
    ls_results_percentile = []
    ls_results_percentile_adj = []


    for index_P in range(len(P_range)):
        #this wrapper function takes some time (depending on your machine) so a proof o
f life might be nice
        print "working on P ({}/{})".format(index_P, int(P_count))

        #declare the zeroed 2D array to store the data in (they need to be filled with
 something to work, just declaring size wont work)
        results_pagerank = np.zeros([len(q_range),len(f_range)])
        results_percentile = np.zeros([len(q_range),len(f_range)])
        results_percentile_adj = np.zeros([len(q_range),len(f_range)])

        #fill the 2D array with appropriate data
        for index_f in range(len(f_range)):
            for index_q in range(len(q_range)):
                f_count = f_range[index_f]
                quality = q_range[index_q]
                T_pagerank, T_percentile, T_percentile_adj = return_T_rank(G, index_P,
quality, f_count, ordered_nodes, False)

                results_pagerank[index_q][index_f] = T_pagerank
                results_percentile[index_q][index_f] = T_percentile
                results_percentile_adj[index_q][index_f] = T_percentile_adj

        #and append the data to its respective list
        ls_results_pagerank.append(results_pagerank)
        ls_results_percentile.append(results_percentile)
        ls_results_percentile_adj.append(results_percentile_adj)

    print 'done'
    #send it all out to be plotted
    plot_assignment3_results(ls_results_pagerank, "Pagerank", q_range)
    plot_assignment3_results(ls_results_percentile, "Pagerank percentile ",q_range)
    plot_assignment3_results(ls_results_percentile_adj, "Adjusted pagerank percentile",
q_range)

    #done
    return None

def draw_network(G, P, q, f):
    '''
    quick and fairly dirty code for drawing the graph the code uses, not neccessary but
can give some insights
    G staring = graph
    '''
    ranked_nodes = pagerank(G, .85)
    ordered_nodes, ordinal_rank = order_nodes_by_rank(list(G.nodes),ranked_nodes)
    get_rid_of_return_print = return_T_rank(G, P, q, f, ordered_nodes, True)


G = generate_network(150, 3, 0) #create the network without T and f to work on, with 15
0 n, with up to 3 out per node and 0 in on spawn

