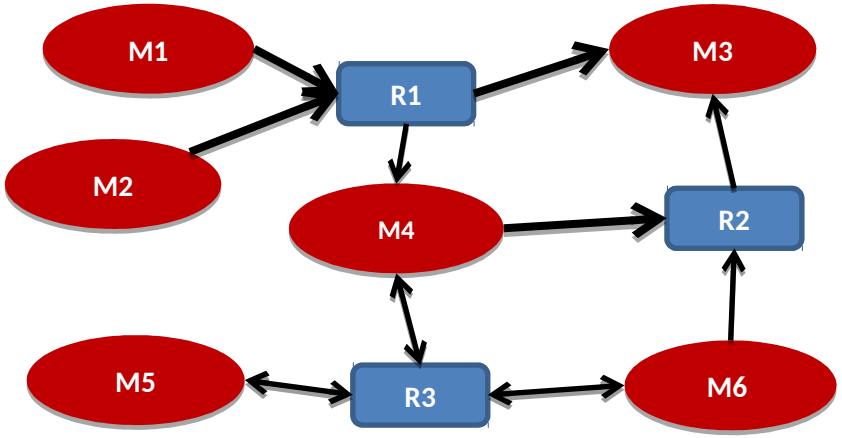
Redes biológicas

Tipos de redes, construção, análise topológica

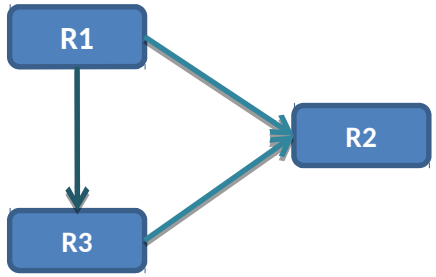
Redes metabólicas



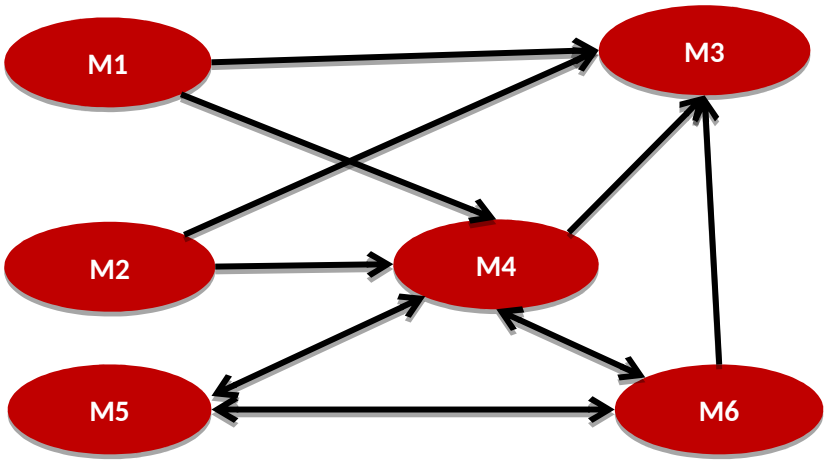
Sistema metabólico:

R1: M1 + M2 => M3 + M4
R2: M4 + M6 => M3
R3: M4 + M5 <=> M6

Rede de **metabolitos e reações**



Rede de **reações**



Rede de **metabolitos**

Implementando redes metabólicas

- Com base no código genérico para grafos orientados (classe **MyGraph**) vamos criar uma classe para representar e analisar os vários tipos de rede metabólica
- Esta classe será uma **sub-classe** da classe MyGraph, **herdando** assim todos os seus atributos e métodos

Implementando redes metabólicas

```
from MyGraph import MyGraph

class MetabolicNetwork (MyGraph):

    def __init__(self, networktype = "metabolite-reaction", graph = {}):
        ...
```

Classe ***MetabolicNetwork*** irá representar redes metabólicas, extendendo classe ***MyGraph***

Atributos da classe:

- ***graph*** – dicionário herdado da classe MyGraph
- ***net_type*** – tipo de rede: “metabolite-reaction”, “reaction”, “metabolite”

Implementando redes metabólicas

- Redes metabólicas serão criadas no nosso código a partir de 3 ficheiros de texto:
  - Ficheiro definindo **metabolitos**, um por linha
    - *Meta_id*
  - Ficheiro definindo **reações** uma por linha:
    - *R_ID, lower_bound, upper_bound*
  - Ficheiro com a **matriz** estequiométrica

