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

Q1a: imposed gradient - first stabilize the gradient

We want to solve PDE

$$C_t = C_{xx}.$$

Using finite differences, it can be discretized using explicit scheme as

$$\frac{C_j^{i+1} - C_j^i}{\tau} = \frac{C_{j-1}^i - 2C_j^i + C_{j+1}^i}{h^2}$$

, where τ is the step over time, h is step over space (in Q1 space is repersented by N nodes that are at the distance $h = 1$ from each other), and C_j^i represents $C(t_i, x_j)$. We know that the explicit scheme is stable only if $\tau < 0.5h^2$, so in our time-marching loop we wil take $\tau = 0.4h^2$. For speeding-up computations, we will use vectorized operations. It means, that we want to calculate $C_j^{i+1} = C_j^i + \tau\frac{C_{j-1}^i - 2C_j^i + C_{j+1}^i}{h^2}$ for the whol massiv of C. This means that we have

$$C_{new} = C_{old} + \tau\frac{L@C.T}{h^2}$$

From discretization and two boundary conditions, that derivatives of C with respect to x are zero at both ends of our line of N nodes, Laplacian is given by a matrix

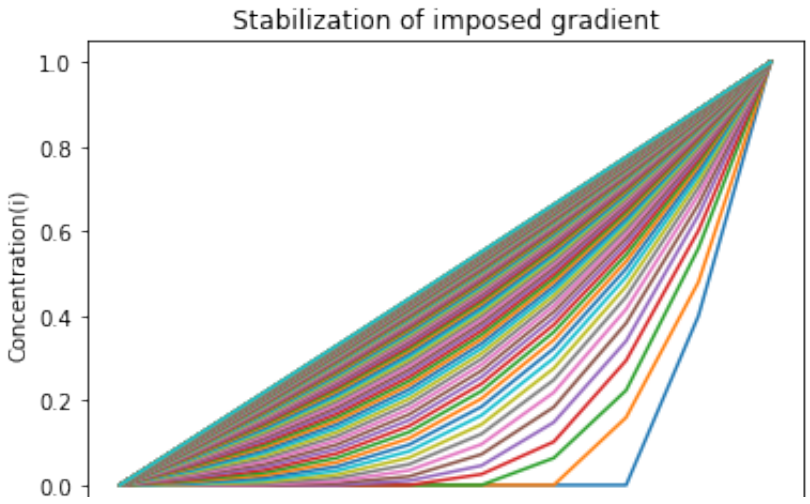
$$\begin{matrix} 1 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ \cdots & 0 & 0 & & -1 & 1 \end{matrix}$$

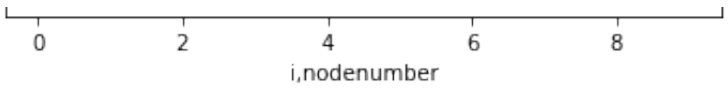
But from course MA933 by Susana we know, that Laplacian on a graph also admits reprecentation as $L = A - D$, where A is the adjacency matrix of a graph, and D is a diagonal matrix, where diag(i) is the degree of the $i-$th node. In our case (for graph being a line of N nodes), $D = diag([1, 2, \cdots, 2, 1])$, and A is a zero matrix with ones over and under the main diagona, so $L = A - D$ really gives the needed matrix, that we obtain from discretization by finite differences.

So let's do time-marching with such formula for C_{new}, but we also need to take into account that our boundary conditions are not that derivatives of C with respect to x at each of the ends equals zero, but instead that $C(0) = 0, C(N-1) = 1$, which means that left end is a sink, and right end is a reservoir. And initail condition is that C is equal zero everywhere except the reservoir (that is the last point).

In [143]:

```
def getAForLineGraph(N):
    A=np.zeros((N,N))
    for i in range(N):
        if i-1>=0:
            A[i,i-1]=1
        if i+1<=N-1:
            A[i,i+1]=1
    return A

def getDForLineGraph(N):
    return np.diag([1]+[2 for g in range(N-2)]+[1])


def get_imposed_gradient(N,Doplot=False):
    masC=np.zeros(N)
    masC[-1]=1
    A=getAForLineGraph(N)
    D=np.diag([1]+[2 for g in range(N-2)]+[1])
    L=A-D
    h=1
    tau=0.4*h**2
    t=0
    for s in range(100):
        t+=tau
        masC+=tau*np.matmul(L,masC.reshape(-1,1)).flatten()/h**2
        masC[0]=0 #because C_0=0 is a boundary condition
        masC[-1]=1 #because C_N=1 is a boundary condition
        if Doplot:
            plt.plot(np.arange(N),masC)
    if Doplot:
        plt.xlabel('i,nodenumber')
        plt.ylabel('Concentration(i)')
        plt.title('Stabilization of imposed gradient')
        plt.show()
    return masC

masC=get_imposed_gradient(N=10,Doplot=True)
print("Concentration after stabilization is :")
masC
```






```
Concentration after stabilization is :
```

