Programação em Lógica

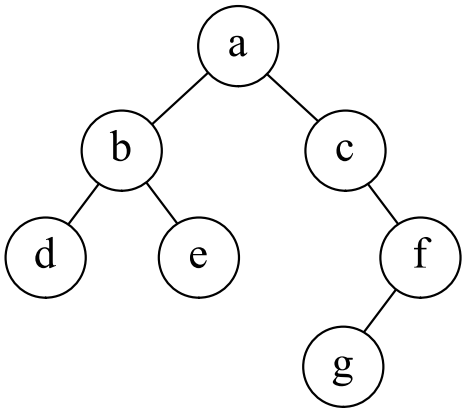
Prof. A. G. Silva

19 de outubro de 2017

Árvores binárias

- Uma árvore binária é vazia, ou é composta por uma raiz e duas subárvores também binárias

- Em Prolog, a árvore vazia é representada pelo átomo "nil" e árvore não vazia pelo termo t(X,L,R), onde X é o nó raiz, e L e R as subárvores esquerda e direita. Exemplo:




```
T1 = t( a,
        t(b,t(d,nil,nil),t(e,nil,nil)),
        t(c,nil,t(f,t(g,nil,nil),nil))
      ).
```

- Outros exemplos:
  - Árvore com somente a raiz: `T2 = t(a,nil,nil).`
  - Árvore vazia: `T3 = nil.`

Árvores de busca binária

- Exemplo:

```
?- construct([3,2,5,7,1],T).
T = t(3, t(2, t(1, nil, nil), nil), t(5, nil, t(7, nil, nil)))
```

- Implementação:

```
add(X,nil,t(X,nil,nil)).
add(X,t(Root,L,R),t(Root,L1,R)) :- X @< Root, add(X,L,L1).
add(X,t(Root,L,R),t(Root,L,R1)) :- X @> Root, add(X,R,R1).

construct(L,T) :- construct(L,T,nil).

construct([],T,T).
construct([N|Ns],T,T0) :- add(N,T0,T1), construct(Ns,T,T1).
```

Árvores binárias quase-completas

- Exemplos (primeiro argumento é o número de nós):

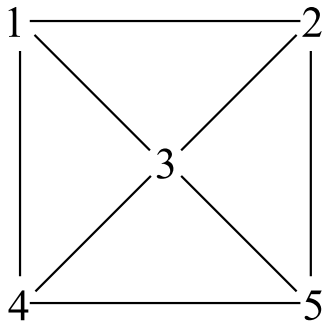
```
?- complete_binary_tree(3, t(3,t(5,nil,nil),t(1,nil,nil))).
true.
?- complete_binary_tree(2, t(3,t(5,nil,nil),nil)).
true.
?- complete_binary_tree(2, t(3,nil,t(1,nil,nil))).
false.
```

- Implementação:

```
complete_binary_tree(N,T) :- complete_binary_tree(N,T,1).

complete_binary_tree(N,nil,A) :- A > N, !.
complete_binary_tree(N,t(_,L,R),A) :- A =< N,
    AL is 2 * A, AR is AL + 1,
    complete_binary_tree(N,L,AL),
    complete_binary_tree(N,R,AR).
```

Grafos

- Arestas representadas por fatos:




```
edge(1,2).
edge(1,4).
edge(1,3).
edge(2,3).
edge(2,5).
edge(3,4).
edge(3,5).
edge(4,5).
```

- Para arestas bidirecionais, pode-se adicionar 8 novas cláusulas (como `edge(2,1)`) ou tentar a regra:

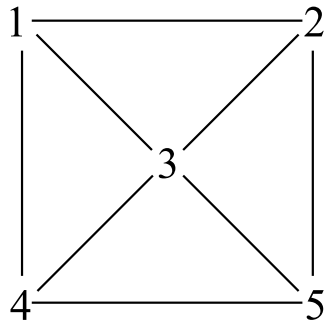
```
edge(X,Y) :- edge(Y,X).
```

Porém não é uma boa ideia (gera *loop* infinito). Solução:

```
connected(X,Y) :- edge(X,Y).
connected(X,Y) :- edge(Y,X).
```

