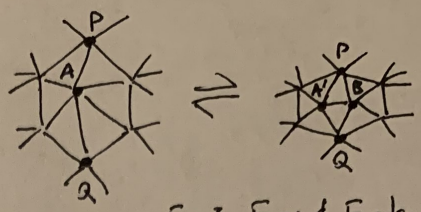


Fig.3 Facr & Fink

- node A can be "pushed" into A' & B
- node A' & B can be "poped" into A

push_pop

July 21, 2021

```
In [1]: import numpy as np
        import networkx as nx
        import matplotlib.pyplot as plt
        import random
```

```
In [2]: def get_adj(H):                                    (1)
            A = nx.adjacency_matrix(H)
            print(A.todense())
            print("\n")
            return

        def get_maxclique(H, num):
            check = False
            n = 1
            Y = nx.find_cliques(H)                         (2)
            for i in Y:
                #print(i)
                if len(i) != 3:                            (3)
                    check = True
                    print(i)
                if num == 0:                               (4)
                    plt.figure(n, figsize=(1,1))
                    nx.draw(H.subgraph(i).copy(), pos=nx.spring_layout(H), node_color='lightgr
                        edge_color='black', with_labels=True)
                    n += 1
            print("\n")
            return check


        def get_testgraph():                               (5)
            list = [i for i in range(0,8)]
            print(list)
            H = nx.Graph()
            for i in list:
                H.add_node(i)

            H.add_edge(0,1)
```

(1) function to return adjacency matrix from graph

(2) find_cliques finds all the cliques within the graph

(3) Did a check to make sure all cliques remain 3

(4) if input parameter is a zero, print out all the cliques graphically, rather than in list form.

(5) function to create a triangular planar graph like in fig 3 of Facr & Fink. (mainly for trouble shooting purposes)

```
H.add_edge(0,2)
H.add_edge(0,3)
H.add_edge(0,4)
H.add_edge(0,5)
H.add_edge(1,2)
H.add_edge(1,5)
H.add_edge(1,6)
H.add_edge(1,7)

H.add_edge(2,3)
H.add_edge(3,4)
H.add_edge(4,5)
H.add_edge(5,6)
H.add_edge(6,7)
H.add_edge(2,7)

nx.draw(H, pos=nx.spring_layout(H), node_color='lightgray', \
        edge_color='black', with_labels=True)
return H
```

```
In [4]: def push(G):
            check = True
        (6) while check is True:

            listofnodes = [k for k in G]          (7)
            node = random.choice(listofnodes)
            print('random node: ',node)

            nbr = [n for n in G[node]]            (8)
            print('neighbor: ', nbr)

            p = random.choice(nbr)
            for q in nbr:
                #print(q)
        (9)     boo = p in G.neighbors(q)
                if boo == False and p != q:
                    print('Chosen neighbors: ', p,q)
                    break

            if boo == True or p == q:
        (10)    print('ERROR: Did not find neighbors\n')
                continue
```

(6) while loop because sometimes the random chosen node cannot be "pushed" so the process will have to start over.

(7) get a list of all the nodes, nodes get added & deleted as graph evolves so the random choice has to be from this list.

(8) getting a list of neighbors of the randomly chosen nodes.

(9) - nodes P & Q (refer to fig 3 Farr & Fink) must have at least 1 node separating them.
- I randomly choose p from nbr and I loop through nbr to see if they are connected, if boo ("boolean") is false, meaning P and the current q are not neighbors (and p!=q), we can break out of loop.

(10) Sometimes we loop through nbr and none of the q is satisfactory, we "continue", meaning "check = True" throughout the rest of the while loop restarts the process.

(11) nodes are enumerated, max(G.nodes) gives the highest value of all current existing nodes. We name the new_node the next value.

(12) Remove all edges to A ("Farr & Fink fig 3")

(13) A' & B are always connected (fig 3)

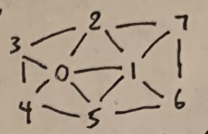
```
else:                                    (11)
    new_node = max(G.nodes)+1
    G.add_node(new_node)
    print('Adding new_node: ', new_node)

    for i in nbr:                        (12)
        G.remove_edge(node,i)
    #print('Removing all edges to ...', node)

    G.add_edge(node,new_node)            (13)
```

(14) Tricky part. Looking at the RHS of fig 3, we see half the neighbors belong to A' and the other half belong to B. However, networkx will list out the ~~nodes~~ neighbors in numerical ascending order.