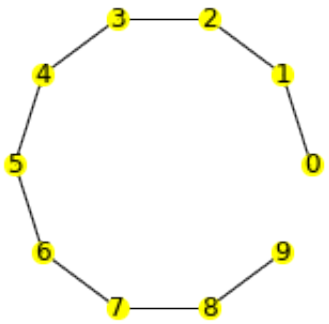
Out[143]:
```
array([0.        , 0.10957661, 0.21933831, 0.32944785, 0.44002604,
       0.55113715, 0.66278119, 0.77489387, 0.88735439, 1.        ]
)
```

We see that when time goes, the concentration becomes uniformly spread between zero at left end and one at right end - and that is what we intuitively expect.

Q1b: tumor sells under imposed gradient

First lets make in nx a graph of N=10 points in a line and plot it.

In [3]:
```
N=10 #amount of graph nodes in the line
G = nx.Graph()
for i in range(N):
    G.add_node(str(i))
    if i-1>=0:
        G.add_edge(str(i-1),str(i))
    if i+1<=N-1:
        G.add_edge(str(i),str(i+1))


plt.figure(figsize = (3,3))
pos=nx.circular_layout(G)
#pos = nx.fruchterman_reingold_layout(G);
#pos=nx.spring_layout(G)
#pos=nx.bipartite_layout(G,G.nodes,align='horizontal')
nx.draw_networkx(G, pos=pos, with_labels = True, node_color = 'yell
nx.draw_networkx_labels(G, pos=pos)
plt.axis('off');
```



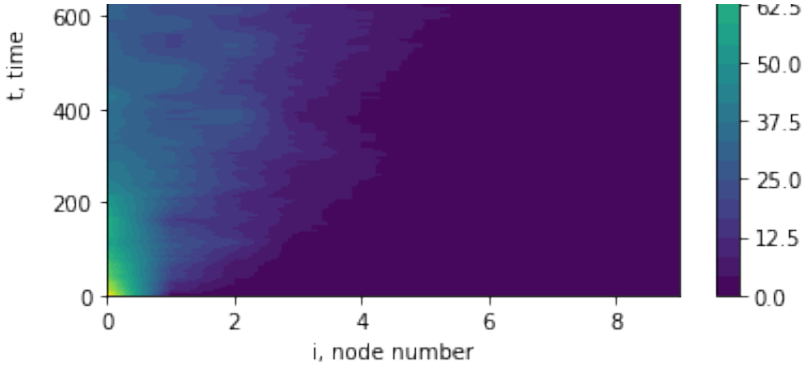
We need a rule for tumor sells to migrate from node to node in a graph. It seems reasonable that the greater the difference in concentrations between nodes i and j, the higher the chance that i will migrate to j and not somewhere else. So my choice of propensities is f(ci,cj)=e^{|c_i-c_j|}, and we can use list comprehension not to compute propensities for all pairs of nodes in a cycle --- see function def get_propensities(masC). And how actually the move happens: first we get the matrix of propensities and multiply it by the adjecency matrix (because tumor sells can move only along edges) and also multiply by the massiv of amounts of tumor sells at each of the node (because the larger the amount of tumor sells, staying in node i, the greater the chance that the tumor sell from i will move somwhere, and not a tumor sell from some other node.) After doing this, we sum up the propensities matrix by raw and simulate a random element using this massiv (divided by the sum of the massiv) - to find out FROM which node the tumor sell will move. And then we look at the row of propensities of the node FROM which a tumor sell will move - and use the propensities of this row (divided by the sum of this row) to simulate a random number that will say TO which node (chosen from the neighbours) will the tumor sell move.

N is again the number of sells in the graph in a line. K is the number of tumor points, that we place at the left end of the line in the beginning of experiment. And the concentrations we assume to be unchanged and equal to stable-state concentrations, found in Q1a.

