Discrete Math Tools for AI

Last modified on 2022/03/02 by f.maire@qut.edu.au

Key discrete math concepts for AI

- Recurrence relations and recursive functions
- Graphs and trees
  - data structure, abstraction, object oriented programming
- Graph properties
  - directedness, paths, connectivity, connected components, cycles, roots, sinks
- Dijsktra algorithm
  - computation of the shortest paths from a source to all other vertices

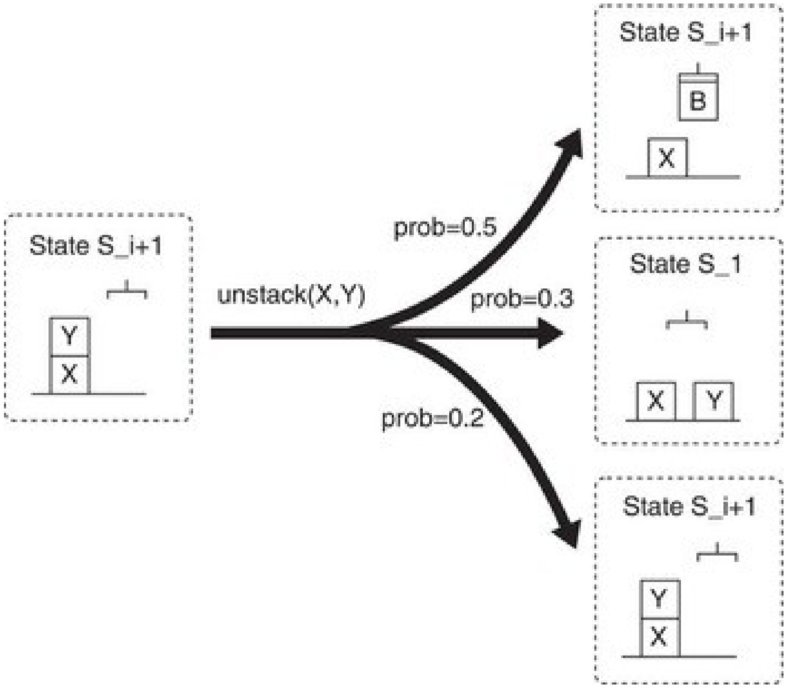
Graphs

- Formally, a graph *G* is a pair *(V, E)* where *V* is a set of **vertices** and *E* is a set of pairs of vertices called **edges.**
- *V* can be any set, *E* can be **any binary relation**.
- A graph does not need a visual representation to exist!
- If the edges are ordered, then they are usually called **arcs**. The graph is then called a **directed graph** (often shortened to **digraph**)

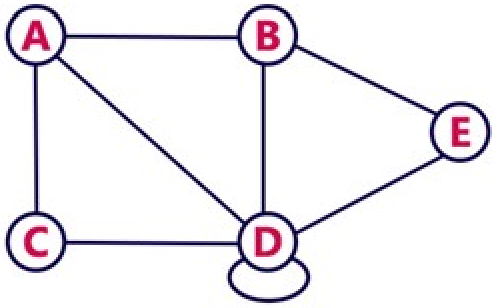
Vertices are also called **nodes**

Why graphs play a critical role in AI?

- All planning problems, finding a “good” sequence of actions, can be reduced to the problem of find a “good” path in an associated state graph.
- In knowledge representation, graphs are often the natural data structure (social network, scene representation, protein folding, etc)



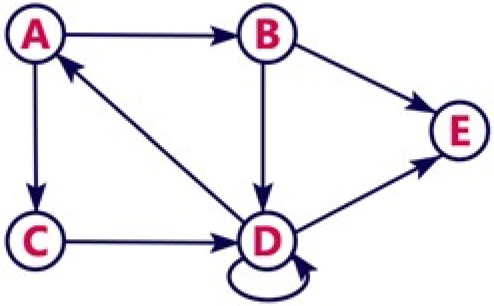
Undirected graph representation



$$
\begin{array}{c c}
 & \begin{array}{ccccc} A & B & C & D & E \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} & \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}
\end{array}
$$

