UCS1524 – Logic Programming

Graph

Session Meta Data

Author	**Dr. D. Thenmozhi**
Reviewer	
Version Number	1.2
Release Date	15 September 2022

Session Objectives

- Understanding graph representation and operations on graph in Prolog.
- Learn about graph representation, finding path algorithm spanning tree algorithm in graph.

Session Outcomes

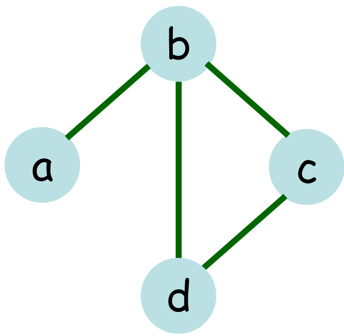
- At the end of this session, participants will be able to
  - explain the graph representation and algorithms for finding path and spanning tree in graph using Prolog.
  - Apply algorithms for real time applications

Agenda

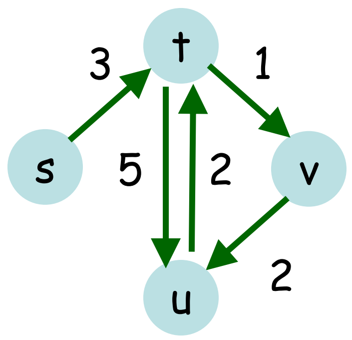
- Representation of graph
- Operations on graph
  - Finding path
  - Finding spanning tree of a graph

Graphs
Representing graphs

- A graph is defined by a set of nodes and a set of edges, where each edge is a pair of nodes.
- When the edges are directed they are also called arcs. Arcs are represented by ordered pairs. Such a graph is a directed graph.



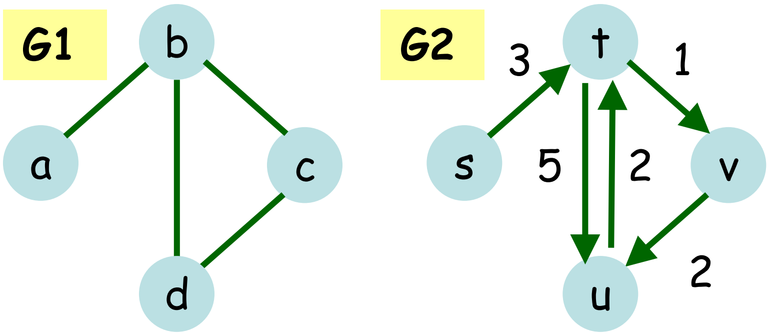
Undirected graph



Directed graph

Representing graphs

- The representation of graphs:
  - Method 1:
    - connected(a, b),
      connected(b, c),…
    - arc(s, t, 3),
      arc(t, v, 1),…



  - Method 2:
    - G1 = graph([a,b,c,d], [e(a,b), e(b,d), e(b,c), e(c,d)])
    - G2 = digraph([s, t, u, v], [a(s,t,3), a(t,v,1), a(t,u,5), a(u,t,2), a(v,u,2)])
  - Method 3:
    - G1 = [a->[b], b->[a,c,d], c->[b,d], d->[b,c]]
    - G2 = [s->[t/3], t->[u/5,v/1], u->[t/2],v->[u/2]]
    - The symbols '->' and '/' are infix operators.

Representing graphs

- What is the most suitable representation?
  - Depending on the application and on operations to be performed on graphs.
- Two typical operations are:
  - Find a path between two given nodes;
  - Find a subgraph, with some specified properties, of a graph.

Finding a path

- Let G be a graph, and A and Z two nodes in G. Let us define the relation:

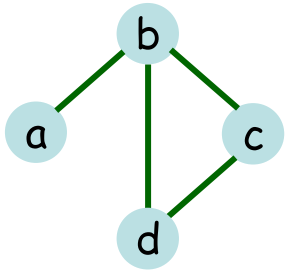
  path(A, Z, G, P)

  where P is an acyclic path between A and Z in G.