In [4]:

```
def get_propensities(masC):
    return np.exp(np.abs(masC.reshape(-1,1)-masC))

N=10 #amount of graph nodes in the line
K=100 #amount of tumor sells
A=getAForLineGraph(N)
masC=get_imposed_gradient(N) #function from Q1
mas_tumor_sell_counts=np.array([K]+[0 for g in range(N-1)])

history_of_mas_tumor_sell_counts=[]
history_of_mas_tumor_sell_counts.append(np.copy(mas_tumor_sell_coun


h=1
tau=0.4*h**2
Tmax=1000

def run_imposed_grad(Tmax,masC,A,tau,h,N):
    t=0
    for s in range(1,Tmax):
        t+=tau
        matr_propensities=np.multiply(get_propensities(masC),A) #no
        tumor_sell_weighted_matr_propensities=np.multiply(mas_tumor
        tumor_sell_weighted_mas_propensities=tumor_sell_weighted_ma

        node_from=np.random.choice(a=np.arange(N),p=tumor_sell_weig

        from_sell_propensities=tumor_sell_weighted_matr_propensitie
        node_to=np.random.choice(a=np.arange(N),p=from_sell_propens

        mas_tumor_sell_counts[node_from]-=1
        mas_tumor_sell_counts[node_to]+=1
        history_of_mas_tumor_sell_counts.append(np.copy(mas_tumor_s

    print("After step",s+1, mas_tumor_sell_counts)
    return history_of_mas_tumor_sell_counts

history_of_mas_tumor_sell_counts=run_imposed_grad(Tmax,masC,A,tau,h

plt.contourf(np.arange(N),np.arange(Tmax),np.array(history_of_mas_t
plt.colorbar()
plt.xlabel('i, node number')
plt.ylabel('t, time')
plt.title('Spread of tumor sells under imposed gradient')
plt.show()
```

```
After step 1000 [30 14 21 14  8  8  2  1  2  0]
```



On the graph the x line to the right is the graph-nodes-on the line, and on the y axis, from bottom to top, is the time. And color shows the amount of tumor sells at each graph-node at each point of time. And we can see that the sells start from left and migrate right when time flows, which is what we expect to happen.

Q1c: tumor sells under self-generated gradient

Now our concentration is not constant (as was in case of imposed stable-state gradient), but insted concentration starts from being one everywhere and then diminishes according to the formula:

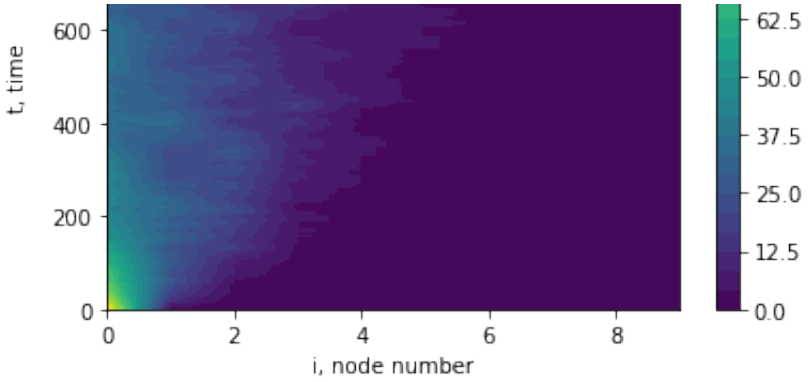
$$\frac{dC}{dt} = -N v_{max} \frac{C}{C + k_{max}} + \nabla C$$

, here N is the number of tumor sells, that stay at C_i. Since we already know how to deal with Laplacian, we can simulate this problem.