Caminho em grafos (I)

- Queremos uma definição para gerar caminhos entre quaisquer dois nós do grafo. Exemplos:



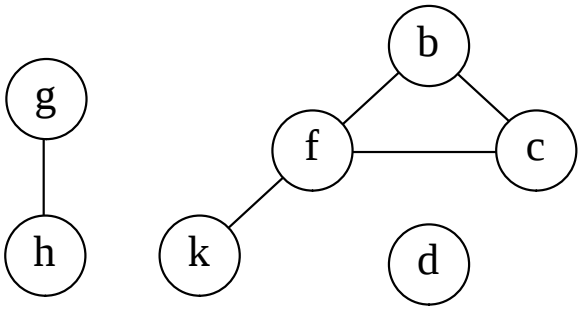

```
?- path(1,5,P).
P = [1,2,5] ;
P = [1,2,3,5] ;
P = [1,2,3,4,5] ;
P = [1,4,5] ;
P = [1,4,3,5] ;
P = [1,4,3,2,5] ;
P = [1,3,5] ;
P = [1,3,4,5] ;
P = [1,3,2,5] ;
false
```

Caminho em grafos (II)

- Implementação:
  - Um caminho de A a B é obtido, se A e B estão conectados
  - Um caminho de A a B é obtido, desde que A esteja conectado a um nó C, diferente de B, que não está conectado ao caminho já visitado, e pode ser encontrado um caminho de C a B

```
path(A,B,Path) :-
       travel(A,B,[A],Q),
       reverse(Q,Path).

travel(A,B,P,[B|P]) :-
       connected(A,B).
travel(A,B,Visited,Path) :-
       connected(A,C),
       C \== B,
       \+member(C,Visited),
       travel(C,B,[C|Visited],Path).
```

Representações de grafos (I)



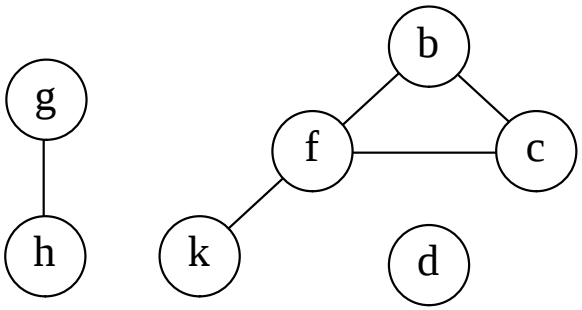
- Cláusulas na forma de arestas (visto inicialmente)

```
edge(h,g).
edge(k,f).
edge(f,b).
...
```

- **Cláusulas na forma de termos de grafo (escolhida)**
  - Representa nós isolados
  - Representação padrão (há predicados pré-definidos para conjuntos)

```
graph([b,c,d,f,g,h,k],[e(b,c),e(b,f),e(c,f),e(f,k),e(g,h)])
```

Representações de grafos (II)



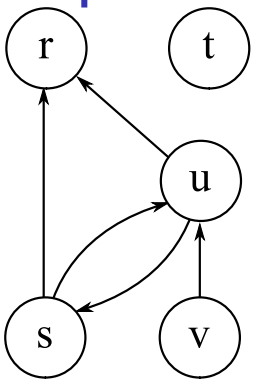
- Cláusulas na forma de lista de adjacências

```
[n(b,[c,f]), n(c,[b,f]), n(d,[]), n(f,[b,c,k]), ...]
```

- Cláusulas na forma amigável (*human-friendly*)
  - Modo para facilitar a digitação à mão
  - Uso de X-Y (funtor "-" com aridade 2)

```
[b-c, f-c, g-h, d, f-b, k-f, h-g]
```

Representações de grafos direcionados



- Cláusulas na forma de arcos

```
arc(s,u).
arc(u,r).
...
```

- **Cláusulas na forma de termos de grafo (escolhida)**

```
digraph([r,s,t,u,v],[a(s,r),a(s,u),a(u,r),a(u,s),a(v,u)])
```

