# 5. Graph Theory and Decision Tree

## 5.1 Graph

In computer science, a graph is an abstract data type that is meant to implement the undirected graph and directed graph concepts from the field of graph theory within mathematics.

A graph data structure consists of a finite (and possibly mutable) set of vertices (also called nodes or points), together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as edges (also called links or lines), and for a directed graph are also known as edges but also sometimes arrows or arcs. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

In one restricted but very common sense of the term, a graph is an ordered pair $G = (V, E)$ comprising: $V$, a set of vertices (also called nodes or points); $E \subseteq \{\{x, y\} \mid x, y \in V \text{ and } x \neq y\}$, a set of edges (also called links or lines), which are unordered pairs of vertices (that is, an edge is associated with two distinct vertices). To avoid ambiguity, this type of object may be called precisely an undirected simple graph.

In the edge $\{x, y\}$, the vertices $x$ and $y$ are called the endpoints of the edge. The edge is said to join $x$ and $y$ and to be incident on $x$ and on $y$. A vertex may exist in a graph and not belong to an edge. Multiple edges, not allowed under the definition above, are two or more edges that join the same two vertices.

In one more general sense of the term allowing multiple edges, a graph is an ordered triple

$G = (V, E, \phi)$ comprising: $V$, a set of vertices (also called nodes or points); $E$, a set of edges (also called links or lines); $\phi : E \to \{\{x,y\} \mid x,y \in V \text{ and } x \neq y\}$, an incidence function mapping every edge to an unordered pair of vertices (that is, an edge is associated with two distinct vertices). To avoid ambiguity, this type of object may be called precisely an undirected multigraph.

A loop is an edge that joins a vertex to itself. Graphs as defined in the two definitions above cannot have loops, because a loop joining a vertex $x$ to itself is the edge (for an undirected simple graph) or is incident on (for an undirected multigraph) $\{x,x\} = \{x\}$ which is not in $\{\{x,y\} \mid x,y \in V \text{ and } x \neq y\}$. So to allow loops the definitions must be expanded. For undirected simple graphs, the definition of $E$ should be modified to $E \subseteq \{\{x,y\} \mid x,y \in V\}$. For undirected multigraphs, the definition of $\phi$ should be modified to $\phi : E \to \{\{x,y\} \mid x,y \in V\}$. To avoid ambiguity, these types of objects may be called undirected simple graph permitting loops and undirected multigraph permitting loops (sometimes also undirected pseudograph), respectively.

$V$ and $E$ are usually taken to be finite, and many of the well-known results are not true (or are rather different) for infinite graphs because many of the arguments fail in the infinite case. Moreover, $V$ is often assumed to be non-empty, but $E$ is allowed to be the empty set. The order of a graph is $|V|$, its number of vertices. The size of a graph is $|E|$, its number of edges. The degree or valency of a vertex is the number of edges that are incident to it, where a loop is counted twice. The degree of a graph is the maximum of the degrees of its vertices.

In an undirected simple graph of order n, the maximum degree of each vertex is $n-1$ and the maximum size of the graph is $n(n-1)/2$.

The edges of an undirected simple graph permitting loops $G$ induce a symmetric homogeneous relation  on the vertices of $G$ that is called the adjacency relation of $G$. Specifically, for each edge $(x,y)$, its endpoints $x$ and $y$ are said to be adjacent to one another, which is denoted $x$   $y$.

阅读后结合上课内容掌握图论的基础概念，能够利用图结构进行相关数据的形式化建模。

## 5.2 Directed Graph

A directed graph or digraph is a graph in which edges have orientations.

In one restricted but very common sense of the term, a directed graph is an ordered pair $G = (V, E)$ comprising: $V$, a set of vertices (also called nodes or points); $E \subseteq \left\{(x,y) \mid (x,y) \in V^2 \text{ and } x \neq y\right\}$, a set of edges (also called directed edges, directed links, directed lines, arrows or arcs) which are ordered pairs of vertices (that is, an edge is associated with two distinct vertices). To avoid ambiguity, this type of object may be called precisely a directed simple graph.

In the edge $(x,y)$ directed from $x$ to $y$, the vertices $x$ and $y$ are called the endpoints of the edge, $x$ the tail of the edge and $y$ the head of the edge. The edge is said to join $x$ and $y$ and to be incident on $x$ and on $y$. A vertex may exist in a graph and not belong to an edge. The edge $(y,x)$ is called the inverted edge of $(x,y)$. Multiple edges, not allowed under the definition above, are two or more edges with both the same tail and the same head.