In [5]:
```
N=10 #amount of graph nodes in the line
K=100 #amount of tumor sells
A=getAForLineGraph(N)
D=getDForLineGraph(N)
L=A-D
masC=np.ones(N)
mas_tumor_sell_counts=np.array([K]+[0 for g in range(N-1)])

history_of_mas_tumor_sell_counts=[]
history_of_mas_tumor_sell_counts.append(np.copy(mas_tumor_sell_coun


h=1
tau=0.4*h**2
Tmax=1000
vmax=0.01
kmax=1

def run_self_generated_grad(Tmax,masC,A,tau,h,N,vmax,kmax, reservoi
    t=0
    for s in range(1,Tmax):
        # first determine tau
```

```
        #print(get_propensities(masC))
        matr_propensities=np.multiply(get_propensities(masC),A) #no
        tumor_sell_weighted_matr_propensities=np.multiply(mas_tumor
        sum_of_alphas=tumor_sell_weighted_matr_propensities.sum()

        r1=np.random.random()
        tau=np.log(1/r1)/sum_of_alphas
        if (tau>0.49*h**2):
            raise Exception('tau>0.49*h**2')
        t+=tau

        #now update masC (it has degraded a bit) and make a step of
        masC+=tau*(-mas_tumor_sell_counts*vmax*masC/(masC+kmax) + n
        for b in reservoir_points:
            masC[b]=1
        for b in dead_end_points:
            masC[b]=masC[b-1]

        matr_propensities=np.multiply(get_propensities(masC),A) #no
        tumor_sell_weighted_matr_propensities=np.multiply(mas_tumor
        tumor_sell_weighted_mas_propensities=tumor_sell_weighted_ma

        node_from=np.random.choice(a=np.arange(N),p=tumor_sell_weig

        from_sell_propensities=tumor_sell_weighted_matr_propensitie
        node_to=np.random.choice(a=np.arange(N),p=from_sell_propens

        mas_tumor_sell_counts[node_from]-=1
        mas_tumor_sell_counts[node_to]+=1
        history_of_mas_tumor_sell_counts.append(np.copy(mas_tumor_s

    print("After step",s+1, mas_tumor_sell_counts)
    print("Final concentrations:",masC)
    return history_of_mas_tumor_sell_counts

history_of_mas_tumor_sell_counts=run_self_generated_grad(Tmax,masC,

plt.contourf(np.arange(N),np.arange(Tmax),np.array(history_of_mas_t
plt.colorbar()
plt.xlabel('i, node number')
plt.ylabel('t, time')
plt.title('Spread of tumor sells under self-generated gradient')
plt.show()
```

```
After step 1000 [24 27 14 11  8  8  5  2  0  1]
Final concentrations: [0.44882391 0.49505247 0.57457009 0.67615308
0.76922597 0.83959799
 0.90130772 0.94814524 0.97330655 0.98318472]
```

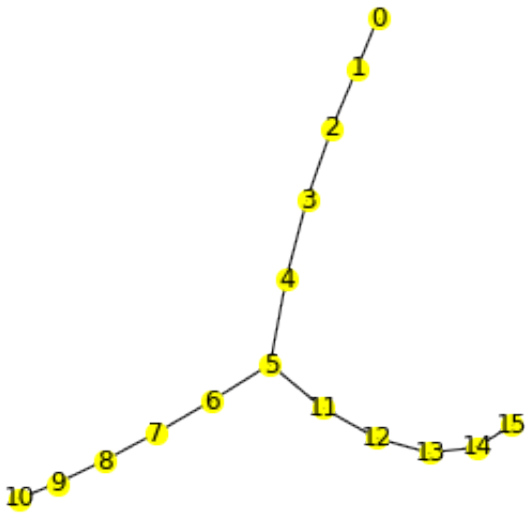


We see that in the self-generated gradient case sells also migrate from left to right, maybe a bit faster, than in case with imposed gradient.

Q2a: a branch with two imposed gradients

Now instead of 10 graph-nodes in a line, lets have 5 nodes in a line and then a junction, leading to two branches each of lencgth 5. Let's plot a picture of this configuration.

In [107]:
```
N=5*3+1

def getGTwoBraches(N):
    G = nx.Graph()
    for i in range(N//3*2):
        G.add_node(str(i))
        if i-1>=0:
            G.add_edge(str(i-1),str(i))
        if i+1<=N-1:
            G.add_edge(str(i),str(i+1))
    for i in range(N//3*2+1,N):
        G.add_node(str(i))
        if i-1>=N//3*2+1:
            G.add_edge(str(i-1),str(i))
        if i+1<=N-1:
            G.add_edge(str(i),str(i+1))
    G.add_edge(str(N//3),str(N//3*2+1))
    return G

G=getGTwoBraches(N)
plt.figure(figsize = (5,5))
#pos=nx.circular_layout(G)
#pos = nx.fruchterman_reingold_layout(G);
pos=nx.spring_layout(G)
#pos=nx.bipartite_layout(G,G.nodes,align='horizontal')
nx.draw_networkx(G, pos=pos, with_labels = True, node_color = 'yell
nx.draw_networkx_labels(G, pos=pos)
plt.axis('off');
```