draw_network(G, 5, 0.5, 5) #draws network we work on, with 5 pages P, of quality 50%, a
nd 5 pages f
```



The displayed graph shows the generated graph with

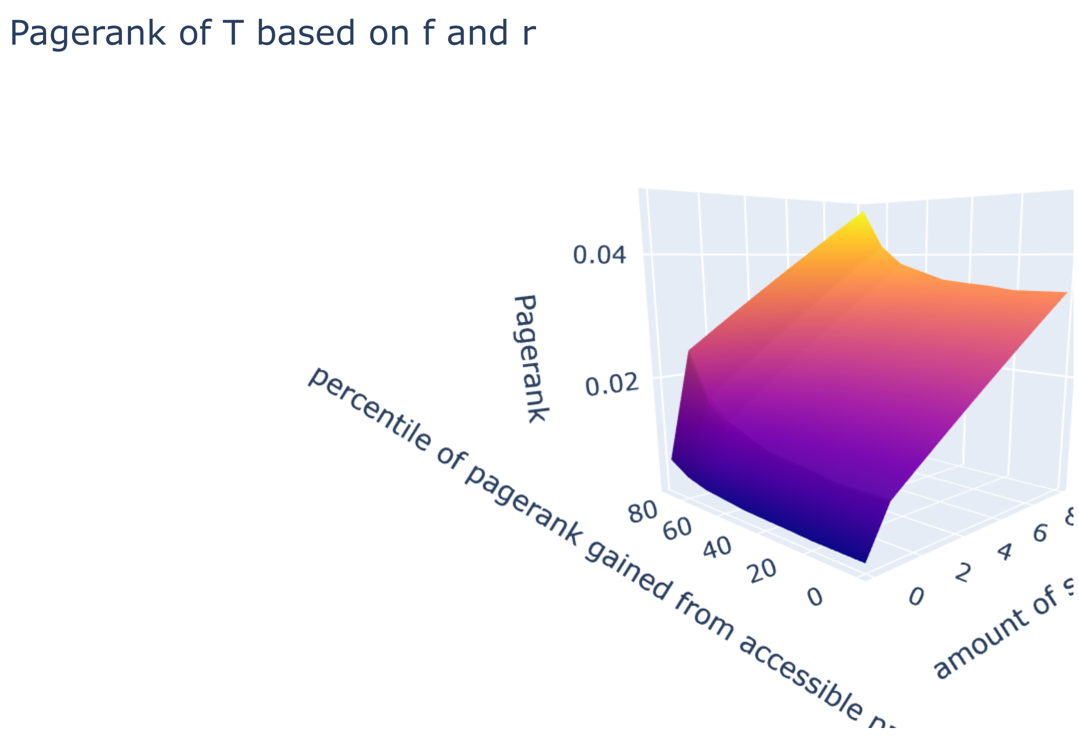
- The target page (T) in red
- 5 supporting pages (f) in beige
- 150 inaccesible pages (n) in white
- of which 6 became accessible pages (P) in yellow

The graph is created by repeatedly adding nodes n to an initially empty network. When a node is added it will link out to and get links in from existing nodes. for this assignment we chose 3 out & 0 in. A pageranking is then generated for this component. T is added to the graph after and P accessible pages are chosen based on their pageranking percentile to link to T. Finally f support pages are added and connected to and from T.

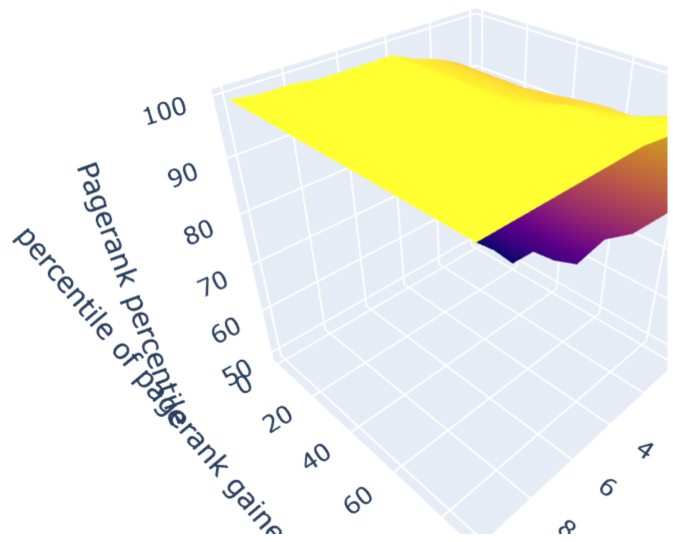
In [6]:

```
P_range = np.linspace(0,6,7) #With 0->6 accessible pages P
q_range = np.linspace(0.1,1,10) #With -> increments of 10% in quality of those P pages
f_range = np.linspace(0,10,11) #with 0->10 supporting pages f
wrapper(G, P_range, q_range, f_range)
```

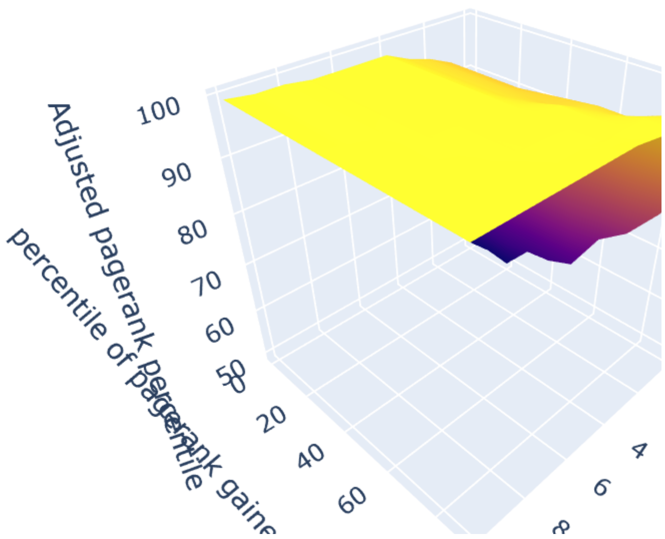
```
working on P (0/6)
working on P (1/6)
working on P (2/6)
working on P (3/6)
working on P (4/6)
working on P (5/6)
working on P (6/6)
done
```



Pagerank percentile of T based on f and r



Adjusted pagerank percentile of T based on f and r



Assignment 3 explanation

The first plot contains the PageRank: a higher amount of incoming pages (P) increases the importance of f, the amount of supporting pages. There is an obvious linear trend, both the pagerank of P and amount of supporting pages lead to an increase in page rank. Obviously, with P=0, the pagerank of the incoming pages makes no difference. Still, the performance is quite alarming. The component is disconnected and still, T gets quite a high rank with ease.

The percentile in the second plot is quite simple. If the PageRank of T is the highest in the graph, it is in the 100th percentile. If it is the worst, it is the 0th percentile. If it has the exact median of all PageRanks, it is in the 50th percentile. The pagerank of the incoming, accessible pages makes no difference here if p=0, like in the previous example. There is a strong trend between T's pagerank percentile and the amount of supporting pages. With only a few pages, it is already the page with the highest PageRank. It is quite astonishing how easy we have exploited the PageRank algorithm. The importance of the incoming PageRank is increased with higher P, but still, for T's PageRank percentile it does not matter much. The amount of supporting pages is the most important factor here. With only one supporting page, T is already in the top 10%.

The third plot contains the adjusted PageRank percentile. This should be lower than the original calculated percentile. We now divide by n-(amount of supporting pages) instead of dividing by n to get the percentile. Obviously, when n=10 we add 40 supporting pages, we will automatically be in the top 20%, because 10/50 = 20%. already even if we are still at spot 10 when we order by PageRank. When diving by n-(amount of supporting pages), we get: 10/10 = 100%. Thus, our page would be the 0th percentile. However, it appears this does not make much difference, when comparing this plot to the previous one. Thus, we know the massive increase in percentile is not simply due to the amount of pages added.

Thus, it is quite clear each variable leads to an increase in the ranking of T. However, when looking at the percentile of T's rank, f (the amount of supporting pages) is clearly the dominant factor. This is an obvious exploit in PageRank.

Assignment 4 (RANK): Describing different ranking algorithms

There are three different ranking systems discussed in this week's chapters and lectures. The first is an inDegree-based ranking system. The ranking of a page depends on how many pages link to it. This system is easily fooled, and dead-end pages receive incredibly high rankings.

The second ranking system discussed is Google's PageRank. This system expands upon the idea of inDegree rankings, by having a node "spread" its incoming ranking over its outgoing links. This can also include a degree of randomness, where a node will spread some of its incoming ranking over *all* nods (to avoid spider traps and dead ends).

The third ranking system discussed HITS, based on the hub-authority model. In this model, a page is considered a good hub if it links to good authorities (pages with good content that people want to see). A page is considered a good authority if it is linked to by proper hubs. The page's authority ranking is the one actually used in the search.

The equations used for HITS are quite simple:

$$h' = \frac{Ma}{max(h')}$$

$$a' = \frac{M^T h}{max(a')}$$

The equation used for PageRank:

$$v' = \beta M v + (1-\beta)\begin{bmatrix}1/n\\1/n\\1/n\\1/n\end{bmatrix}$$

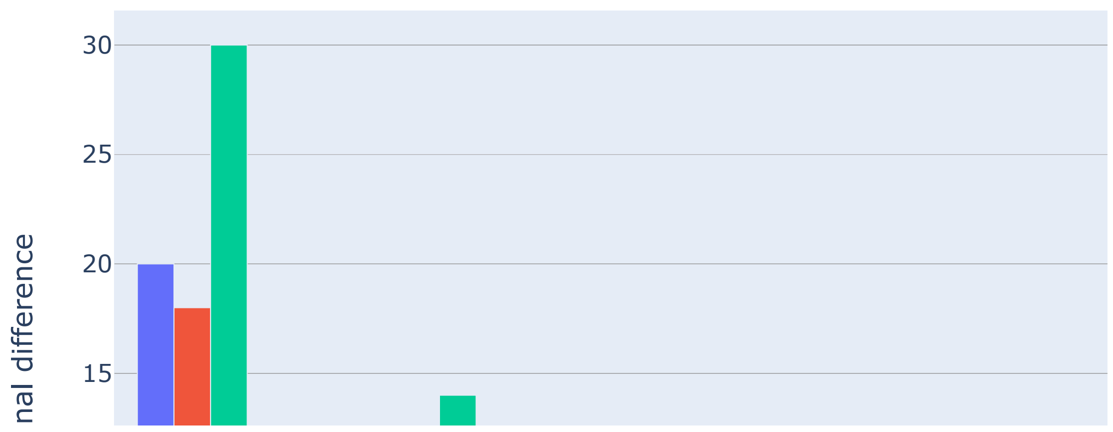
In [7]:

```
#Note: the pagerank functions can be found at the top of the file, as they were needed
 in assignment 3 already
def generate_indegree_score(graph):
    # Returns a list of in-degree score per node.
    in_degrees = []
    for node in graph.nodes:
        if type(graph) == nx.classes.digraph.DiGraph:
            in_degrees.append(graph.in_degree(node))
        else:
            in_degrees.append(graph.degree(node)) # To make this method not fail for un
directed graphs
    return in_degrees

def ordinal_difference(a, b):
    # calculate the difference between ordinal rankings, as defined in the text for ass
ignment 4 on Canvas
    if type(a) == list:
        a = np.array(a,dtype=float)
    if type(b) == list:
        b = np.array(b,dtype=float)
    return sum(abs(a-b))

def hits(M):
    # Our implementation of HITS
    if type(M) != np.matrix:
        # If the user accidentally uses a graph as input, convert to adjacency matrix
        M = nx.adjacency_matrix(M)
    v=np.ones(len(M),dtype=float) # generate v: a vector ones with length n, as per the
definition in the slides
    change_was_made = True
    h=np.copy(v)
    a=np.copy(v)
    while change_was_made:
        # M remains unchanged
        previous_a = np.copy(a)
        previous_h = np.copy(h)

        a = np.array(np.dot(M.transpose(),h)).flatten() # a' = M^T * h * 1/max(a') as p
er the definition
        a /= a.max()

        h = np.array(np.dot(M,a)).flatten() # h' = M * a * 1/max(h') as per the definit
ion
        h /= h.max()

        # Stop once the values don't change anymore
        change_was_made = ((abs(a-previous_a).max() ) > 0) or ((abs(h-previous_h).max()
) > 0)

    return a,h
```

In [8]:

```
prints = False # turn this to true to enable printing of more info!

graphs = [generate_test_web(), gen_lasso(), gen_arrow(),gen_inward(),gen_grid()] # Use
 these graphs in the exercise
differences=[]
for graph in graphs:
    M,v = generate_M_and_v(graph)
    if prints:
        print graph.name
        print str(v) + "\n"

    in_degrees = generate_indegree_score(graph)
    human_readable, indegree_order = order_nodes_by_rank(v, in_degrees)
    if prints:
        print "Solution to in_degree ranking:"
        print "Human readable ranking (nodes sorted by their rank): " + str(human_reada
ble)
        print "Ordinal ranking: " + str(indegree_order) + "\n"

    a , h = hits(M)
    human_readable, hits_order = order_nodes_by_rank(v, a)
    if prints:
        print "Our solution to HITS ranking"
        print "Human readable ranking: " + str(human_readable)
        print "Ordinal ranking: " + str(hits_order) + "\n"

        print "HITS ranking (networkx implementation, for reference)"
    (real_h_dict, real_a_dict) = nx.hits(graph,max_iter=100000)
    difference = []
    real_a=[]
    for i in range(len(v)):
        node = v[i]
        real_a.append(real_a_dict[node])
        difference.append(abs(real_a_dict[node]-a[i]))
    real_human_readable, real_hits_order = order_nodes_by_rank(v, real_a)
    if prints:
        print "Human readable ranking: " + str(real_human_readable)
        print "Ordinal ranking: " + str(real_hits_order)
        print "Ordinal ranking difference between our implementation and networkx: " +
str (ordinal_difference(hits_order,real_hits_order)) + "\n"