  - For example:

    path(a, d, G, [a,b,d])

    path(a, d, G, [a,b,c,d])



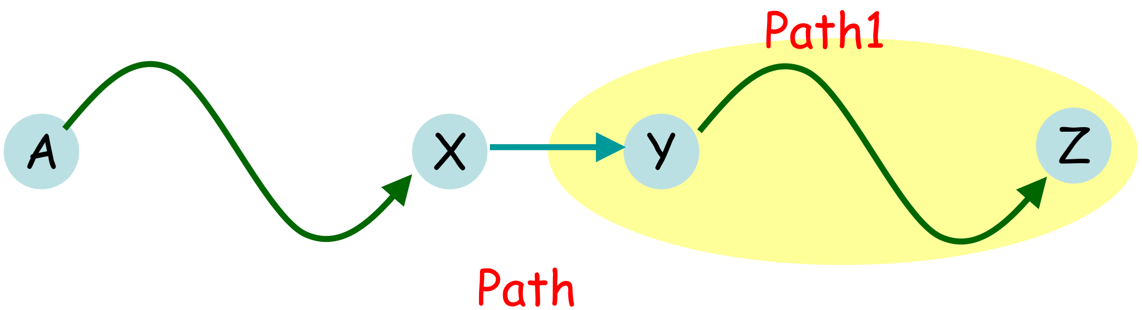
- To find an acyclic path, P, between A and Z in a graph, G:

  If A = Z then P = [A], otherwise

  find an acyclic path, P1, from some node Y to Z,

  and find a path from A to Y avoiding the nodes in P1.

Finding a path



- Define a procedure:

  path1(A, Path1, G, Path)

  – **A** is a node,
  – **G** is a graph,
  – **Path1** is a path in **G**,
  – **Path** is an acyclic path in **G** that goes from **A** to the beginning of **Path1** and continues along **Path1** up to its end.

- The relation between path and path1 is:

  path(A, Z, G, Path) :- path1(A, [Z], G, Path).

Finding a path

% Figure 9.20 Finding an acyclic path, Path, from A to Z in Graph.

```
path( A, Z, Graph, Path) :-
    path1( A, [Z], Graph, Path).

path1( A, [A | Path1], _, [A | Path1] ).

path1( A, [Y | Path1], Graph, Path) :-
    adjacent( X, Y, Graph),
    \+ member( X, Path1),              % not a member
    path1( A, [X, Y | Path1], Graph, Path).
```

- In this program, **member** is the list membership relation.
- The relation **adjacent(X, Y, G)** means that there is an arc from X to Y in graph G. The definition of this relation depends on the representation of graphs.

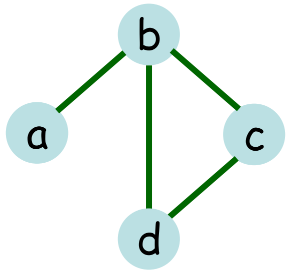
Finding a path

- The **adjacent(X, Y, G)** relation :
  - If G is represented as a pair of sets,

    G = graph(Nodes, Edges)

    then

```
adjacent( X, Y, graph( Nodes, Edges)) :-
    member( e( X, Y), Edges)
    ;
    member( e( Y, X), Edges).
```



  - For example

```
| ?-  G1 = graph([a,b,c,d], [e(a,b), e(b,d), e(b,c), e(c,d)]),
      path(a, d, G1, Path).

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
Path = [a,b,d] ? ;

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
Path = [a,b,c,d] ? ;

no
```

Finding a path

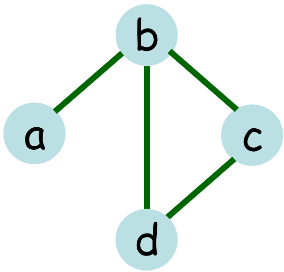
- A classical problem on graphs is to find a Hamiltonian path, that is, an acyclic path comprising all the nodes in the graph.
- Using path this can be done as follows:

```
hamiltonian( Graph, Path) :-
    path(_, _, Graph, Path),
    covers( Path, Graph).

covers( Path, Graph) :-
   \+ ( (node( N, Graph), \+ member( N, Path))).

node( Node, Graph)  :-
    adjacent( Node, _, Graph).
```

  - **node(N, Graph)** means N is a node in Graph.

Finding a path

```
| ?- G1 = graph([a,b,c,d], [e(a,b), e(b,d), e(b,c), e(c,d)]),
     hamiltonian( G1, Path).

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
Path = [_] ? ;

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
Path = [a,b,c,d] ? ;

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
Path = [c,d,b,a] ? ;

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
Path = [d,c,b,a] ? ;

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
Path = [a,b,d,c] ? ;

(15 ms) no
```



Finding a path

- We can attach cost to paths.
  - The cost of a path is the sum of the costs of the arcs in the path.
  - If there are no costs attached to the arcs then we can talk about the length instead, counting 1 for each arc in the path.
  - The **path** and **path1** relation can be modified to handle cost by introducing an additional argument, the cost, for each path:

  path(A, Z, G, P, C)

  path1(A, P1, C1, G, P, C)

  where **C** is the cost of **P** and **C1** is the cost of **P1**.