Ex.



neighbors of node 1 will be [0, 2, 5, 6, 7] so its hard to split them into left & right half like in fig. 3.

(15) I sorted the list of neighbors("nbr") into a clockwise or anticlockwise manner, i.e. [0, 5, 6, 7, 2]

or

[0, 2, 7, 6, 5]

which makes it easier to distinguish the "left" and "right" side of nodes P & Q (fig 3).

```
sort_nbr = [nbr[0]]                      (14)
check1 = False
while check1 is False:
    for i in range(1,len(nbr)):          (15)                                    (17)
        if (nbr[i] in G.neighbors(sort_nbr[-1])) == True and (nbr[i] in so
            sort_nbr.append(nbr[i])
        if (nbr[i] in G.neighbors(sort_nbr[0])) == True and (nbr[i] in sor   (16)
            sort_nbr.insert(0,nbr[i])
        if len(sort_nbr) == len(nbr):
            check1 = True
            break
print(sort_nbr)
```

(16) - So first I created a list ("sort_nbr") and append the first element of the existing nbr list, just because the sorting needs to start somewhere.

- 1st if statement: if i is connected to the last element of sort_nbr, ~~appe~~ AND i is not already in sort_nbr, append.
- 2nd if statement: if i is connected to the first element of sort_nbr AND i not in sort_nbr already, append.
- This algorithm creates the clockwise/counter-clockwise ordering.

(17) We need the "not already in sort_nbr" condition because of the following:

nbr = [a, b, c, d]

sort_nbr = [a]

a – d – c – b

- As we loop through nbr, d gets appended first as it is connected to a.

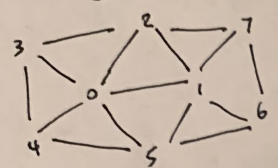
  sort_nbr = [a, d]

- next, c gets appended as it is connected to d, however, c is also connected to a, we'd end up with [a, d, c, c]

(18) - Create empty lists to store "left" & "right" side neighbors (fig3)
i.e. temp1 will store all neighbors to the left of nodes P & Q

- Sorting method:

(19)



Assume node A = 1
P = 2
Q = 6

sort_nbr = [0, 2, 7, 6, 5]

- retrieve list index of P & q
so P = 2 has index 1,
and q = 6 has index 3.
- all values in sort_nbr with index number between the index of P & q (i.e. 1 and 3), they are append into temp1.
- all other goes into temp2

(18)
```
temp1=[]
temp2=[]
if sort_nbr.index(p) < sort_nbr.index(q):
    for i in range(0,len(sort_nbr)):
        if i >= sort_nbr.index(p) and i <= sort_nbr.index(q):
(19)            temp1.append(sort_nbr[i])
        if i <= sort_nbr.index(p) or i >= sort_nbr.index(q):
            temp2.append(sort_nbr[i])
else:
    for i in range(0,len(sort_nbr)):
        if i <= sort_nbr.index(p) and i >= sort_nbr.index(q):
(20)            temp1.append(sort_nbr[i])
        if i >= sort_nbr.index(p) or i <= sort_nbr.index(q):
            temp2.append(sort_nbr[i])

print(temp1, temp2)

for i in temp1:                    (21)
    G.add_edge(node,i)

node_nbr = [i for i in G[node]]    (23)
#print('Neighbors of %i: ' %node, node_nbr)

for i in temp2:
    G.add_edge(new_node,i)         (21)
```

(20) if sort_nbr is [0, 5, 6, 7, 2]
P = 2 has index 4
and q = 6 has index 2,
because P now has a higher index than q, the previous
$p \leq$ values $\leq q$
wouldn't work, ~~instead~~
instead we need the condition to be
$q \leq$ values $\leq p$ and
append the values with separate conditions.

(21) Connecting nodes in temp1 with node A' (fig3),
Same with temp2 to B

(23) Storing neighbors of A' & B, not necessary but I printed them for troubleshooting purposes.

(24) Set check = False so while loop will stop since we've already "pushed" once successfully.

(25) Same thing, while loop because condition to "pop" might not be satisfactory somewhere.

```
new_node_nbr = [i for i in G[new_node]]   (23)
#print('Neighbors of %i: ' %new_node, new_node_nbr)

check = False   (24)
print('SUCCESS\n')
```

```
In [126]: def pop(G):
              check = True   (25)
              while check is True:
```

(26) Here's how I choose A' and B so they can be popped to just A. You start with a random chosen node and you get all of its neighbors, nbr. You want to check whether a neighbor of "random node" has two shared neighbors with "random node". i.e. A' is the random node, B is the neighbor from nbr, P & Q are the shared neighbors.