    -1 1 0

    0 1 -1

  Exemplo: ficheiros "*exemplo-metab.txt*", "*exemplo-reac.txt*" e "*exemplo-mat.txt*" representam rede no exemplo de um slide anterior

Implementando redes metabólicas

```
def load_from_files(self, ...)
        ...

#auxiliary functions (outside of class)
def read_file_rm (filename, sep = ","):
        open file
        create a empty dictionary as dict
        reac_ids
        for each line:
                split the tokens by sep separator
                add the reaction id to the list reac_ids
                populate the dictionary where the key is the first token and the rest
                of the line are stored as a list in the value of the dict[key]
        return reac_ids, dict
```

Funções auxiliares para ler ficheiros anteriores:

- **read_file_rm**: lê ficheiros de reações ou metabolitos, criando lista de ids e dicionário com ids como chaves e atributos associados como valores
  - Ex: output the

Implementando redes metabólicas

```
def load_from_files(self, ...)
        ...

#auxiliary functions
def read_file_mat (filename, sep = "\t"):
        open files
        for each line:
                split like by sep and populate the tuple
                add tuple to the result list
        return result
```

- **read_file_mat**: lê ficheiro representando a matriz, dando uma lista de tuplos. Cada tuplo tem 3 elementos:
  (index_metabolite, index_reaction, coeficiente_value)

Implementando redes metabólicas

Método ***load_from_files*** – irá criar a rede metabólica a partir dos 3 ficheiros anteriores.

```
        def load_from_files(self, reacs_file, meta_file, matrix_file):
                reacs_info = read_file_rm(reacs_file)
                meta_info = read_file_rm(meta_file)
                mat = read_file_mat(matrix_file)
                create a aux_graph with information of reacs and meta in the nodes;
                if network type is “metabolite-reaction”:
                        self.g = aux_graph
....

def  get_all_graph(meta_ids, reacs_ids,reacs_attrs, matrix):
        create empty graph
        add all reactions ids as a node
        add all metabolites ids as a node
        for each tuple in mat:
                get meta_id and reac_id based on the indexes present in tuple
                get direction from tuple (<0 meta is consumed, >0 meta is
                        produced)
                get lower_bound of reaction
                if meta is consumed or lower_bound <0  add edge (meta_id, reac_id)
                if meta is produced or lower_bound <0 add edge(reac_id, meta_id)
        end for
        return graph
```

Implementando redes metabólicas

```
mrn = MetabolicNetwork("metabolite-reaction", {})
mrn.load_from_files("exemplo-reac.txt", "exemplo-metab.txt", "exemplo-mat.txt")
print ("Metabolite-reaction network:")
mrn.print_graph()
```

Vamos testar o nosso código com os ficheiros exemplo

Verifique se o grafo imprimido corresponde ao esperado

Como fazer para criar os outros dois tipos de rede ? :

- Só Metabolitos
- Só Reações

Sugestão: pense como criar as redes anteriores tendo já criada a rede reações-metabolitos

Implementando redes metabólicas

```
else if network type is “metabolite”:
        for each meta in meta_ids
                L= get all reactions that are successors of meta
                for each reac present in L
                        meta_dest = get all metabolites that are sucessors of reac
                        for each metaD from meta_dest
                                insert a new edge from meta to metaD

else if network type is “reaction”:
        … the same  as previous but considering the reactions instead
 metabolites
```

Implementando redes metabólicas

```
mn = MetabolicNetwork("metabolite", { })
mn.load_from_files("exemplo-reac.txt", "exemplo-metab.txt", "exemplo-mat.txt")
print ("Metabolite network:" )
mn.print_graph()


mr = MetabolicNetwork("reaction", { })
mr.load_from_files("exemplo-reac.txt", "exemplo-metab.txt", "exemplo-mat.txt")
print ("Reaction network:" )
mr.printGraph()
```

De novo, verifique se os grafos correspondem ao esperado ...

