Elementary Introduction to Graph Theory

(01EIG 2025/2026)

Lecture 5

Francesco Dolce
dolcefra@fit.cvut.cz

October 31, 2025

updated: October 30, 2025

PDF available at the address: dolcefra.pages.fit/ens/2526/EIG-lecture-05.pdf

Solution of Exercise in previous Lecture. Let us order the edges of G according to their weights: $e_1 < e_2 < \cdots < e_{10}$ (this order is not unique). The label ad weight of each vertex is shown on the left of Figure 1. The minimal spanning tree obtained using Kruskal's algorithm is shown on the right of the same figure.

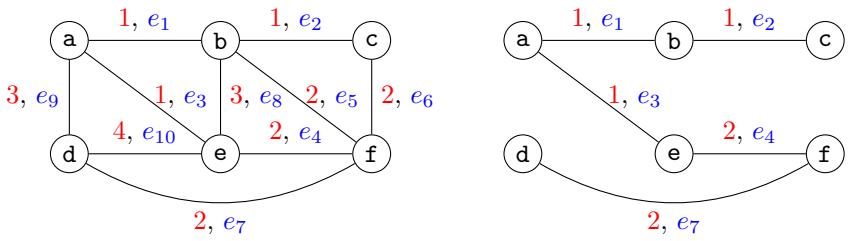


Figure 1: A weighted graph and one of its minimal spanning tree.

1 Time complexity, Union-Find operation

The time complexity of the algorithms seen in the previous lecture depends on the used data structures. For instance, in Kruskal's Algorithm, time complexity heavily depends on the choice of data structure used in the step where the algorithm has to test whether by addition of an edge we create a cycle in G_i.

To test whether we have a cycle is quite simple: it is enough to remember for each vertex v the component the vertex belong to: a new edge $\{x, y\}$ does create a cycle if and only if x and y already belong to the same component.

A naive implementation could be to use an array to store the label of its component for each vertex.

The operation FIND (u) which finds the label of the component containing the vertex v has time complexity $\mathcal{O}(1)$. This seems to be perfect.

However, there is another operation we need to implement: the UNION $(\cdot)$ operation. Suppose we are trying to add an edge $e = \{x, y\}$. If x and y are in the same component, we skip e. If x and y are in different component – i.e., if FIND $(x) \neq$ FIND (y) – then we add e to the graph. This operation merges the components containing x and y into a single component.

In our naive implementation we have to go through the whole array used to store the labels of components and to perform the merging.

For example, if x is in component A and y is in component B, we must scan the entire array and replace all instances of B with A.

This operation has time complexity $\mathcal{O}(n)$, where $n = \#V$.

The total time complexity of Kruskal's Algorithm with naive implementation is thus $\mathcal{O}(m \cdot n)$, where $m = \#E$ and $n = \#V$, since we have at most m steps of the algorithm.

We can consider a more elaborate implementation of UNION-FIND scheme.

- We represent individual components by trees.
- The root of a tree is the label of the component, or a representative of the component.
- FIND (a): find out the root of the tree representing the component containing the vertex a. The complexity is proportional to the length of the tree, i.e., $\mathcal{O}(\ln n)$.
- UNION (a, b): we need to merge two components. We merge the trees representing both components so that the height of the resulting tree is as small as possible. To achieve this goal we connect the trees so that the root of the smaller tree is a child of the root of the longer tree. In this way the height increases at most by 1. The complexity is $\mathcal{O}(1)$ – if we remember the length of trees and we ran FIND $(\cdot)$ before UNION $(\cdot)$ – or $\mathcal{O}(\ln n)$.
- The total time complexity is $\mathcal{O}(m \cdot \ln n)$, where $m = \#E$ and $n = \#V$.

Example 1 Let G be a graph having two components A and X as in the left of Figure 2. One can check that $\textsc{find}(\mathtt{b}) = A$ and $\textsc{find}(\mathtt{z}) = B$. Two possible spanning trees of the components are shown in center of the same figure (we select `a` as representative of A and `x` as representative of X). The union of the two components containing `b` and `z` is shown on the right of the figure.