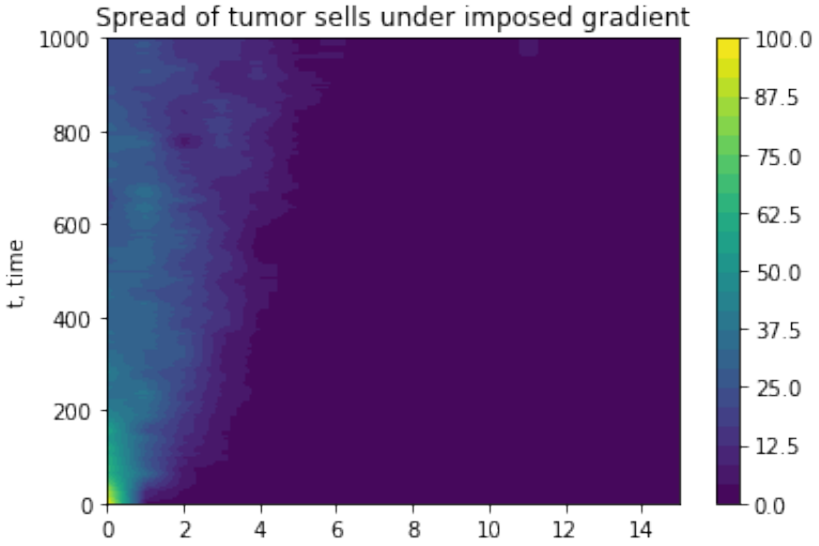


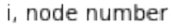
Now to run the imposed-gradient code we simply need to make a function to get the adjacency matrix of this graph (which is easy from the picture) and specify that both branches have reservoirs at the ends.

In [47]:

```
def getAFromGraph(G):
    N=len(G.nodes)
    A=np.zeros((N,N))
    for i in G.adj:
        for j in G.adj[i]:
            A[int(i)][int(j)]=1
    return A

N=5*3+1
G=getGTwoBraches(N)
A=getAFromGraph(G)
D=np.diag([el[1] for el in G.degree()])
L=A-D
mas=get_imposed_gradient(N//3*2+1)
masC=np.concatenate([mas,mas[N//3+1:]])
#print(masC)
K=100 #amount of tumor sells

mas_tumor_sell_counts=np.array([K]+[0 for g in range(N-1)])

history_of_mas_tumor_sell_counts=[]
history_of_mas_tumor_sell_counts.append(np.copy(mas_tumor_sell_coun


h=1
tau=0.4*h**2
Tmax=1000

history_of_mas_tumor_sell_counts=run_imposed_grad(Tmax,masC,A,tau,h

plt.contourf(np.arange(N),np.arange(Tmax),np.array(history_of_mas_t
plt.colorbar()
plt.xlabel('i, node number')
plt.ylabel('t, time')
plt.title('Spread of tumor sells under imposed gradient')
plt.show()
```

```
After step 1000 [23 21 13 15 11  3  5  1  0  0  0  4  3  1  0  0]
```





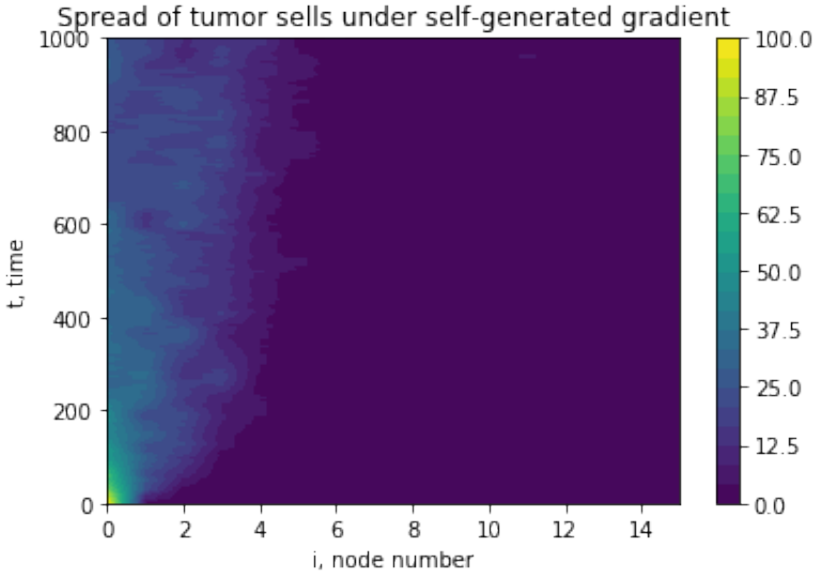
We see that the sells migrate from left to right and also that to branches ones and two went approximately the same number of tumor sells (9 vs 8)

Q2b: a branch with two self-generated gradients (both ends are reservoirs, that is C =1 for all t)

Now use computed in Q2a adjacency matrix A to compute Laplacian L and run code with degradation of concentration (that is, with self-generated gradient).