In one more general sense of the term allowing multiple edges, a directed graph is an ordered triple $G = (V,E,\phi)$ comprising:

$V$, a set of vertices (also called nodes or points); $E$, a set of edges (also called directed edges, directed links, directed lines, arrows or arcs); $\phi : E \to \{(x,y) \mid (x,y) \in V^2 \text{ and } x \neq y\}$, an incidence function mapping every edge to an ordered pair of vertices (that is, an edge is associated with two distinct vertices). To avoid ambiguity, this type of object may be called precisely a directed multigraph.

A loop is an edge that joins a vertex to itself. Directed graphs as defined in the two definitions above cannot have loops, because a loop joining a vertex $x$ to itself is the edge (for a directed simple graph) or is incident on (for a directed multigraph) $(x,x)$ which is not in $\{(x,y) \mid (x,y) \in V^2 \text{ and } x \neq y\}$. So to allow loops the definitions must be expanded. For directed simple graphs, the definition of $E$ should be modified to $E \subseteq \{(x,y) \mid (x,y) \in V^2\}$. For directed multigraphs, the definition of $\phi$ should be modified to $\phi : E \to \{(x,y) \mid (x,y) \in V^2\}$. To avoid ambiguity, these types of objects may be called precisely a directed simple graph permitting loops and a directed multigraph permitting loops (or a quiver) respectively.

The edges of a directed simple graph permitting loops $G$ is a homogeneous relation   on the vertices of $G$ that is called the adjacency relation of $G$. Specifically, for each edge $(x,y)$, its endpoints $x$ and $y$ are said to be adjacent to one another, which is denoted $x \quad y$.

阅读后结合上课内容掌握有向图论的基础概念。

## 5.3 Bipartite Graph

In the mathematical field of graph theory, a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint and independent sets $U$ and $V$, that is every edge connects a vertex in $U$ to one in $V$. Vertex sets $U$ and $V$ are usually called the parts of the graph. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles.

The two sets $U$ and $V$ may be thought of as a coloring of the graph with two colors: if one colors all nodes in $U$ blue, and all nodes in $V$ green, each edge has endpoints of differing colors, as is required in the graph coloring problem. In contrast, such a coloring is impossible in the case of a non-bipartite graph, such as a triangle: after one node is colored blue and another green, the third vertex of the triangle is connected to vertices of both colors, preventing it from being assigned either color.

One often writes $G = (U,V,E)$ to denote a bipartite graph whose partition has the parts $U$

and $V$, with $E$ denoting the edges of the graph. If a bipartite graph is not connected, it may have more than one bipartition; in this case, the $(U,V,E)$ notation is helpful in specifying one particular bipartition that may be of importance in an application. If $|U| = |V|$, that is, if the two subsets have equal cardinality, then $G$ is called a balanced bipartite graph. If all vertices on the same side of the bipartition have the same degree, then $G$ is called biregular.

阅读后结合上课内容掌握二分图的定义，理解其与一般神经网络层与层结构之间的关系

## 5.4 Incidence Matrix

In mathematics, an incidence matrix is a logical matrix that shows the relationship between two classes of objects, usually called an incidence relation. If the first class is $X$ and the second is $Y$, the matrix has one row for each element of $X$ and one column for each element of $Y$. The entry in row $x$ and column $y$ is 1 if $x$ and $y$ are related (called incident in this context) and 0 if they are not.

In graph theory an undirected graph has two kinds of incidence matrices: unoriented and oriented.

The unoriented incidence matrix (or simply incidence matrix) of an undirected graph is a $n \times m$ matrix $B$, where $n$ and $m$ are the numbers of vertices and edges respectively, such that $B_{ij} = 1$ if the vertex $v_i$ and edge $e_j$ are incident and 0 otherwise.

For example, the incidence matrix of the undirected graph shown on the right is a matrix consisting of 4 rows (corresponding to the four vertices, 1–4) and 4 columns (corresponding to the four edges, $e_1, e_2, e_3, e_4$:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

If we look at the incidence matrix, we see that the sum of each column is equal to 2. This is because each edge has a vertex connected to each end.