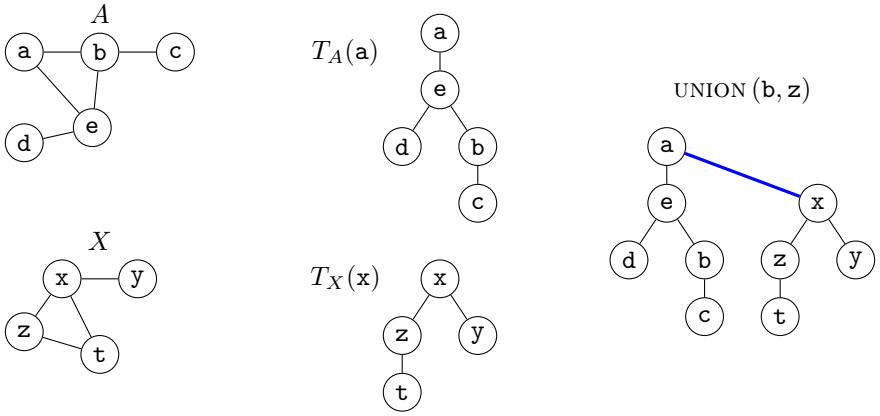


Figure 2: Union-Find scheme using trees.

We discussed the Union-Find scheme to demonstrate how algorithmic efficiency can depend on the choice of data structure. This scheme, also called *disjoint-set data structure* is a very general data structure that has different applications.

In practice, a faster version of the Union-Find scheme is usually used: tree representation with path compression. This version achieves near constant *amortized* time complexity for each operation. More precisely, for a sequence of m Union-Find applications on a set with n nodes, the total time required is $\mathcal{O}(m \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function.

2 Euler tours

Let $G = (V, E)$ be a (non necessarily simple) graph of order n and size m. A walk $(v_0 \overset{e_1}{\to} v_1 \overset{e_2}{\to} \cdots \overset{e_m}{\to} v_m)$ is called an *Eulerian walk* if for every $1 \leq i, j \leq m$ with $i \neq j$ we have $e_i \neq e_j$. Thus, an Eulerian walk is a trail that traverses every edge, i.e., every edge in E is used exactly once.

An *Eulerian closed walk* is called an *Euler tour*. A graph is called *Eulerian* if it admits an Euler tour.

Example 2 Let G be the simple graph in Figure 3. An Eulerian walk in the graph is given by the path $(e_5, e_1, e_2, e_3, e_4, e_8, e_7, e_6)$. There is no Euler tour in G, so the graph is not Eulerian.

One can also check that the graph K in the same figure is not Eulerian and does not contain any Eulerian walks neither.

Figure 3: Two non-Eulerian graphs.

A necessary condition for having an Eulerian walk or an Euler tour in a graph is connectedness. This condition is not sufficient, though, as shown in Example 2.

Theorem 1 (Euler (1736)) *A connected graph is Eulerian if and only if it is even.*

Proof.

($\Rightarrow$) Every vertex appearing k times in an Euler tour must have degree $2k$.

($\Leftarrow$) Let G be a connected even graph. We construct an Euler tour in the following way.

0) • $i \leftarrow 1$.
   • Let us choose a random starting vertex v_i and a colour c_i.

1) Walk, as long as possible, along the edges (at random) of G. When leaving a vertex choose an edge with no colour. While traversing an edge, colour it using c_i.

2) If there is no way how to continue, we have returned to the starting vertex v_i and, moreover, all edges incident with v_i have already been coloured.
   a) If there are no colourless edges go to Step 3) (RECONSTRUCTION).
   b) If there are some colourless edges left:
      • $i \leftarrow i + 1$.
      • Choose a new colour c_i that has not been used so far.
      • Choose a new starting vertex v_i such that there are some colourless as well as some coloured edges incident with it (it is possible to find such a vertex v_i due to the assumption that G is connected).
      • Go to Step 1).