In [48]:
```
N=5*3+1
G=getGTwoBraches(N)
A=getAFromGraph(G)
D=np.diag([el[1] for el in G.degree()])
L=A-D
mas=get_imposed_gradient(N//3*2+1)
masC=np.concatenate([mas,mas[N//3+1:]])
#print(masC)
K=100 #amount of tumor sells

mas_tumor_sell_counts=np.array([K]+[0 for g in range(N-1)])

history_of_mas_tumor_sell_counts=[]
history_of_mas_tumor_sell_counts.append(np.copy(mas_tumor_sell_coun


h=1
tau=0.4*h**2
Tmax=1000
vmax=0.01
kmax=1

history_of_mas_tumor_sell_counts=run_self_generated_grad(Tmax,masC,

plt.contourf(np.arange(N),np.arange(Tmax),np.array(history_of_mas_t
plt.colorbar()
plt.xlabel('i, node number')
plt.ylabel('t, time')
plt.title('Spread of tumor sells under self-generated gradient')
plt.show()
```

```
After step 1000 [24 23 11 14 13  2  5  1  0  0  0  4  2  0  1  0]
Final concentrations: [0.10346061 0.13233279 0.18985485 0.27544343
0.39794223 0.54795627
 0.63012967 0.71817161 0.81142896 0.90531744 1.         0.62678297
 0.71610497 0.80897402 0.90301117 1.        ]
```



We see that to branches ones and two went approximately the same number of tumor sells (8 vs 7); But in Tweedy thay had that in self-generated gradint case the tumor sells moved faster, it is strange that in my pictures I don't see such behaviour.

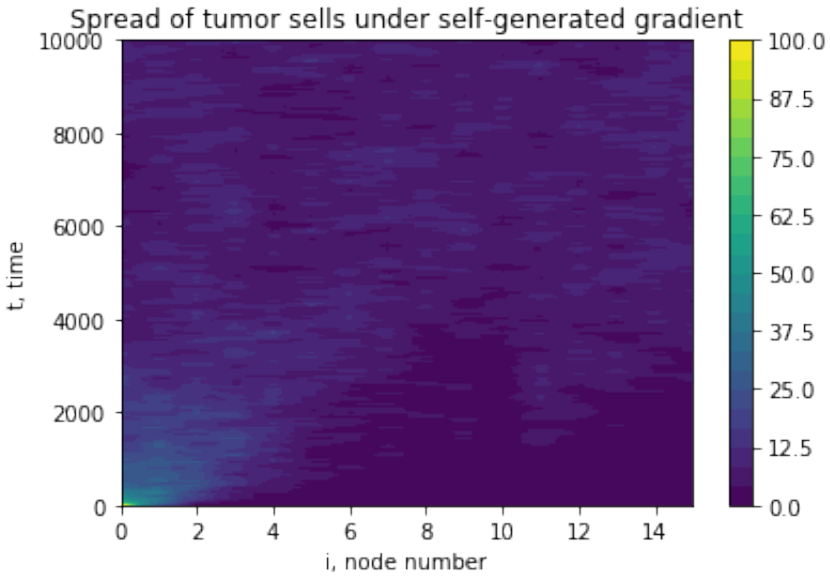
Q2c: a branch with two self-generated gradients (upper brancg is reservoir, lower is dead-end)

Now make (upper brancg is reservoir, lower is dead-end) - that is C(N//3*2)==1 for all t.