Redes metabólicas: exemplo

- Para testar o código anterior com um exemplo maior, use os ficheiros: "*ijr904-metab.txt*", "*ijr904-reac.txt*" e "*ijr904-matrix.txt*"
- Estes foram criados a partir de um modelo de escala genómica de *Escherichia coli (iJR904)*
- Os ficheiros têm a mesma estrutura dos anteriores pelo que o código deve poder ser usado sem grandes alterações
- Verifique o tamanho dos grafos gerados

Implementando redes metabólicas

```
ecoli_mrn = MetabolicNetwork()
ecoli_mrn.load_from_files("ijr904-reac.txt", "ijr904-metab.txt", "ijr904-matrix.txt")
ecoli_mrn.print_graph()
print (ecoli_mrn.size())
```

```
def size(self):
        return len(self.get_nodes()), len(self.get_edges())
```

Função adicionada a classe MyGraph

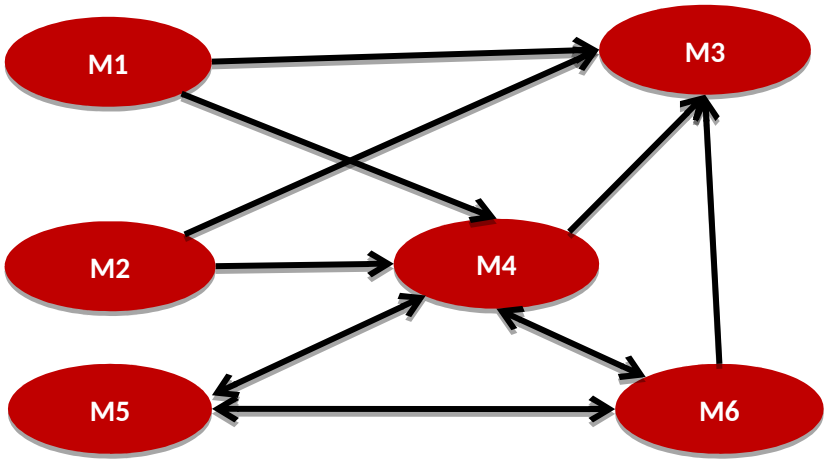
```
ecoli_mn = MetabolicNetwork("metabolite",{})
ecoli_mn.load_from_files("ijr904-reac.txt", "ijr904-metab.txt", "ijr904-matrix.txt")
ecoli_mn.print_graph()
print (ecoli_mn.size())

ecoli_mr = MetabolicNetwork("reaction", {})
ecoli_mr.load_from_files("ijr904-reac.txt", "ijr904-metab.txt", "ijr904-matrix.txt")
ecoli_mr.print_graph()
print (ecoli_mr.size())
```

Análise topológica de redes: graus e distribuição de graus

- **Grau médio <k>** - média do grau calculada sobre todos os nós (pode ser calculado apenas para graus de entrada/ saída em grafos orientados)
- **Distribuição do grau P(k)**: probabilidade que um nó tenha grau k. P(k) é independente do tamanho da rede
- Redes **"Scale-free"**
  - Distribuição dos graus aproxima a *power law* $P(k) \sim k^{-\gamma}$ $(2<\gamma<3)$
  - Isto implica que há poucos nós com muitas ligações e muitos nós com poucas ligações

Implementação de graus

- Graus “in” “out” e “inout”



degrees

In : {'M1': 0, 'M2': 0, 'M3': 4, 'M4': 4, 'M5': 2, 'M6': 2}

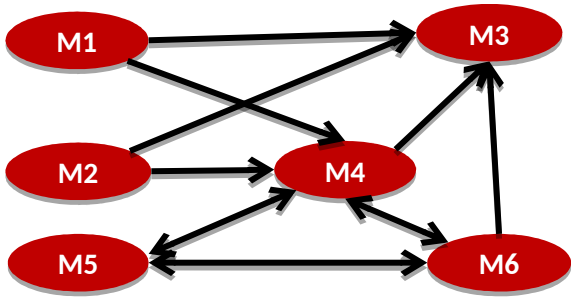
Out: {'M1': 2, 'M2': 2, 'M3': 0, 'M4': 3, 'M5': 2, 'M6': 3}