The vertex-vextex **incidence matrix** is called the **adjacency matrix**

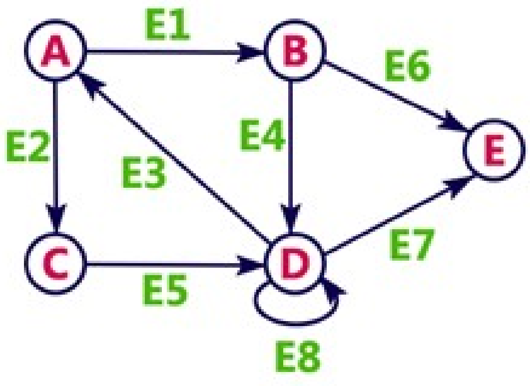
Directed graph representation



$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

The “1” at the intersection of row “B” and column “D” corresponds to the arc BD

Directed graph representation



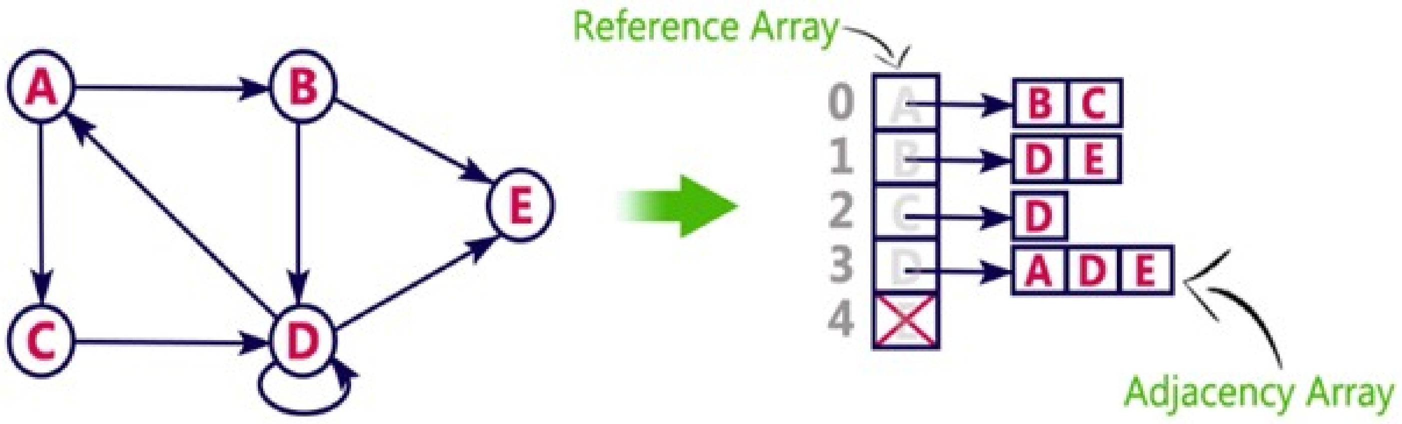
$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{cccccccc} E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \end{array} \\ \begin{pmatrix} 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{pmatrix} \end{array}$$

Vertex-arc
incidence matrix

The **in-degree** is the number of incoming arcs of a vertex.
The **out-degree** is the number of outgoing arcs to a vertex