- So I loop through nbr, choose a neighbor, get the neighbors of that neighbor, which I call nbr2, and I check whether the elements of nbr2 are also in nbr.

(28) Check number of elements in nnbr because we need at least 2. Once nnbr has enough elements, the 2nd random node, which I called "node2" aka node B (fig 3) is just set to be whatever element in nbr that we are on.

```
listofnodes = [k for k in G]
node = random.choice(listofnodes)

nbr = [n for n in G[node]]
#print('neighbor: ', nbr)
```

```
nnbr = []   (26)
for i in nbr:
    nbr2 = [n for n in G[i]]
    #print(nbr2)
    for j in nbr2:
        if (j in nbr) == True and j != node:
            nnbr.append(j)   (27)
        else:
            continue
    if len(nnbr) >= 2:   (28)
        node2 = i
        break
    else:
        nnbr.clear()
                (29)
```

(29) If element i, or "node2" or node B does not have at least 2 shared neighbors, nnbr, clear the nnbr list and proceed to check the next element i within the nbr list.

(27) If a node is in both nbr & nbr2 but not being the random node, I append it to a list called nnbr. I keep checking because we need 2 elements in nbr2 that is also in nbr, i.e. P & Q.

(30) need another check here because sometimes one none of the elements in nbr satisfy the condition and nnbr just remains empty, we will have to restart from beginning.

(31) if there is enough shared neighbors, we can just set the first 2 elements as p & q.

```
(30) if len(nnbr) < 2:
         print('ERROR: The two chosen nodes did not have enough shared neighbors\
         continue

     print('Chosen random nodes: ',node, 'and', node2)
     p = nnbr[0]
(31) q = nnbr[1]
     print('Chosen neighbors: ',p,'and',q)
```

(32) I made copies of neighbors of A' ("nbr") and neighbors of B ("nbr2").

(33) We need to test whether neighbors of A' are connected with neighbors of B, excluding the "special" node which are A', B, P, and Q (aka "node", "node2", "p", and "q"). So I deleted the "special" node for temp1 and temp2, which are copies of nbr and nbr2.

(34) Start of algorithm to check if temp1 and temp2 have any ~~shared neighbors~~ connecting element because that's not allowed.

```
(32) temp1 = nbr.copy()
     temp2 = nbr2.copy()

     temp1.remove(p)
(33) temp1.remove(q)
     temp1.remove(node2)
     temp2.remove(p)
     temp2.remove(q)
     temp2.remove(node)
```

```
(34) check1 = False
     for i in temp1:
```

(35) In this line of code, i is an element of temp1 and j is an element of temp2, if i and j are connected, check1 = true.

```
            if check1 == True:
                print('break')          (36)
                break
            else:
                for j in temp2:
                    check1 = i in G.neighbors(j)   (35)
                    if check1 == True:
                        break
                    else:
                        continue

        if check1 == True:
            print("ERROR: Neighbors of node and node2 are connected.")
            print('Exiting...\n')       (37)
```

(36) This is here and below because once check1 = true, it will break out of the 2nd for loop, need another "break" to get out of 1st for loop.

(37) Once we break out of loop, we have to start over.

(38) If temp1 has no connecting elements with temp2, by the "pop" algorithm, node B disappears, A' becomes node A, and A gets all the neighbors of A' and B, which is temp1 + temp2.

```
        else:
            new_nbr = temp1 + temp2     (38)
            new_nbr.append(p)
            new_nbr.append(q)           (39)
            #print(new_nbr)

            for i in nbr:
                #print(i)                (40)
                G.remove_edge(node,i)
            print('Removing all edges to ...', node)

            for i in nbr2:
                if i != node:
                    G.remove_edge(node2,i)
            print('Removing all edges to ...', node2)

            G.remove_node(node2)

            for i in new_nbr:
                G.add_edge(node,i)
            print('Adding required edges...')
            print('SUCCESS\n')
            check = False               (41)
```

(39) P and Q are also part of new_nbr, we deleted them earlier on because they were "special" and does not need to be checked.

(40) Transformation process: delete all connection to A' and B, delete B, add all elements of new_nbr to A. In this case, I didn't really rename A' to A, it's just labels.

(41) Make check equal False to break out of while loop since we now have one successful pop.