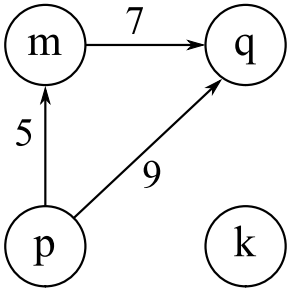
- Cláusulas na forma de lista de adjacências

```
[n(r,[]),n(s,[r,u]),n(t,[]),n(u,[r]),n(v,[u])]
```

- Cláusulas na forma *human-friendly*

```
[s > r, t, u > r, s > u, u > s, v > u]
```

Representações de grafos orientados



- Cláusulas na forma de arcos

```
arc(m,q,7).
arc(p,q,9).
arc(p,m,5).
```

- **Cláusulas na forma de termos de grafo (escolhida)**

```
digraph([k,m,p,q],[a(m,p,7),a(p,m,5),a(p,q,9)])
```

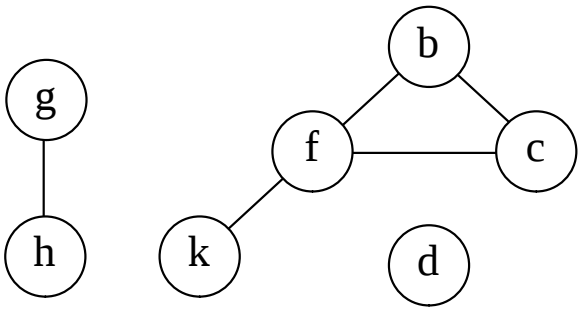
- Cláusulas na forma de lista de adjacências

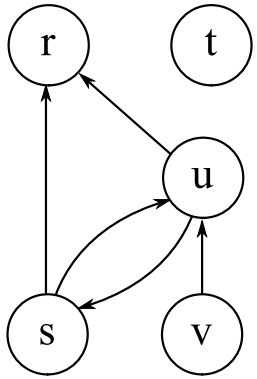
```
[n(k,[]),n(m,[q/7]),n(p,[m/5,q/9]),n(q,[])]
```

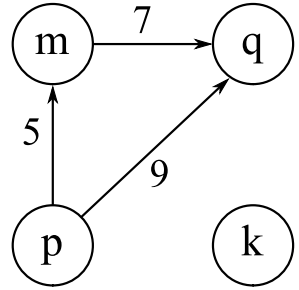
- Cláusulas na forma *human-friendly*

```
[p>q/9, m>q/7, k, p>m/5]
```

Resumo das representações de grafos







- Cláusulas na forma de termos de grafo

```
graph([b,c,d,f,g,h,k],[e(b,c),e(b,f),e(c,f),e(f,k),e(g,h)])
```

- Grafos direcionados

```
digraph([r,s,t,u,v],[a(s,r),a(s,u),a(u,r),a(u,s),a(v,u)])
```

- Grafos valorados (ponderados)

```
digraph([k,m,p,q],[a(m,p,7),a(p,m,5),a(p,q,9)])
```

Caminho de um nó a outro

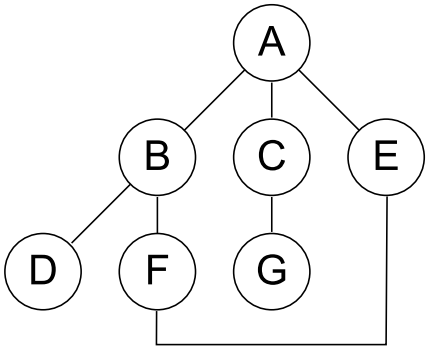
- P é um caminho (sem ciclo) do nó A ao nó B no grafo G

```
path(G,A,B,P) :- path1(G,A,[B],P).

path1(_,A,[A|P1],[A|P1]).
path1(G,A,[Y|P1],P) :-
   adjacent(X,Y,G), \+ memberchk(X,[Y|P1]), path1(G,A,[X,Y|P1],P).

adjacent(X,Y,graph(_,Es)) :- member(e(X,Y),Es).
adjacent(X,Y,graph(_,Es)) :- member(e(Y,X),Es).
adjacent(X,Y,graph(_,Es)) :- member(e(X,Y,_),Es).
adjacent(X,Y,graph(_,Es)) :- member(e(Y,X,_),Es).
adjacent(X,Y,digraph(_,As)) :- member(a(X,Y),As).
adjacent(X,Y,digraph(_,As)) :- member(a(X,Y,_),As).
```