    - The relation adjacent now also has an extra argument, the cost of an arc.

Finding a path

```
%   Path-finding in a graph:  Path is
    an acyclic path with cost Cost from A to Z in Graph.

path( A, Z, Graph, Path, Cost)  :-
    path1( A, [Z], 0, Graph, Path, Cost).

path1( A, [A | Path1], Cost1, Graph, [A | Path1], Cost1).

path1( A, [Y | Path1], Cost1, Graph, Path, Cost)  :-
     adjacent( X, Y, CostXY, Graph),
     \+ member( X, Path1),
     Cost2 is Cost1 + CostXY,
     path1( A, [X, Y | Path1], Cost2, Graph, Path, Cost).

adjacent( X, Y, Cost, graph( Nodes, Edges)) :-
    member( e( X, Y), Edges), Cost is 1
    ;
    member( e( Y, X), Edges), Cost is 1.
```

Finding a path

```
%   Path-finding in a graph:  Path is
    an acyclic path with cost Cost from A to Z in Graph.

path( A, Z, Graph, Path, Cost)  :-
    path1( A, [Z], 0, Graph, Path, Cost).

path1( A, [A | Path1], Cost1, Graph, [A | Path1], Cost1).

path1( A, [Y | Path1], Cost1, Graph, Path, Cost)  :-
     adjacent( X, Y, CostXY, Graph),
     \+ member( X, Path1),
     Cost2 is Cost1 + CostXY,
     path1( A, [X, Y | Path1], Cost2, Graph, Path, Cost).

adjacent( X, Y, Cost, graph( Nodes, Edges)) :-
    member( e( X, Y, C), Edges), Cost is C
    ;
    member( e( Y, X, C), Edges), Cost is C.
```

Finding a path

| ?- G1 = graph([a,b,c,d], [e(a,b), e(b,d), e(b,c), e(c,d)]), path(a, c, G1, Path, C).

C = 2

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])

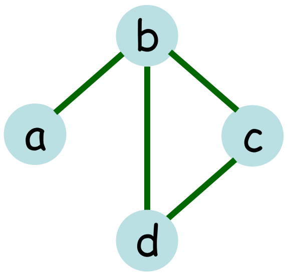
Path = [a,b,c] ? ;

C = 3

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])

Path = [a,b,d,c] ? ;

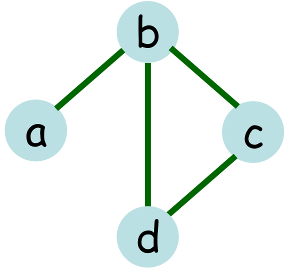
(16 ms) no



Finding a path

- We can fine the minimum cost path:

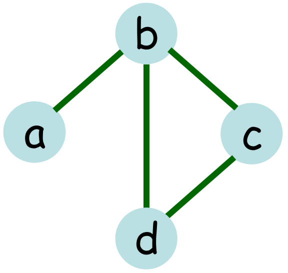
```
| ?- G1 = graph([a,b,c,d], [e(a,b), e(b,d), e(b,c), e(c,d)]), path(a, c, G1,
     MinPath, MinCost),
\+(( path( a,c, G1, _, Cost), Cost < MinCost)).

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
MinCost = 2
MinPath = [a,b,c] ? ;

no
```



Finding a path

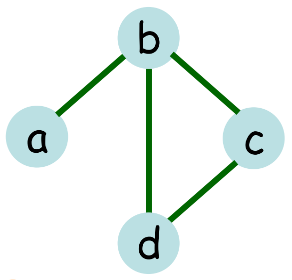
- We can fine the maximum cost path:

```
| ?- G1 = graph([a,b,c,d], [e(a,b), e(b,d), e(b,c), e(c,d)]), path(a, c, G1,
    MaxPath, MaxCost),
\+(( path( a,c, G1, _, Cost), Cost > MaxCost)).

G1 = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
MaxCost = 3
MaxPath = [a,b,d,c] ? ;

no
```



Finding a spanning tree of a graph

- A graph is connected if there is a path from any node to any other node.
- Let G = (V, E) be a connected graph with the set of nodes V and the set of edges E.
- A spanning tree of G is a connected graph T = (V, E’) where E’ is a subset of E such that:
  - T is connected, and
  - There is no cycle in T.
- For example:

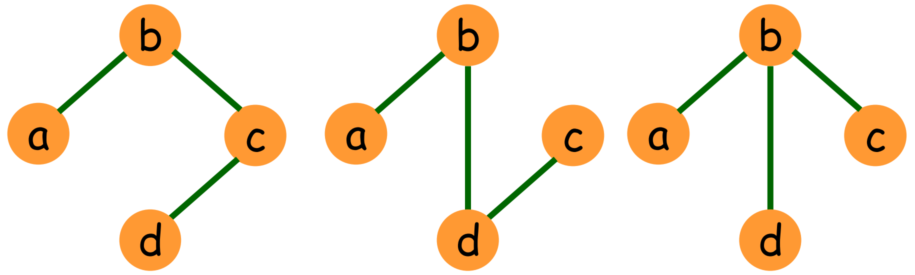


This graph has three spanning trees:

Tree1 = [a-b, b-c, c-d]

Tree2 = [a-b, b-d, d-c]

Tree3 = [a-b, b-d, b-c]



Tree1 Tree2 Tree3

Finding a spanning tree of a graph

- In the edge list of a spanning tree, we can pick any node in such a list as the root of a tree.
- Spanning trees are of interest in communication problems because they provide, with the minimum number of communication lines, a path between any pair of nodes.
- Define a procedure

  stree(G, T)

  where **T** is a spanning tree of **G**.
  - We assume that G is connected.
  - Start with the empty set of edges and gradually add new edges from G, taking care that a cycle is never created, until no more edge can be added because it would create a cycle.
  - The resulting set of edges defines a spanning tree.

Finding a spanning tree of a graph

- The no-cycle condition can be maintained by a simple rule:
  - An edge can be added only if one of its nodes is already in the growing tree, and the other node is not yet in the tree.
- The key relation in the probram Figure 9.22 is:

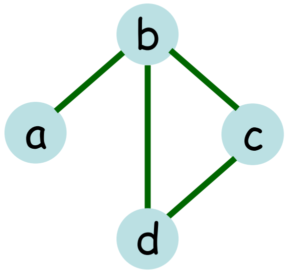
  spread(Tree1, Tree, G)
  - All the three arguments are set of edges.
  - **G** is a connected graph.
  - **Tree1** and **Tree** are subsets of **G** such that they both represent trees.
  - **Tree** is a spanning tree of **G** obtained by adding zero or more edges of **G** to **Tree1**.
  - We can say that ‘Tree1 gets spread to Tree’.

Finding a spanning tree of a graph

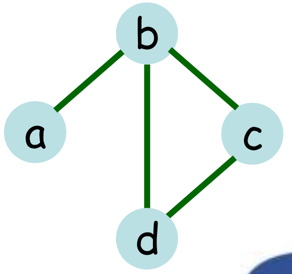
```
% Figure 9.22   Finding a spanning tree of a graph: an `algorithmic'
   program. The program assumes that the graph is connected.

stree( Graph, Tree)  :-
     member( Edge, Graph),  spread( [Edge], Tree, Graph).

spread( Tree1, Tree, Graph)  :-
     addedge( Tree1, Tree2, Graph), spread( Tree2, Tree, Graph).

spread( Tree, Tree, Graph)  :-  \+ addedge( Tree, _, Graph).

addedge( Tree, [A-B | Tree], Graph)  :-
    adjacent( A, B, Graph),
    node( A, Tree), \+ node( B, Tree).

adjacent( Node1, Node2, Graph)  :-
    member( Node1-Node2, Graph)
    ;
    member( Node2-Node1, Graph).

node( Node, Graph)  :-  adjacent( Node, _, Graph).
```

Finding a spanning tree of a graph

```
| ?- stree([a-b, b-c, b-d, c-d], Tree).
Tree = [b-d,b-c,a-b] ? ;
Tree = [b-d,b-c,a-b] ? ;
Tree = [c-d,b-c,a-b] ? ;
Tree = [b-c,b-d,a-b] ? ;
Tree = [b-c,b-d,a-b] ? ;
Tree = [d-c,b-d,a-b] ? ;
Tree = [b-a,b-d,b-c] ? ; …
```




```
| ?- G = [a-b, b-c, b-d, c-d], stree( G, Tree).
G = [a-b,b-c,b-d,c-d]
Tree = [b-d,b-c,a-b] ? ;
G = [a-b,b-c,b-d,c-d]
Tree = [b-d,b-c,a-b] ? ;
G = [a-b,b-c,b-d,c-d]
Tree = [c-d,b-c,a-b] ? ;
G = [a-b,b-c,b-d,c-d]
Tree = [b-c,b-d,a-b] ? ;…
```



Finding a spanning tree of a graph

- We can also develop a working program for constructing a spanning tree in another completely declarative way.