The incidence matrix of a directed graph is a $n \times m$ matrix $B$ where $n$ and $m$ are the number of vertices and edges respectively, such that $B_{ij} = -1$ if the edge $e_j$ leaves vertex $v_i$, 1 if it enters vertex $v_i$ and 0 otherwise (many authors use the opposite sign convention).

The oriented incidence matrix of an undirected graph is the incidence matrix, in the sense of directed graphs, of any orientation of the graph. That is, in the column of edge e, there is one 1 in the row corresponding to one vertex of e and one -1 in the row corresponding to the other vertex of e, and all other rows have 0. The oriented incidence matrix is unique up to negation of any of the columns, since negating the entries of a column corresponds to reversing the orientation of an edge.

The unoriented incidence matrix of a graph G is related to the adjacency matrix of its line graph $L(G)$ by the following theorem:

$$A(L(G)) = B(G)^{\mathsf{T}} B(G) - 2I_m.$$

where $A(L(G))$ is the adjacency matrix of the line graph of $G$, $B(G)$ is the incidence matrix, and $I_m$ is the identity matrix of dimension $m$.

The discrete Laplacian (or Kirchhoff matrix) is obtained from the oriented incidence matrix B(G) by the formula

$$B(G)B(G)^{\mathsf{T}}.$$

The integral cycle space of a graph is equal to the null space of its oriented incidence matrix, viewed as a matrix over the integers or real or complex numbers. The binary cycle space is the null space of its oriented or unoriented incidence matrix, viewed as a matrix over the two-element field.

阅读后结合上课内容掌握关联矩阵的计算方式，能够根据给定的图结构计算其关联矩阵。

## 5.5 Adjacency Matrix

In graph theory and computer science, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected (i.e. all of its edges are bidirectional), the adjacency matrix is symmetric. The relationship between a graph and the eigenvalues and eigenvectors of its adjacency matrix is studied in spectral graph theory.

The adjacency matrix of a graph should be distinguished from its incidence matrix, a different matrix representation whose elements indicate whether vertex–edge pairs are incident or not, and its degree matrix, which contains information about the degree of each vertex.

For a simple graph with vertex set $U = \{u_1, \dots, u_n\}$, the adjacency matrix is a square $n \times n$ matrix $A$ such that its element $A_{ij}$ is one when there is an edge from vertex $u_i$ to vertex $u_j$, and zero when there is no edge. The diagonal elements of the matrix are all zero, since edges from a vertex to itself (loops) are not allowed in simple graphs. It is also sometimes useful in algebraic graph theory to replace the nonzero elements with algebraic variables. The same concept can be extended to multigraphs and graphs with loops by storing the number of edges between each two vertices in the corresponding matrix element, and by allowing nonzero diagonal elements. Loops may be counted either once (as a single edge) or twice (as two vertex-edge incidences), as long as a consistent convention is followed. Undirected graphs often use the latter convention of counting loops twice, whereas directed graphs typically use the former convention.

阅读后结合上课内容掌握邻接矩阵的计算方式，能够根据给定的图结构计算其邻接矩阵，理解其余关联矩阵的差异。

## 5.6 Eulerian Path

In graph theory, an Eulerian trail (or Eulerian path) is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices). Similarly, an Eulerian circuit or Eulerian cycle is an Eulerian trail that starts and ends on the same vertex. They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Königsberg problem in 1736. The problem can be stated mathematically like this:

**Given the graph in the image, is it possible to construct a path (or a cycle; i.e., a path starting and ending on the same vertex) that visits each edge exactly once?**

Euler proved that a necessary condition for the existence of Eulerian circuits is that all vertices in the graph have an even degree, and stated without proof that connected graphs with all vertices of even degree have an Eulerian circuit. The first complete proof of this latter claim was published posthumously in 1873 by Carl Hierholzer. This is known as Euler's Theorem:

**A connected graph has an Euler cycle if and only if every vertex has even degree.**

The term Eulerian graph has two common meanings in graph theory. One meaning is a graph with an Eulerian circuit, and the other is a graph with every vertex of even degree. These definitions coincide for connected graphs.

For the existence of Eulerian trails it is necessary that zero or two vertices have an odd degree; this means the Königsberg graph is not Eulerian. If there are no vertices of odd degree, all Eulerian trails are circuits. If there are exactly two vertices of odd degree, all Eulerian trails start at one of them and end at the other. A graph that has an Eulerian trail but not an Eulerian circuit is called semi-Eulerian.