- Exemplo

```
?- path(graph([b,c,d,f,g,h,k],[e(b,c),e(b,f),e(c,f),e(f,k),e(g,h)]), k, c, P).
P = [k, f, b, c] ;
P = [k, f, c] ;
false.
```

Algoritmos de busca



- Busca em profundidade (*depth-first*) – uso de pilha
  - Ordem de visitação (*loop* infinito, não alcança C ou G)

    `A, B, D, F, E, A, B, D, F, E, ...`
  - Ordem de visitação (sem repetição, mantendo nós visitados para grafos "pequenos")

    `A, B, D, F, E, C, G`

- Busca em largura (*breadth-first*) – uso de fila
  - Ordem de visitação

    `A, B, C, E, D, F, G`

Busca em profundidade

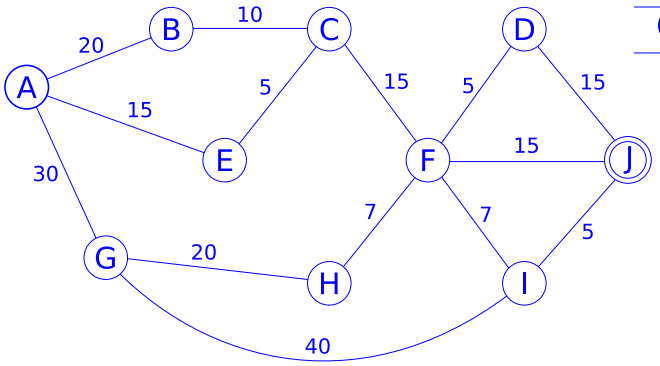
- Nesta solução, é usado "**recorded database**" como alternativa mais eficiente que o mecanismo `assert`/`retract`.

```
depth_first_order(Graph,Start,Seq) :-
   (Graph = graph(Ns,_), !; Graph = digraph(Ns,_)),
   memberchk(Start,Ns),
   clear_rdb(dfo),
   recorda(dfo,Start),
   (dfo(Graph,Start); true),
   bagof(X,recorded(dfo,X),Seq).

dfo(Graph,X) :-
   adjacent(X,Y,Graph),
   \+ recorded(dfo,Y),
   recordz(dfo,Y),
   dfo(Graph,Y).

clear_rdb(Key) :-
   recorded(Key,_,Ref), erase(Ref), fail.
clear_rdb(_).
```

- Exemplo

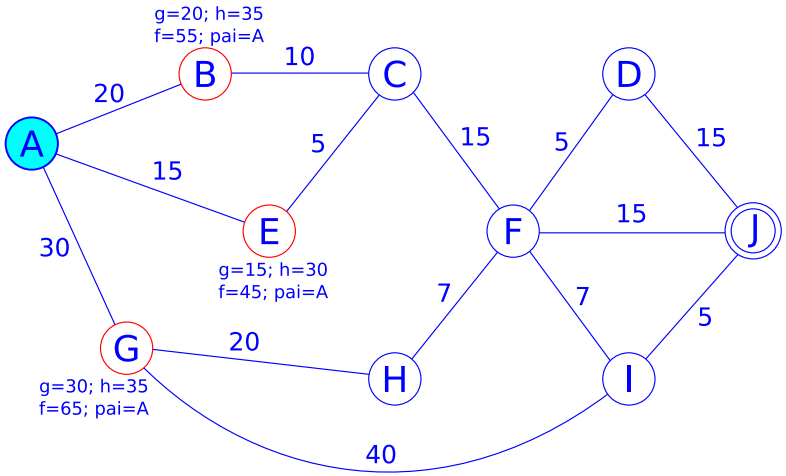
```
?- depth_first_order(graph([a,b,c,d,e,f,g],[e(a,b),e(a,c),e(a,e),e(b,d),e(b,f),e(c,g),e(e,f)]), a, S).
S = [a, b, d, f, e, c, g].
false.
```