3) (RECONSTRUCTION) All edges of G have been coloured, i.e., the set E is partitioned by a set of disjoint closed walks $\{W_i\}_i$. We have to join these walks into one Euler tour.
   i) Start at v_1, follow the edges of the first closed walk.
   ii) If we meet a starting vertex v_k of a closed walk W_k which has not yet been processed (i.e., joined into the Euler tour), we "transfer" to W_k, circle the whole closed walk W_k before returning back to the edges of the previous walk.

■

By applying rule *ii*) in Theorem 1, the walks can "nest" several times and they can be proessed in an order different to the order in which they have been created.

Example 3 Let H be the graph in Figure 4.

(Step 0) We start by choosing the vertex `b` and the colour red.

(Step 1) We consider the closed walk $W_r = (\texttt{b} \overset{e_1}{\to} \texttt{d} \overset{e_2}{\to} \texttt{c} \overset{e_3}{\to} \texttt{a} \overset{e_4}{\to} \texttt{b} \overset{e_5}{\to} \texttt{e} \overset{e_6}{\to} \texttt{c} \overset{e_7}{\to} \texttt{b})$.

(Step 2) We choose the vertex `d` and the colour green.

(Step 1) We consider the closed walk $W_g = (\texttt{d} \overset{e_8}{\to} \texttt{h} \overset{e_9}{\to} \texttt{g} \overset{e_{10}}{\to} \texttt{d})$.

(Step 2) We choose the vertex `g` and the colour blue.

(Step 1) We consider the closed walk $W_b = (\texttt{g} \overset{e_{11}}{\to} \texttt{f} \overset{e_{12}}{\to} \texttt{e} \overset{e_{13}}{\to} \texttt{g})$.

(Step 2) All edges are coloured.

(Step 3) The Euler tour is $(e_1, e_8, e_9, e_{11}, e_{12}, e_{13}, e_{10}, e_2, e_3, e_4, e_5, e_6, e_7)$.

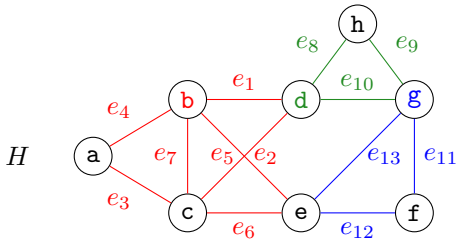


Figure 4: Finding an Eulerian tour in a graph.

The proof of Theorem 1 gives an algorithm for finding Euler tours in an Eulerian graph. There is, however, a more efficient algorithm based on Tarry's traversal algorithm with complexity $\mathcal{O}(m+n)$, where m and n are respectively the size and the order of the graph.

- Choose a starting vertex v_0.
- Traverse G using as long as possible the two rules:

  (R1) no edge can be used twice in the same direction;

  (R2) the order in which we choose an edge when leaving a vertex u is:

  a) an edge which has not yet been used,

  b) an edge used to arrive to u with the exception of the edge of first arrival,

  c) the edge of first arrival to u.

  During the traversal we record the edge of first arrival for every vertex v (except at the beginning for v_0), and add the edges in a sequence called *return sequence*, according to the order in which they are used for the second time.

- When there is no way to proceed in the traversal using rules (R1) and (R2): we have returned to v_0, every edge has been traversed exactly twice (once in every direction); and one can prove that the return sequence corresponds to an Euler tour.

Example 4 Let H be the graph in Example 3. Let us show an application of the algorithm described above on H, starting from the vertex b (see Figure 5).