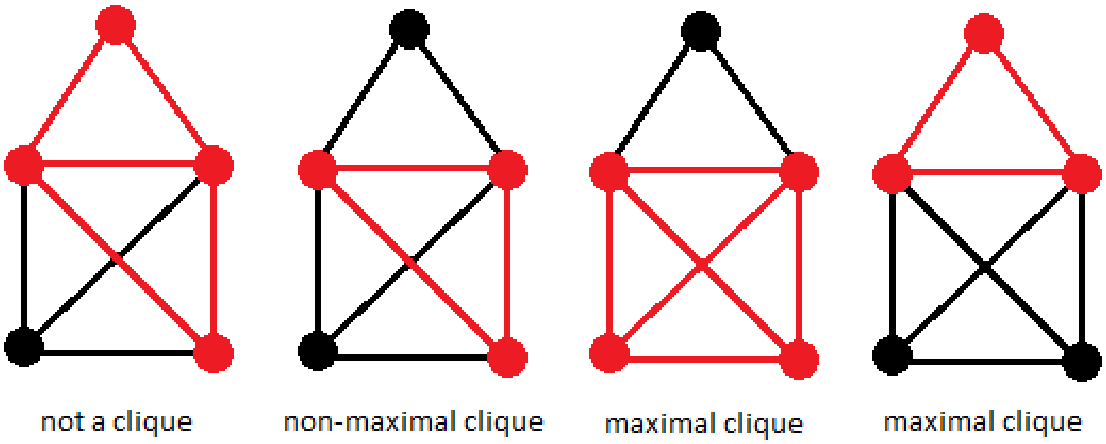
The in-degree of “B” is 1.
The out-degree of “B” is 2.

Adjacency list



In Python, we can use **dictionaries** to represent the adjacency relation. The keys are the vertices, the values are lists of neighbours.

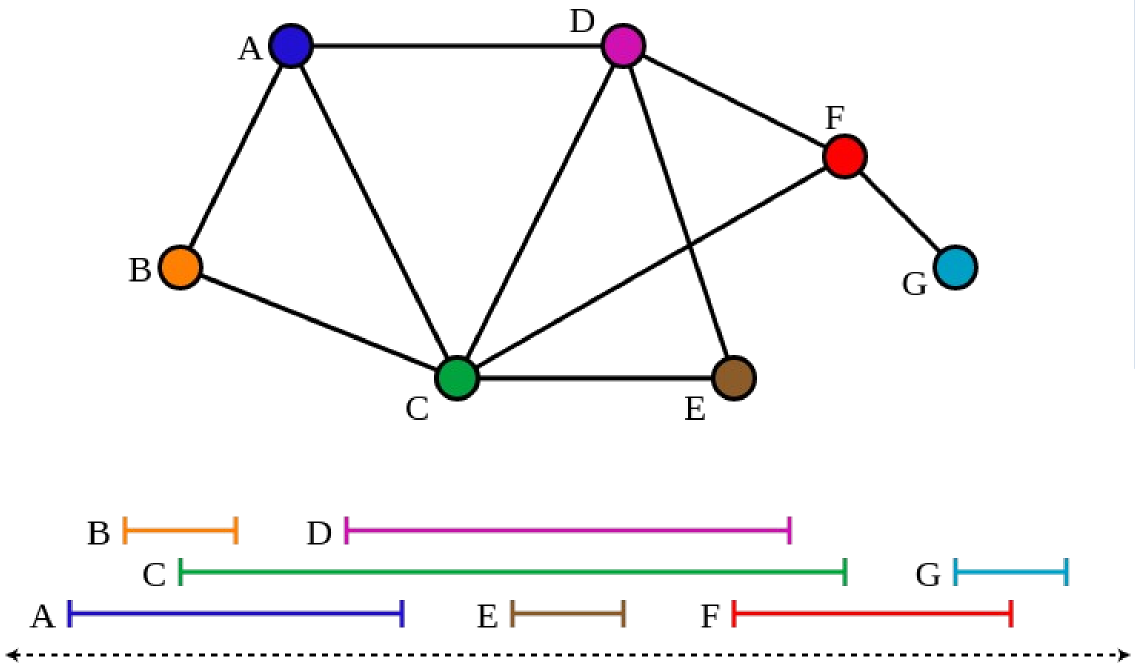
Clique graphs



A **clique** is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent.

A **maximal clique** is a clique that cannot be extended by including one more adjacent vertex.