阅读后结合上课内容掌握欧拉回路的定义。

## 5.7 Hamiltonian Path

In the mathematical field of graph theory, a Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian path that is a cycle. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.

Hamiltonian paths and cycles are named after William Rowan Hamilton who invented the icosian game, now also known as Hamilton's puzzle, which involves finding a Hamiltonian cycle in the edge graph of the dodecahedron. Hamilton solved this problem using the icosian calculus, an algebraic structure based on roots of unity with many similarities to the quaternions (also invented by Hamilton). This solution does not generalize to arbitrary graphs.

Despite being named after Hamilton, Hamiltonian cycles in polyhedra had also been studied a year earlier by Thomas Kirkman, who, in particular, gave an example of a polyhedron without Hamiltonian cycles. Even earlier, Hamiltonian cycles and paths in the knight's graph of the chessboard, the knight's tour, had been studied in the 9th century in Indian mathematics by Rudrata, and around the same time in Islamic mathematics by al-Adli ar-Rumi. In 18th century Europe, knight's tours were published by Abraham de Moivre and Leonhard Euler.

阅读后结合上课内容掌握汉密尔顿图的定义。

## 5.8 Dijkstra's Algorithm

Dijkstra's algorithm ) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithms is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF). It is also employed as a subroutine in other algorithms such as Johnson's.

The Dijkstra algorithm uses labels that are positive integers or real numbers, which are totally ordered. It can be generalised to use any labels that are partially ordered, provided the subsequent labels (a subsequent label is produced when traversing an edge) are monotonically non-decreasing. This generalisation is called the generic Dijkstra shortest-path algorithm.

Dijkstra's algorithm uses a data structure for storing and querying partial solutions sorted by distance from the start. While the original algorithm uses a min-priority queue and runs in time $\Theta((|V|+|E|)\log|V|)$ (where $|V|$ is the number of nodes and $|E|$ is the number of edges), it can also be implemented in $\Theta(|V|^2)$ using an array. The idea of this algorithm is also given in Leyzorek et al. 1957. Fredman andTarjan 1984 propose using a Fibonacci heap min-priority queue to optimize the running time complexity to $\Theta(|E|+|V|\log|V|)$. This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. However, specialised cases (such as bounded/integer weights, directed acyclic graphs etc.) can indeed be improved further as detailed in Specialized variants. Additionally, if preprocessing is allowed algorithms such as contraction hierarchies can be up to seven orders of magnitude faster.

In some fields, artificial intelligence in particular, Dijkstra's algorithm or a variant of it is known as uniform cost search and formulated as an instance of the more general idea of best-first search.

**Pseudocode** In the following pseudocode algorithm, dist is an array that contains the current distances from the source to other vertices, i.e. dist[u] is the current distance from the source to the vertex u. The prev array contains pointers to previous-hop nodes on the shortest path from source to the given vertex (equivalently, it is the next-hop on the path from the given vertex to the source). The code u  vertex in Q with min dist[u], searches for the vertex u in the vertex set Q that has the least dist[u] value. Graph.Edges(u, v) returns the length of the edge joining (i.e. the distance between) the two neighbor-nodes u and v. The variable alt on line 14 is the length of the path from the root node to the neighbor node v if it were to go through u. If this path is shorter than the current shortest path recorded for v, that current path is replaced with this alt path.

```
1  function Dijkstra(Graph, source):
2
3      for each vertex v in Graph.Vertices:
4          dist[v]   INFINITY
5          prev[v]   UNDEFINED
6          add v to Q
7      dist[source]   0
8
9      while Q is not empty:
10         u   vertex in Q with min dist[u]
11         remove u from Q
12
13         for each neighbor v of u still in Q:
14             alt   dist[u] + Graph.Edges(u, v)
15             if alt < dist[v]:
16                 dist[v]   alt
17                 prev[v]   u
```

```
18
19      return dist[], prev[]
```

If we are only interested in a shortest path between vertices source and target, we can terminate the search after line 10 if u = target. Now we can read the shortest path from source to target by reverse iteration:

```
1  S    empty sequence
2  u    target
3  if prev[u] is defined or u = source:          // Do something only if the vertex is reachable
4      while u is defined:                       // Construct the shortest path with a stack S
5          insert u at the beginning of S        // Push the vertex onto the stack
6          u    prev[u]                          // Traverse from target to source
```

Now sequence S is the list of vertices constituting one of the shortest paths from source to target, or the empty sequence if no path exists.