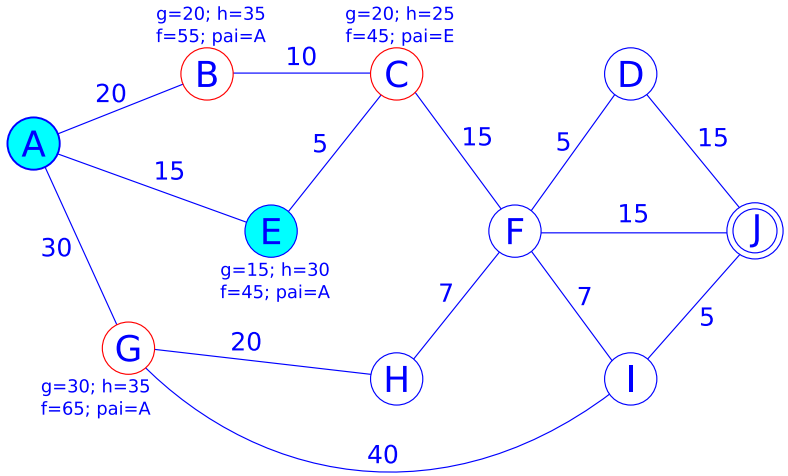
Busca heurística A* (I)



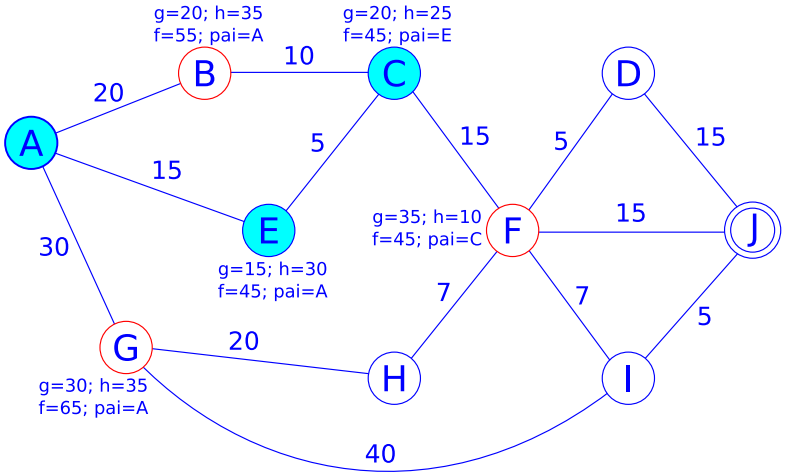
Cidade n	Heurística h(n)	
A	40	← origem
B	35	
C	25	
D	10	
E	30	
F	10	
G	35	
H	15	
I	4	
J	0	← destino

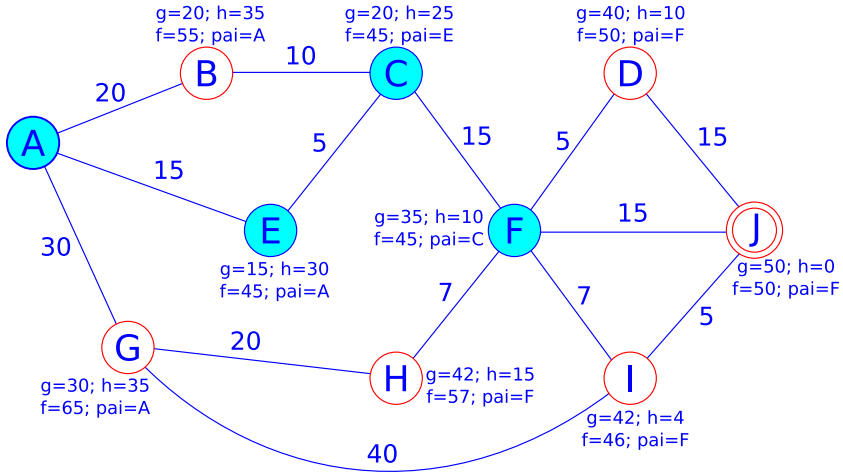
A cidade origem escolhida é A e a destino é J. A heurística adotada é a distância Euclidiana (menor em linha reta) de cada cidade à J (veja tabela ao lado). Segue busca com A*:
A → E (15km), E → C (20km), C → F (35km), F → I (42km), I → J (47km)