- Assume that both graphs and trees are represented by list of their edges:
  - T is a spanning tree of G if:
    - T is a subset of G, and
    - T is a tree, and
    - T covers G; that is, each node of G is also in T.
  - A set of edges T is a tree if:
    - T is connected, and
    - T has no cycle.

Finding a spanning tree of a graph

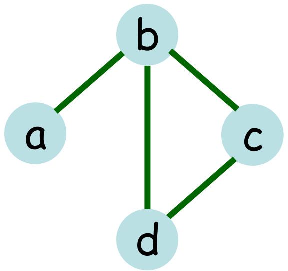
```
% Finding a spanning tree of a graph: a `declarative'
   program. Relations node and adjacent are as in Figure 9.22.

:- op(900, fy, not).
stree1( Graph, Tree)  :-
   subset( Graph, Tree),  tree( Tree),  covers( Tree, Graph).

tree( Tree)  :-
   connected( Tree),  not hasacycle( Tree).

connected( Graph)  :-
   not ( node( A, Graph), node( B, Graph), not path( A, B, Graph, _) ).

hasacycle( Graph)  :-
    adjacent( Node1, Node2, Graph),
    path( Node1, Node2, Graph, [Node1, X, Y | _] ).

covers( Tree, Graph)  :-
   not ( node( Node, Graph), not node( Node, Tree) ).

subset( [], [] ).
subset( [X | Set], Subset)  :-  subset( Set, Subset).
subset( [X | Set], [X | Subset])  :-  subset( Set, Subset).
```

Finding a spanning tree of a graph

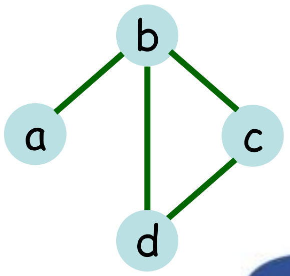
```
% (con.)  Finding a spanning tree of a graph: a `declarative' program. Relations node
       and adjacent are as in Figure 9.22.

adjacent( Node1, Node2, Graph)  :-
      member( Node1-Node2, Graph)
      ;
     member( Node2-Node1, Graph).

node( Node, Graph)  :-
      adjacent( Node, _, Graph).

path( A, Z, Graph, Path)  :-
      path1( A, [Z], Graph, Path).

path1( A, [A | Path1], _, [A | Path1] ).

path1( A, [Y | Path1], Graph, Path)  :-
      adjacent( X, Y, Graph),
      not member( X, Path1),
      path1( A, [X, Y | Path1], Graph, Path).
```

Finding a spanning tree of a graph

```
| ?- stree1([a-b, b-c, b-d, c-d], Tree).

Tree = [a-b,b-d,c-d] ? ;
Tree = [a-b,b-c,c-d] ? ;
Tree = [a-b,b-c,b-d] ? ;
(31 ms) no
```




```
| ?- G1 = [a-b, b-c, b-d, c-d], stree1(G1, Tree).

G1 = [a-b,b-c,b-d,c-d]
Tree = [a-b,b-d,c-d] ? ;
G1 = [a-b,b-c,b-d,c-d]
Tree = [a-b,b-c,c-d] ? ;
G1 = [a-b,b-c,b-d,c-d]
Tree = [a-b,b-c,b-d] ? ;
no
```



Finding a spanning tree of a graph

```
| ?- covers([a-b, b-c], [a-b, b-c, a-c]).
yes

| ?- covers([a-b, b-c], [a-b, a-c]).
yes

| ?- covers([a-b, b-c], [a-b, b-c]).
yes

| ?- covers([a-b, b-c], [a-b, b-c, c-d]).
no

| ?- covers([a-b], [a-b, b-c]).
no
```

```
| ?- subset([a-b, b-c, a-c], [a-b, b-c]).
true ?
yes

| ?- subset([a-b, b-c, a-c], [a-b, a-c]).
true ?
yes

| ?- subset([a-b, b-c, a-c], [a-c, a-c]).
no

| ?- subset([a-b, b-c, a-c], [a-c, a-b]).
no
```

Summary

- Representation of graph
  - 3 ways
  - Using node list and edge list is a common way
- Finding path
  - Path between 2 nodes
  - Hamiltonian path
  - Finding cost of a path
  - Path with minimum cost and maximum cost
- Finding spanning tree of a graph