InOut:{'M1': 2, 'M2': 2, 'M3': 4, 'M4': 5, 'M5': 2, 'M6': 3}

Implementação de graus

Cálculo de graus de entrada e saída (ou ambos) para todos os nós da rede
Adicionar à classe *MyGraph*

```
def all_degrees (self, deg_type = "inout"):
        create a empty dictionary
        for each node of graph
                if the deg_type is “out” or ”inout”
                        insert in dictionary the number of out
                        degrees of node
                        (Note: number of out degrees is the number
                        of successors)

                if the deg_type is “in” or ”inout”
                        add to the degree of each successor of node
                        the value 1 if deg_type is “in” or node aren’t
                        a successor
                end for
        return degs
```



ligações em dois sentidos só contam como 1 grau

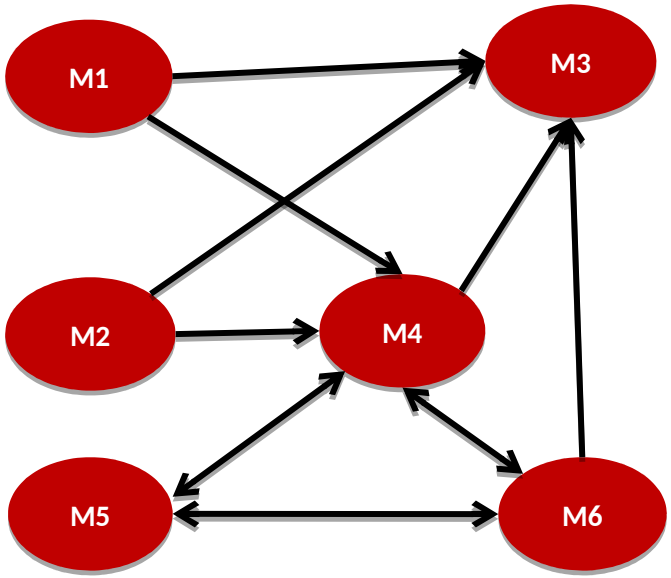
Implementação de graus

```
def mean_degree (self, deg_type = "inout"):
        calculate all_degrees
        result is sum all degrees / number of nodes

def prob_degree (self, deg_type = "inout"):
        degs = calcula all_degrees
        res = {}
        for each node
                get the number of degree
                if degree in res sum 1 else insert  the degree✉1 in res
        divide each value of res for the number of nodes.
        return res
```

```
print (ecoli_mrn.mean_degree("out"))
d = ecoli_mrn.prob_degree("out")
for x in sorted(d.keys()):
        print (x, "\t", d[x])
```

Aplique as funções anteriores à rede criada para a *E. coli*.

Implementação de distância média



Node: M1
[('M3', 1), ('M4', 1), ('M5', 2), ('M6', 2)]
Node: M2
[('M3', 1), ('M4', 1), ('M5', 2), ('M6', 2)]
Node: M3
[]
Node: M4
[('M3', 1), ('M5', 1), ('M6', 1)]
Node: M5
[('M4', 1), ('M6', 1), ('M3', 2)]
Node: M6
[('M3', 1), ('M4', 1), ('M5', 1)]

Distância média e proporção de nós atingíveis
(1.2941176470588236, 0.5666666666666667)

Implementação de distância média

Adição ao código anterior (*MyGraph*)
Ignora nós não atingíveis

```
def mean_distances(self):
        total = 0
        num_reachable = 0
        for each node
                d = calculate the distance from node to all other nodes of the
                        graph (use function reachableWithDist(node))
                sum all distances of d to total
                sum to num_reachable the number of nodes that can de
                reachable from node

        meandist = total/ num_reachable
        n = number of nodes in the graph
        return meandist, float(num_reachable)/((n-1)*n)
```

distância média

proporção de nós atingíveis

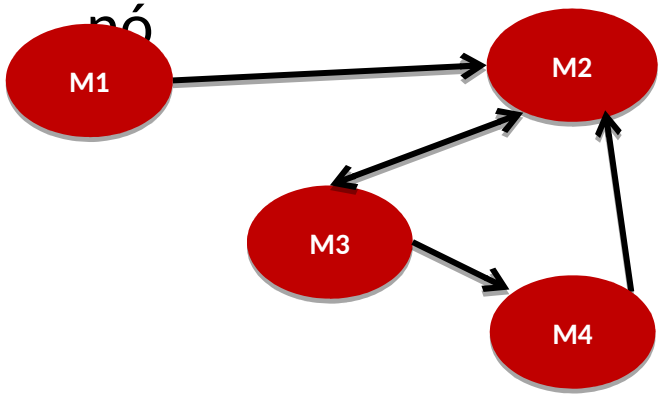
```
print (ecoli_mrn.meanDistances())
```