In [56]:

```
N=5*3+1
G=getGTwoBraches(N)
A=getAFromGraph(G)
D=np.diag([el[1] for el in G.degree()])
L=A-D
mas=get_imposed_gradient(N//3*2+1)
masC=np.concatenate([mas,mas[N//3+1:]])
#print(masC)
K=100 #amount of tumor sells

mas_tumor_sell_counts=np.array([K]+[0 for g in range(N-1)])

history_of_mas_tumor_sell_counts=[]
history_of_mas_tumor_sell_counts.append(np.copy(mas_tumor_sell_coun


h=1
tau=0.4*h**2
Tmax=10000
vmax=0.01
kmax=1

history_of_mas_tumor_sell_counts=run_self_generated_grad(Tmax,masC,

plt.contourf(np.arange(N),np.arange(Tmax),np.array(history_of_mas_t
plt.colorbar()
plt.xlabel('i, node number')
plt.ylabel('t, time')
plt.title('Spread of tumor sells under self-generated gradient')
plt.show()
```

```
After step 10000 [11  4  6  8  6  7  7  6  6  6  7  2  6  4  6  8]
Final concentrations: [0.15793605 0.16942835 0.18859195 0.21503866
0.25417685 0.30533977
 0.26180991 0.23284634 0.21504273 0.20670145 0.20670145 0.41085583
 0.52346293 0.66047543 0.81541129 1.        ]
```



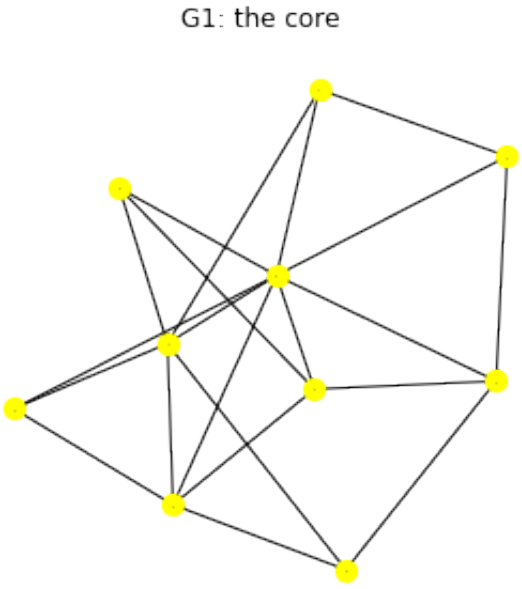
we see that concentration is right (it goes to one in down branch and it is 0.2 in the upper branch, where there is a the dead, but for some reason tumor points don't seem to tend to down branch more)

Q3: sponge

Now make a 2D simplified graph of a sponge: G1 is core, consiting of 10 nodes, each with 3 edges (I am using Barabasi-Albert model from Susana to make them up), G2 is the oughter layer, and G3, consisting of three points, is a bridge between the core and the oughter layer.

In [104]:
```
G1=nx.barabasi_albert_graph(10,3)
print("G1 nodes:", G1.nodes)
G2=nx.barabasi_albert_graph(10,3)
print("G2 nodes:", G2.nodes)

plt.figure(figsize = (5,5))
pos=nx.spring_layout(G1)
nx.draw_networkx(G1, pos=pos, with_labels = True, node_color = 'yel
plt.axis('off');
plt.title('G1: the core')
plt.show()
```

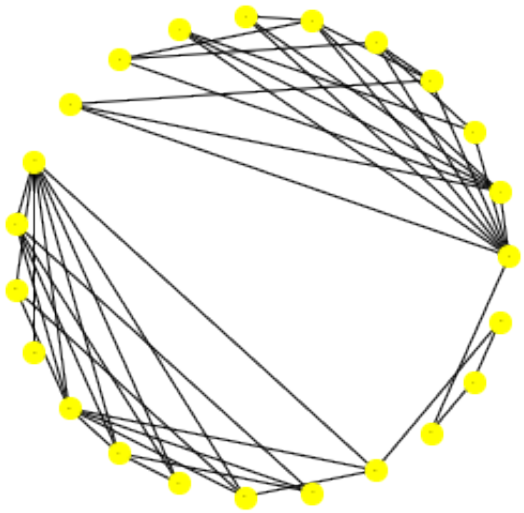
```
G1 nodes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
G2 nodes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```



Lets joint core (G1) and outer layer(G2) and add three nodes for bridge

In [105]:
```
G3 = nx.disjoint_union(G1,G2)
print(G3.nodes)