Interval graphs



The **intersection graph** of a set of intervals has the Intervals as its vertices. There is an edge between two vertices if the corresponding intervals intersect.

Can you show that in a interval graph there is always a vertex whose neighbours form a clique?

Connectivity

- In an undirected graph G, two vertices u and v are called **connected** if G contains a **path** from u to v. Otherwise, they are called **disconnected**. If the two vertices are additionally connected by a path of length 1, i.e. by a single edge, the vertices are called **adjacent**.
- A graph is said to be **connected** if every pair of vertices in the graph is connected. This means that there is a path between every pair of vertices.
- An undirected graph that is not connected is called **disconnected**.

A **path** is a sequence of vertices {v1, v2, …, vn} such that two successive vertices in the sequence are connected by an edge. If v1=vn, the path is called a **cycle**

Connectivity

- A **connected component** is a maximal connected **subgraph** of an undirected graph. Each vertex belongs to exactly one connected component, as does each edge. A graph is connected if and only if it has exactly one connected component.
- A **vertex cut** or **separating set** of a connected graph G is a set of vertices whose removal renders G disconnected.

Trees

- A **tree** is an undirected graph G that satisfies any of the following equivalent conditions:
  - G is connected and acyclic (contains no cycles).
  - G is acyclic, and a simple cycle is formed if any edge is added to G.
  - G is connected, but would become disconnected if any single edge is removed from G.
  - Any two vertices in G can be connected by a unique simple path.

Dijkstra Algorithm

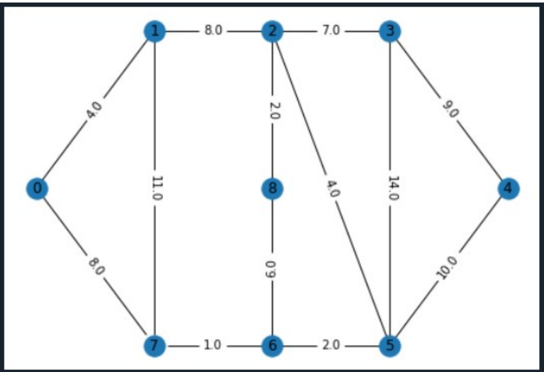
In this context, a source is simply a distinguished vertex

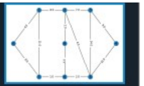
- **Aim**: Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph

- **Main idea**: We generate a ***shortest path tree*** with a given source as its **root**. We maintain two sets, one set contains vertices included in the growing shortest-path tree, the other set L is its complement. That is the set of vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex of L that has a minimum distance from the source.

Dijkstra code


```python
def Dijkstra_shortest_path(G, start):
    '''
    Implementation of Dijkstra algorithm
    https://en.wikipedia.org/wiki/Dijkstra
    Compute the shortest paths from node start to all the other nodes in G.
    Return D, P   where D[v]  is the cost of the cheapest path from start to
                  node v,  and P[v] is the parent node of v on this optimal
                  path from start to v.
    '''
    def get_closest():
        '''
        Find in L, the element u with the smallest dist[u]
        Remove u from L and return u, dist[u]
        '''
        u = L[0]
        du = dist[u]
        for v in L[1:]:
            if dist[v]<du:
                u,du = v, dist[v]
        L.remove(u)
        return u, du
    # ......................................................................

    L = list(G.nodes) # List of nodes that have not been finalized

    dist = {v:inf for v in G.nodes} # mapping v -> cost of best path
                                    # known so far from node start to v
    parent = {v:None for v in G.nodes} # mapping v -> parent in the tree of
                                       # shortest paths
    dist[start] = 0

    while L: # while there are unfinalized nodes
        u, du = get_closest() #
        for v in G.neighbors(u):
            if du+G.adj[u][v]['weight'] < dist[v]:
                dist[v] = du+G.adj[u][v]['weight']
                parent[v] = u

    return dist, parent
    # ----------------------------------------------------------------------

G = make_graph_expl_1()
```