    page_rank_list = pagerank(graph,0.85)
    human_readable, pagerank_order = order_nodes_by_rank(v, page_rank_list)
    if prints:
        print "Our solution to pagerank"
        print "Human readable ranking: " + str(human_readable)
        print "Ordinal ranking: " + str(pagerank_order) + "\n"
        print "pagerank (networkx implementation, for reference)"
    (real_v_dict) = nx.pagerank(graph,max_iter=100000)

    difference = []
    real_v=[]
    for i in range(len(v)):
        node = v[i]
        real_v.append(real_v_dict[node])
        difference.append(abs(real_v_dict[node]-real_v[i]))

    real_human_readable, real_pagerank_order = order_nodes_by_rank(v, real_v)
    if prints:
        print "Human readable ranking: " + str(real_human_readable)
        print "Ordinal ranking: " + str(real_pagerank_order)
        print "Ordinal ranking difference between our implementation and networkx: " +
str (ordinal_difference(pagerank_order,real_pagerank_order))

    differences.append([ordinal_difference(indegree_order,hits_order),
                        ordinal_difference(indegree_order,pagerank_order),
                        ordinal_difference(hits_order,pagerank_order),
                        ordinal_difference(hits_order,real_hits_order),
                        ordinal_difference(pagerank_order,real_pagerank_order),
                        ])
    if prints:
        print("\n\n\n")


graphs=["Big (n=10) web", "Lasso", "Arrow", "Inward", "Grid"]
indegree_hits_diff = [x[0] for x in differences]
indegree_pagerank_diff = [x[1] for x in differences]
hits_pagerank_diff = [x[2] for x in differences]
hits_ours_vs_networkx_diff = [x[3] for x in differences]
pagerank_ours_vs_networkx_diff = [x[4] for x in differences]

fig = go.Figure(data=[
    go.Bar(name='In-degree vs HITS', x=graphs, y=indegree_hits_diff),
    go.Bar(name='In-degree vs PageRank', x=graphs, y=indegree_pagerank_diff),
    go.Bar(name='HITS vs PageRank', x=graphs, y=hits_pagerank_diff),
    go.Bar(name='Our HITS implementation vs networkx\'', x=graphs, y=hits_ours_vs_netwo
rkx_diff),
    go.Bar(name='Our PageRank implementation vs networkx\'', x=graphs, y=pagerank_ours_
vs_networkx_diff)
])

fig.update_layout(title='Difference between ranking algorithms for each graph',
                    autosize=True,
                    yaxis_title='ordinal difference')

# Change the bar mode
fig.update_layout(barmode='group')
fig.show()
```

Difference between ranking algorithms for each graph



Explanation for exercise 4

It is clear from this plot that there is no difference between our PageRank implementation and the one in networkx. Therefore we are quite confident that our implementation for PageRank is correct. The bar graph for this is always at 0.

There is a slight difference between our HITS algorithm for the "web" graph (all graphs can be seen a few cells above), and the implementation used in networkx. It is hard to find an explanation for this. It may be because of different execution counts, where the networkx implementation stops before it converges.

The HITS and PageRank seem to have the same ranking for Arrow. However, all algorithms are quite similar there. This makes sense, as the pages just link toward each other in a 1D line.

In the inward graph, all algorithms are the same. This makes sense: there is one graph with a high ranking (the middle), and all others have an equal ranking, where the tie is broken depending on which graph comes earlier in the list.

The bigger differences are seen in Grid, Lasso, and web. In Grid, the in-degree and PageRank algorithm are most similar. This can be expected, as their implementations are slightly similar. Furthermore, the graphs with a high in-degree here will also have a high PageRank due to the graph structure.

In Lasso, In-Degree and HITS are most similar. This, again, makes sense due to the graph structure, where one graph has a large in-degree and is thus a good authority.

For the Web graph, it is surprising how divergent the results for HITS and PageRank are. It seems these algorithms produce wildly different results for larger, complex graphs, even though they seem quite similar on a theoretical level. We did not expect this.

We do not expect these conclusions to always hold for graphs of greater complexity, as this adds lots of new factors and possible stumbling blocks for our algorithms. PageRank and HITS will probably give better rankings than In-Degree, but how similar they are really depends on the graph structure.