U=2*10
G3.add_edge(0,U)
G3.add_edge(U,U+1)
G3.add_edge(U+1,U+2)
G3.add_edge(U+2,U-1)

plt.figure(figsize = (5,5))
pos=nx.circular_layout(G3)
nx.draw_networkx(G3, pos=pos, with_labels = True, node_color = 'yel
plt.axis('off');
```

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

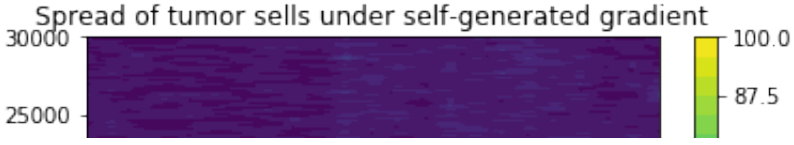


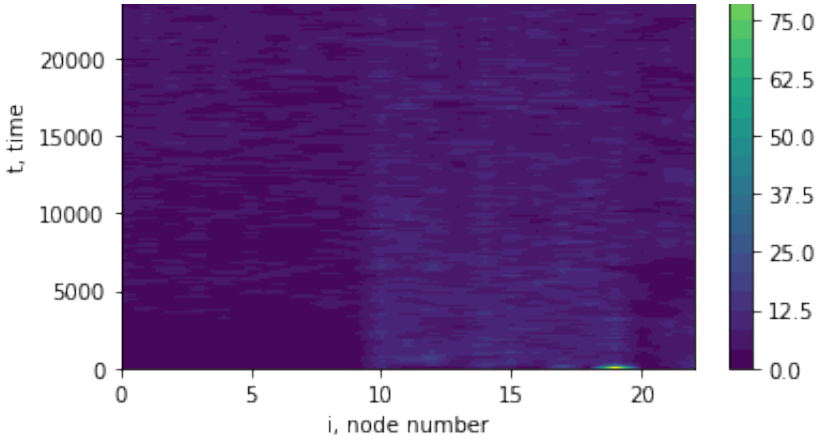
Now run this graph with imposed gradients and masC, which is equal to 1 at core and zero at the outer layer

In [128]:
```
```

```
N=len(G3.nodes())
print("N=",N)

A=getAFromGraph(G3)
D=np.diag([el[1] for el in G3.degree()])
L=A-D

masC=np.zeros(N)
for i in range(10):
    masC[i]=1
print("InitialConcentration=",masC)
K=100 #amount of tumor sells

mas_tumor_sell_counts=np.zeros(N)
mas_tumor_sell_counts[N-4]=K
print('Intial tumor sell locations', mas_tumor_sell_counts)

history_of_mas_tumor_sell_counts=[]
history_of_mas_tumor_sell_counts.append(np.copy(mas_tumor_sell_coun


h=1
tau=0.4*h**2
Tmax=30000
vmax=0.01
kmax=1

history_of_mas_tumor_sell_counts=run_self_generated_grad(Tmax,masC,

plt.contourf(np.arange(N),np.arange(Tmax),np.array(history_of_mas_t
plt.colorbar()
plt.xlabel('i, node number')
plt.ylabel('t, time')
plt.title('Spread of tumor sells under self-generated gradient')
plt.show()
```

```
N= 23
InitialConcentration= [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0
. 0. 0. 0. 0. 0. 0. 0. 0.]
Intial tumor sell locations [  0.   0.   0.   0.   0.   0.   0.
0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0. 100.   0.   0.   0.]
After step 30000 [7. 2. 1. 1. 2. 5. 4. 6. 6. 4. 8. 6. 3. 5. 2. 6.
8. 5. 6. 6. 1. 2. 4.]
Final concentrations: [0.0564088  0.05708374 0.05741496 0.05719587
0.05717846 0.05673447
 0.05685065 0.05677751 0.05694514 0.05705583 0.03518984 0.03494601
 0.0353346  0.03458692 0.03541518 0.03506476 0.03457252 0.03560357
 0.03495454 0.03680978 0.05199758 0.04730769 0.04202107]
```





In [137]:
```
print("Points in the core:", sum(history_of_mas_tumor_sell_counts[-
```

```
Points in the core: 38.0
```

In [139]:
```
print("Points in the outer layer:", sum(history_of_mas_tumor_sell_c
```

```
Points in the outer layer: 55.0
```

In [140]:
```
print("Points in the bridge:", sum(history_of_mas_tumor_sell_counts
```

```
Points in the bridge: 7.0
```

We see that by the time when concentration depleted due to degradation, approximately half of the points left the outer layer and went to the core; it looks a little strange, because I expected the vast majority of cells to reach the core being attracted by chemical, but maybe my configuration of sponge is not very realistic (maybe one bridge is not enough, or maybe Kmax and vmax should be more realistic, or more nodes in ht core and in the outer layer are neeeded)

In [ ]:
```

```