Variable explorer | Help | Plots | Files | Find

Console 3/A

```
Figures now render in the Plots pane by default. To make them also appear
inline in the Console, uncheck "Mute Inline Plotting" under the Plots pane
options menu.

In [2]: D,P
Out[2]:
({0: 0, 1: 4.0, 2: 12.0, 3: 19.0, 4: 21.0, 5: 11.0, 6: 9.0, 7: 8.0, 8: 14.0},
 {0: None, 1: 0, 2: 1, 3: 2, 4: 5, 5: 6, 6: 7, 7: 0, 8: 2})

In [3]: D
Out[3]: {0: 0, 1: 4.0, 2: 12.0, 3: 19.0, 4: 21.0, 5: 11.0, 6: 9.0, 7: 8.0, 8:
14.0}

In [4]: P
Out[4]: {0: None, 1: 0, 2: 1, 3: 2, 4: 5, 5: 6, 6: 7, 7: 0, 8: 2}

In [5]:
```

IPython console | History

LSP Python: ready   conda: base (Python 3.8.3)   Line 89, Col 53   ASCII   LF   RW   Mem 32%

Correctness of Dijkstra Algorithm

Induction on the size of **V-L** where **L** is the set of unfinalized Vertices.

A non-existing edge is equivalent to an edge of infinite weight

Invariant hypothesis:

- For each vertex v, dist[v] is an upper bound of the cost of a cheapest path from vertex *start* to v. The associated path can be reconstructed by following the ancestors of v with the parent relation.
- If v has been removed from *L*, then dist[v] is the cost of a cheapest path from node *start* to v. Moreover this path visits only vertices no longer in *L*. The cost dist[v] is infinity if no such path exists.

Base case: is when there is just one visited node, namely the initial node *start*, in which case the hypothesis is trivial as dist[start] is set to 0.

When v has been removed from *L*, we say that v has been **finalized**

Correctness of Dijkstra Algorithm

General case:

- Assume the invariant is true for the vertices that have been removed from L.
- We then pick the vertex u that has the smallest dist[u] of any node in L.
- *We have to show that the invariant holds after removing u from L.*

Correctness of Dijkstra Algorithm

Lemma

- If T is the tree of finalized vertices, then for any non finalized vertex a adjacent to T, the path a → parent(a) → source is the cheapest among the paths going from a to source using only vertices from T
- *Proof:* When a vertex v is added to T, the cost of going from a to the source via the current parent of a is compared to the cost of the path of going from a to the source via v. After this comparison, the new parent of a is on the cheapest path from a to the source using only vertices from T

Correctness of Dijkstra Algorithm

- By construction dist[u] is the smallest among the vertices of L.
- Assume there is a strictly cheaper path P from start to u than the one obtained by following the parent relation.
- **Case 1**: P uses no vertices of L.
  - According to the lemma, the path via parent[u] is the cheapest because we use only finalized vertices. A contradiction with the assumption that path P is strictly cheaper.

Correctness of Dijkstra Algorithm

- **Case 2**: P uses some vertices belonging to L.
  - Let call w the vertex parent[u].
  - Let call y the first vertex of P not in L.
  - Let call x the vertex parent[y].
  - We have dist[u] > cost(P) >= dist[y]
  - This implies that dist[u] > dist[y].
  - We should have remove y from L instead of u!
  - A contradiction.

Networkx: a Python module for graphs

- NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks
- https://networkx.org/documentation/stable/index.html
- Our implementation of Dijsktra algorithm uses the Graph class of the Networkx library

Lecture Review

- What data structure is suitable to represent a graph?
- What data structure would you use if the graph is sparse (very few edges)?
- What is the time complexity of Dijkstra Algorithm with respect to the number of vertices?
- Find out what is the *complement graph*? How are the adjacency matrices of a graph and its complement related?
- Using the code provided on Blackboard, step through Dijkstra algorithm, reconstruct the associated tree. Change some weights so that the algorithm returns a different tree.