A more general problem would be to find all the shortest paths between source and target (there might be several different ones of the same length). Then instead of storing only a single node in each entry of prev[] we would store all nodes satisfying the relaxation condition. For example, if both r and source connect to target and both of them lie on different shortest paths through target (because the edge cost is the same in both cases), then we would add both r and source to prev[target]. When the algorithm completes, prev[] data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the new graph will be the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these shortest paths between two given nodes we would use a path finding algorithm on the new graph, such as depth-first search.

A min-priority queue is an abstract data type that provides 3 basic operations: add_with_priority(), decrease_priority() and extract_min(). As mentioned earlier, using such a data structure can lead to faster computing times than using a basic queue. Notably, Fibonacci heap (Fredman and Tarjan 1984) or Brodal queue offer optimal implementations for those 3 operations. As the algorithm is slightly different, we mention it here, in pseudo-code as well :

```
1  function Dijkstra(Graph, source):
2      dist[source]    0                          // Initialisation
3
```

```
    create vertex priority queue Q

    for each vertex v in Graph.Vertices:
        if v   source
            dist[v]   INFINITY                 // Unknown distance from source to v
            prev[v]   UNDEFINED                // Predecessor of v

        Q.add_with_priority(v, dist[v])


    while Q is not empty:                      // The main loop
        u   Q.extract_min()                    // Remove and return best vertex
        for each neighbour v of u:              // only v that are still in Q
            alt   dist[u] + Graph.Edges(u, v)
            if alt < dist[v]
                dist[v]   alt
                prev[v]   u
                Q.decrease_priority(v, alt)

    return dist, prev
```

Instead of filling the priority queue with all nodes in the initialisation phase, it is also possible to initialise it to contain only source; then, inside the if alt < dist[v] block, the decrease_priority() becomes an add_with_priority() operation if the node is not already in the queue.

Yet another alternative is to add nodes unconditionally to the priority queue and to instead check after extraction that no shorter connection was found yet. This can be done by additionally extracting the associated priority p from the queue and only processing further if p == dist[u] inside the while Q is not empty loop.

These alternatives can use entirely array-based priority queues without decrease-key functionality, which have been found to achieve even faster computing times in practice. However, the difference in performance was found to be narrower for denser graphs.

阅读后结合上课内容掌握 Dijkstra 算法。 推荐此部分的外部参考资源于 B 站 `https://www.bilibili.com/video/BV1Yx411Q7NQ?spm_id_from=333.337.search-card.all.click`

## 5.9 Decision Tree

A decision tree is a decision support tool that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements.

Decision trees are commonly used in operations research, specifically in decision analysis, to help identify a strategy most likely to reach a goal, but are also a popular tool in machine learning.

A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

In decision analysis, a decision tree and the closely related influence diagram are used as a visual and analytical decision support tool, where the expected values (or expected utility) of competing alternatives are calculated.

A decision tree consists of three types of nodes:

- Decision nodes –typically represented by squares
- Chance nodes –typically represented by circles
- End nodes –typically represented by triangles

Decision trees are commonly used in operations research and operations management. If, in practice, decisions have to be taken online with no recall under incomplete knowledge, a decision tree should be paralleled by a probability model as a best choice model or online selection model algorithm.[citation needed] Another use of decision trees is as a descriptive means for calculating conditional probabilities.

Decision trees, influence diagrams, utility functions, and other decision analysis tools and methods are taught to undergraduate students in schools of business, health economics, and public health, and are examples of operations research or management science methods.

阅读后结合上课内容掌握决策树模型的基本结构和其与其他模型的区别（显示特征鉴别性）。

## 5.10 Decision Tree learning

Decision tree learning or induction of decision trees is one of the predictive modelling approaches used in statistics, data mining and machine learning. It uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). Tree models where the target variable can take a discrete set of values are called classification trees; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values (typically real numbers) are called regression trees. Decision trees are among the most popular machine learning algorithms given their intelligibility and simplicity.

In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. In data mining, a decision tree describes data (but the resulting classification tree can be an input for decision making). This page deals with decision trees in data mining.