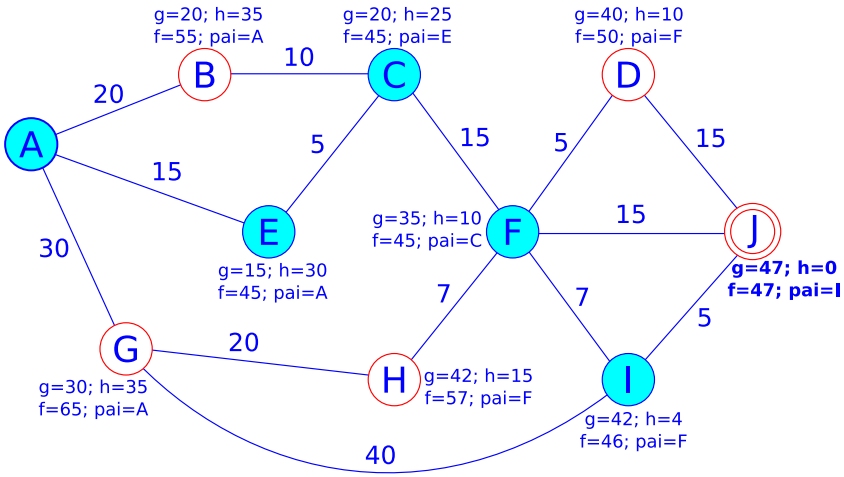


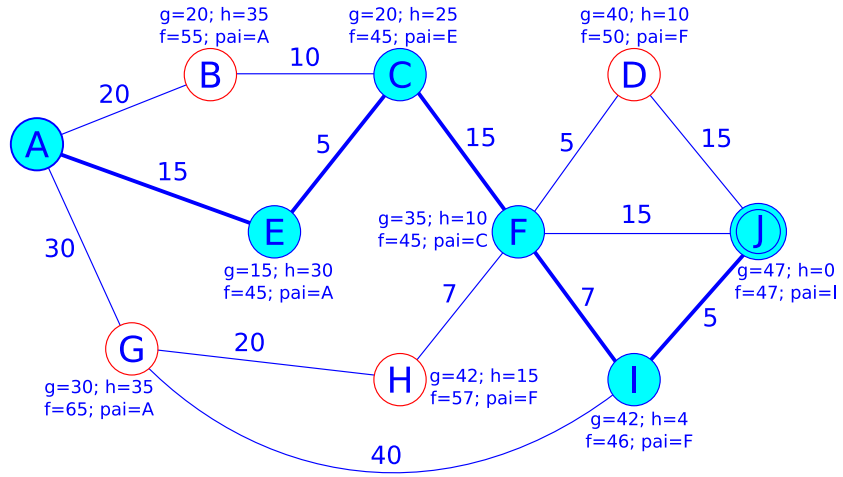


Busca heurística A* (II)









Exercícios (I)