Implementando grafos: clustering

- **coeficiente de clustering** =

  n° de arcos existentes entre adjacentes do nó /

  n° total de arcos que poderiam existir entre vizinhos do nó



clustering_coef for M1:

n° adjacentes = 1

Result: 1.0

clustering_coef for M2:

n° adjacentes: 3

n° ligações entre adjacentes: 1 (3,4)

n° ligações possíveis: n*(n-1) = 3*2

Result:0. 16

Implementando grafos: clustering

```
def clustering_coef(self, v):
        adjs calcula adjacentes
        if size adjs is 0 return 0
        if size adjs is 1 return 1
        ligs = 0
        for each combination os 2 nodes in adjs
                if there is a edge between the 2 nodes
                        increase the number of ligs
        return float(ligs)/(len(adjs)*(len(adjs)-1))
```

```
gr = MyGraph( {1:[2], 2:[3], 3:[2,4], 4:[2]} )
print (gr.clustering_coef(1))
print (gr.clustering_coef(2))

gr2 = MyGraph( {1:[2,3], 2:[4], 3:[5], 4:[], 5:[]} )
print (gr2.clustering_coef(1))

gr3 = MyGraph( {1:[2,3], 2:[1,3], 3:[1,2]} )
print (gr3.clustering_coef(1))
```

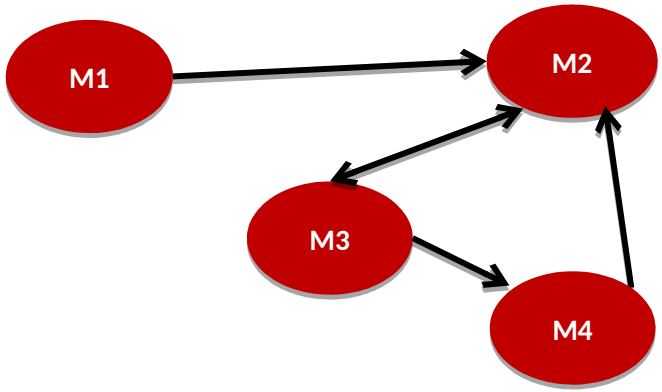
```
def all_clustering_coefs (self):
        run the clustering_coefs for all nodes

def mean_clustering_coef(self):
        return sum all_clustering_coefs / number of nodes
```

Calcula todos os coeficientes

Média global dos coeficientes na rede

Implementando grafos: clustering

- Calculo da média coeficiente de clustering para todos os graus



Grau "inout"
{'n1': 1, 'n2': 3, 'n3': 2, 'n4': 2}

Coeficientes de clustering
{'n1': 1.0, 'n2': 0.1666666666666666, 'n3': 0.5, 'n4': 1.0}

Grau (novo formato)
{1: ['n1'], 3: ['n2'], 2: ['n3', 'n4']}

Média de clustering por grau
Grau: sum(coef_clust_nodos_de_grau)/num_nodos

{1: 1.0, 3: 0.1666666666666666, 2: 0.75}

Implementando grafos: clustering

```
def mean_clustering_per_deg (self, deg_type = "inout"):
        degs =  calculate all degrees ( function all_degrees)
        ccs = calculate all clustering coeficients
        degs_k = {}

        convert the dictionary degs={node: degree} to degs_node={degree: list_nodes}

        results = {}  #{degree:  mean_of_clustering}

        for each degree in degs_node
                total =0
                for each node with this degree
                        sum the clustering coef of this node to the total
                the result of this degree is the total / number of nodes of this degree

        return result
```

Implementando grafos: clustering

```
ecoli_mn = MetabolicNetwork("metabolite",{ })
ecoli_mn.load_from_files("ijr904-reac.txt", "ijr904-metab.txt", "ijr904-matrix.txt")
ecoli_mn.print_graph()
print (ecoli_mn.size())

print (ecoli_mn.all_clustering_coefs())
print (ecoli_mn.mean_clustering_coef())
dc = ecoli_mn.mean_clustering_per_deg()
for cc in sorted(dc.keys()):
        print (cc, "\t" , dc[cc])
```

Analise os resultados. O que conclui ?

Porque não foi usada a rede metabolitos-reações ?
Que resultados iria obter ?