### 5.10.1 Details

Decision tree learning is a method commonly used in data mining. The goal is to create a model that predicts the value of a target variable based on several input variables.

A decision tree is a simple representation for classifying examples. For this section, assume that all of the input features have finite discrete domains, and there is a single target feature called the "classification". Each element of the domain of the classification is called a class. A decision tree or a classification tree is a tree in which each internal (non-leaf) node is labeled with an input feature. The arcs coming from a node labeled with an input feature are labeled with each of the possible values of the target feature or the arc leads to a subordinate decision node on a different input feature. Each leaf of the tree is labeled with a class or a probability distribution over the classes, signifying that the data set has been classified by the tree into either a specific class, or into a particular probability distribution (which, if the decision tree is well-constructed, is skewed towards certain subsets of classes).

A tree is built by splitting the source set, constituting the root node of the tree, into subsets—which constitute the successor children. The splitting is based on a set of splitting rules based on classification features. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node has all the same values of the target variable, or when splitting no longer adds value to the predictions. This process of top-down induction of decision trees (TDIDT) is an example of a greedy algorithm, and it is by far the most common strategy for learning decision trees from data.

In data mining, decision trees can be described also as the combination of mathematical and computational techniques to aid the description, categorization and generalization of a given set of data.

Data comes in records of the form:

$$(\mathbf{x}, Y) = (x_1, x_2, x_3, ..., x_k, Y)$$

The dependent variable, $Y$, is the target variable that we are trying to understand, classify or generalize. The vector $\mathbf{x}$ is composed of the features, $x_1, x_2, x_3$ etc., that are used for that task.

## 5.11 Information Gain

Used by the ID3, C4.5 and C5.0 tree-generation algorithms. Information gain is based on the concept of entropy and information content from information theory.

Entropy is defined as below

$$\mathrm{H}(T) = \mathrm{I}_E\left(p_1, p_2, \ldots, p_J\right) = -\sum_{i=1}^{J} p_i \log_2 p_i$$

where $p_1, p_2, \ldots$ are fractions that add up to 1 and represent the percentage of each class present in the child node that results from a split in the tree.

$$\begin{aligned}\overbrace{IG(T,a)}^{\text{information gain}} &= \overbrace{\mathrm{H}(T)}^{\text{entropy (parent)}} - \overbrace{\mathrm{H}(T \mid a)}^{\text{sum of entropies (children)}} \\ &= -\sum_{i=1}^{J} p_i \log_2 p_i - \sum_{i=1}^{J} - \Pr(i \mid a) \log_2 \Pr(i \mid a)\end{aligned}$$

Averaging over the possible values of $A$,

$$\begin{aligned}\overbrace{E_A(\mathrm{IG}(T,a))}^{\text{expected information gain}} = \overbrace{I(T;A)}^{\text{mutual information between } T \text{ and } A} &= \overbrace{\mathrm{H}(T)}^{\text{entropy (parent)}} - \overbrace{\mathrm{H}(T \mid A)}^{\text{weighted sum of entropies (children)}} \\ &= -\sum_{i=1}^{J} p_i \log_2 p_i - \sum_{a} p(a) \sum_{i=1}^{J} - \Pr(i \mid a) \log_2 \Pr(i \mid a)\end{aligned}$$