1. Implemente um predicado que devolva o caminho que passa pela menor quantidade de nós.

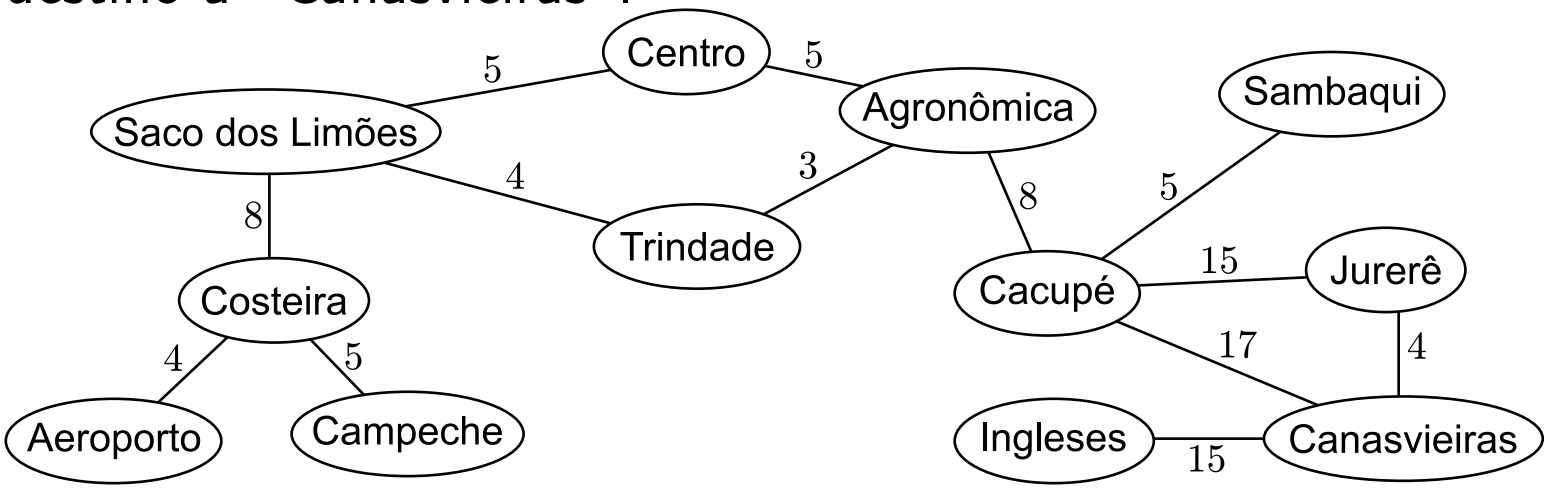
2. Considerando que as arestas têm comprimentos (pesos), determine o caminho de menor distância entre dois pontos (pode ser usada a meta 'setof' para auxiliar nesta tarefa). Adicionar o argumento 'Length' ao 'path' e implementar o predicado 'minimal' a seguir:

```
shortest(Graph,A,B,Path,Length) :-
    setof([P,L],path(Graph,A,B,P,L),Set),
    Set = [_|_],  % falha se nao houver nenhum caminho
    minimal(Set,[Path,Length]).
```

Exercícios (II)

3. Verifique uma implementação de busca em largura. Por exemplo: https://www.cs.unm.edu/~luger/ai-final/code/PROLOG.breadth.html

4. Modele o grafo abaixo (retirado de http://www.inf.ufsc.br/grafos/represen/busca.html) em Prolog e mostre a sequência de bairros visitados pelas implementações dos exercícios 1 e 2, considerando, por exemplo, que partirá de "Saco dos Limões" com destino a "Canasvieiras".



Exercícios (III)

5. Reforçar o entendimento sobre o algoritmo A* e refletir sobre uma implementação em Prolog. Sugestão:

`https://www.inf.ufsc.br/~alexandre.silva/courses/15s1/ine5416/codigos/aestrela/`

   - Busca A* – **astar.pl**
   - Fila de prioridade – **adtPrioQ.pl**
   - Mapa de cidades – **mapa.pl**
   - Mapa ilustrativo – **santacatarina.jpg**
   - Heurística (distância em linha reta) – **heuristicas_cidades.ods:** `https://docs.google.com/spreadsheets/d/1ft2SJzYAMAxa5m_iJCdJ8WQsFwqiqyA3OtQaVuQetVA/edit?usp=sharing`