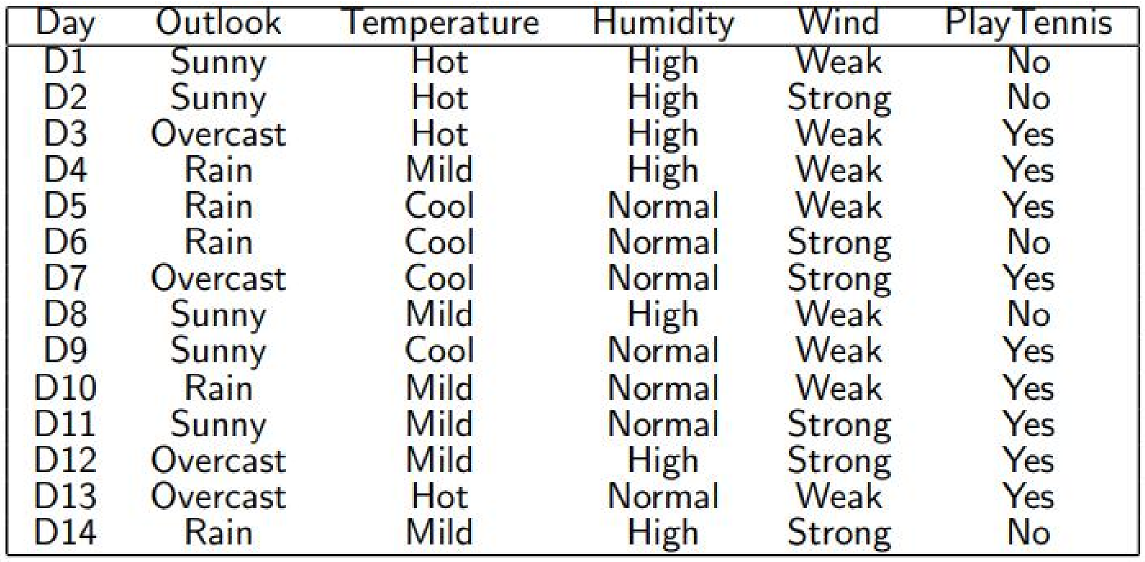
Where weighted sum of entropis is given by,

$$\mathrm{H}(T \mid A) = \sum_{a} p(a) \sum_{i=1}^{J} - \Pr(i \mid a) \log_2 \Pr(i \mid a)$$

That is, the expected information gain is the mutual information, meaning that on average, the reduction in the entropy of T is the mutual information.

Information gain is used to decide which feature to split on at each step in building the tree. Simplicity is best, so we want to keep our tree small. To do so, at each step we should choose the split that results in the most consistent child nodes. A commonly used measure of consistency is called information which is measured in bits. For each node of the tree, the information value "represents the expected amount of information that would be needed to specify whether a new instance should be classified yes or no, given that the example reached that node".

Consider an example data set with four attributes: outlook (sunny, overcast, rainy), temperature (hot, mild, cool), humidity (high, normal), and windy (true, false), with a binary (yes or no) target variable, play, and 14 data points. To construct a decision tree on this data, we need to compare the information gain of each of four trees, each split on one of the four features. The split with



| Day | Outlook | Temperature | Humidity | Wind | PlayTennis |
|---|---|---|---|---|---|
| D1 | Sunny | Hot | High | Weak | No |
| D2 | Sunny | Hot | High | Strong | No |
| D3 | Overcast | Hot | High | Weak | Yes |
| D4 | Rain | Mild | High | Weak | Yes |
| D5 | Rain | Cool | Normal | Weak | Yes |
| D6 | Rain | Cool | Normal | Strong | No |
| D7 | Overcast | Cool | Normal | Strong | Yes |
| D8 | Sunny | Mild | High | Weak | No |
| D9 | Sunny | Cool | Normal | Weak | Yes |
| D10 | Rain | Mild | Normal | Weak | Yes |
| D11 | Sunny | Mild | Normal | Strong | Yes |
| D12 | Overcast | Mild | High | Strong | Yes |
| D13 | Overcast | Hot | Normal | Weak | Yes |
| D14 | Rain | Mild | High | Strong | No |

Figure 5.1: Shall I play tennis today? A sample for decision tree problem.

the highest information gain will be taken as the first split and the process will continue until all children nodes each have consistent data, or until the information gain is 0.

To find the information gain of the split using windy, we must first calculate the information in the data before the split. The original data contained nine yes's and five no's.

$$I_E([9,5]) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.94$$

The split using the feature windy results in two children nodes, one for a windy value of true and one for a windy value of false. In this data set, there are six data points with a true windy value, three of which have a play (where play is the target variable) value of yes and three with a play value of no. The eight remaining data points with a windy value of false contain two no's and six yes's. The information of the windy=true node is calculated using the entropy equation above. Since there is an equal number of yes's and no's in this node, we have

$$I_E([3,3]) = -\frac{3}{6}\log_2\frac{3}{6} - \frac{3}{6}\log_2\frac{3}{6} = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1$$

For the node where windy=false there were eight data points, six yes's and two no's. Thus we have

$$I_E([6,2]) = -\frac{6}{8}\log_2\frac{6}{8} - \frac{2}{8}\log_2\frac{2}{8} = -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} = 0.81$$

To find the information of the split, we take the weighted average of these two numbers based on how many observations fell into which node.

$$I_E([3,3],[6,2]) = I_E(\text{windy or not}) = \frac{6}{14}\cdot 1 + \frac{8}{14}\cdot 0.81 = 0.89$$

Now we can calculate the information gain achieved by splitting on the windy feature.

$$\mathrm{IG}(\mathrm{windy}) = I_E([9,5]) - I_E([3,3],[6,2]) = 0.94 - 0.89 = 0.05$$

To build the tree, the information gain of each possible first split would need to be calculated. The best first split is the one that provides the most information gain. This process is repeated for each impure node until the tree is complete. This example is adapted from the example appearing in Witten et al.

阅读后结合上课内容掌握信息增益的计算方式，以及理解其在决策树学习过程中的意义。

## 5.12 ID3

In decision tree learning, ID3 (Iterative Dichotomiser 3) is an algorithm invented by Ross Quinlan used to generate a decision tree from a dataset. ID3 is the precursor to the C4.5 algorithm, and is typically used in the machine learning and natural language processing domains.

The ID3 algorithm begins with the original set $S$ as the root node. On each iteration of the algorithm, it iterates through every unused attribute of the set $S$ and calculates the entropy $\mathrm{H}(S)$ or the information gain $IG(S)$ of that attribute. It then selects the attribute which has the smallest entropy (or largest information gain) value. The set $S$ is then split or partitioned by the selected attribute to produce subsets of the data. (For example, a node can be split into child nodes based upon the subsets of the population whose ages are less than 50, between 50 and 100, and greater than 100.) The algorithm continues to recurse on each subset, considering only attributes never selected before.

Recursion on a subset may stop in one of these cases: every element in the subset belongs to the same class; in which case the node is turned into a leaf node and labelled with the class of the examples. there are no more attributes to be selected, but the examples still do not belong to the same class. In this case, the node is made a leaf node and labelled with the most common class of the examples in the subset. there are no examples in the subset, which happens when no example in the parent set was found to match a specific value of the selected attribute. An example could be the absence of a person among the population with age over 100 years. Then a leaf node is created and labelled with the most common class of the examples in the parent node's set. Throughout the algorithm, the decision tree is constructed with each non-terminal node (internal node) representing the selected attribute on which the data was split, and terminal nodes (leaf nodes) representing the class label of the final subset of this branch.

### 5.12.1 Details

- Calculate the entropy of every attribute $a$ of the data set $S$.
- Partition ("split") the set $S$ into subsets using the attribute for which the resulting entropy after splitting is minimized; or, equivalently, information gain is maximum.
- Make a decision tree node containing that attribute.
- Recurse on subsets using the remaining attributes.

**Pseudocode**

```
ID3 (Examples, Target_Attribute, Attributes)
    Create a root node for the tree
    If all examples are positive, Return the single-node tree Root, with label = +.
    If all examples are negative, Return the single-node tree Root, with label = -.
    If number of predicting attributes is empty, then Return the single node tree Root,
    with label = most common value of the target attribute in the examples.
    Otherwise Begin
        A   The Attribute that best classifies examples.
        Decision Tree attribute for Root = A.
        For each possible value, vi, of A,
            Add a new tree branch below Root, corresponding to the test A = vi.
            Let Examples(vi) be the subset of examples that have the value vi for A
            If Examples(vi) is empty
                Then below this new branch add a leaf node with label = most common target value in the examples
            Else below this new branch add the subtree ID3 (Examples(vi), Target_Attribute, Attributes - {A})
    End
    Return Root
```

ID3 does not guarantee an optimal solution. It can converge upon local optima. It uses a greedy strategy by selecting the locally best attribute to split the dataset on each iteration. The algorithm's optimality can be improved by using backtracking during the search for the optimal decision tree at the cost of possibly taking longer.

ID3 can overfit the training data. To avoid overfitting, smaller decision trees should be preferred over larger ones. This algorithm usually produces small trees, but it does not always produce the smallest possible decision tree.

ID3 is harder to use on continuous data than on factored data (factored data has a discrete number of possible values, thus reducing the possible branch points). If the values of any given attribute are continuous, then there are many more places to split the data on this attribute, and searching for the best value to split by can be time consuming.

阅读后结合上课内容掌握 ID3 算法（包含其所有运算细节）。推荐此部分的外部参考资源于 B 站 `https://www.bilibili.com/video/BV1SJ411N7yb?share_source=copy_web`