Using rules (R1) and (R2) we can find a closed walk

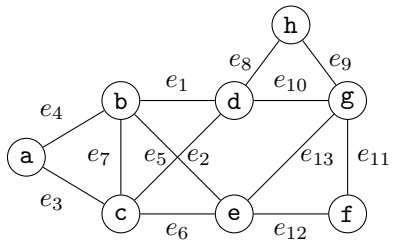
$$(\mathtt{b} \xrightarrow{e_1} \mathtt{d} \xrightarrow{e_2} \mathtt{c} \xrightarrow{e_3} \mathtt{a} \xrightarrow{e_4} \mathtt{b} \xrightarrow{e_5} \mathtt{e} \xrightarrow{e_6} \mathtt{c} \xrightarrow{e_7} \mathtt{b} \xrightarrow{e_7} \mathtt{c} \xrightarrow{e_6} \mathtt{e} \xrightarrow{e_{12}} \mathtt{f} \xrightarrow{e_{11}} \mathtt{g} \xrightarrow{e_{13}} \mathtt{e} \xrightarrow{e_{13}} \mathtt{g}$$
$$\xrightarrow{e_9} \mathtt{h} \xrightarrow{e_8} \mathtt{d} \xrightarrow{e_{10}} \mathtt{g} \xrightarrow{e_{10}} \mathtt{d} \xrightarrow{e_8} \mathtt{h} \xrightarrow{e_9} \mathtt{g} \xrightarrow{e_{11}} \mathtt{f} \xrightarrow{e_{12}} \mathtt{e} \xrightarrow{e_5} \mathtt{b} \xrightarrow{e_4} \mathtt{a} \xrightarrow{e_3} \mathtt{c} \xrightarrow{e_2} \mathtt{d} \xrightarrow{e_1} \mathtt{b}),$$

where we colour in blue the first arrival to a vertex, and in red the return sequence.

Thus, an Euler tour (closed Eulerian walk) over H is given by

$$(e_7, e_6, e_{13}, e_{10}, e_8, e_9, e_{11}, e_{12}, e_5, e_4, e_3, e_2, e_1).$$

H



vertex	first arrival
a	e_3
b	e_4
c	e_2
d	e_1
e	e_5
f	e_{12}
g	e_{11}
h	e_9

Figure 5: Finding an Eulerian tour in a graph.

Remember that the number of vertices with odd degrees in a graph is even. Therefore, in particular, it cannot be 1.

Theorem 2 *A connected graph has an Eulerian trail if and only if the number of vertices with odd degree is at most* 2 *(i.e., either* 0 *or* 2*).*

Example 5 The graphs G and K in Example 2 are not Eulerian. Indeed they are connected but not even, since, e.g., $d_G(\mathtt{e}) = 3$ and $d_K(A) = 5$.

The graph G has an Eulerian walk, since only `d` and `e` have odd degrees.

The graph K does not have an Eulerian path since it has 4 vertices of odd degree.

3 Hamilton cycles

A dual problem of the one of finding an Euler tour in a graph is the one of finding a cycle visiting every vertex exactly once.

We call *Hamilton path* and *Hamilton cycle* respectively a spanning path and a spanning cycle in a graph G.

Thus, every Hamilton path (resp., Hamilton cycle) in $G = (V, E)$ has length $\#V - 1$.

A graph containing a Hamilton path is called *traceable*. A graph containing a Hamilton cycle is called *Hamiltonian*. Clearly, every Hamiltonian graph is also traceable.

Example 6 The G graph in Example 2 is Hamiltonian. A Hamilton path and a Hamilton cycle of the graph are shown in Figure 6.

Figure 6: A Hamilton path (on the left) and a Hamilton cycle (on the right).

There is a major difference in comparison to the problem of the existence of Euler tours. To determine whether or not a given graph has a Hamilton cycle is much harder. No good characterisation is known or even expected to exist (the problem is a $\mathcal{NP}$-complete problem).

Path exchanges. A natural way of looking for a Hamilton path is by extending a path as much as possible. Let suppose that we found a path $P = (v_1, \ldots, v_k)$ in a graph G and let $v \in \mathrm{N}_G(v_k) \setminus \{v_k\}$ (we want to avoid loops).

- If v is not in P, i.e., if $v \neq v_i$ for all $1 \leq i \leq k$, then we can extend the longer path to $P' = P + \{(v_k, v)\}$.
- If v is in the path but $v \neq v_{k-1}$, i.e., $v = v_i$ for some $1 \leq i \leq k-2$, then we can obtain a path of the same length by exchanging $\{v_i, v_{i+1}\}$ with $\{v_k, v_i\}$, i.e., considering the path $P'' = (P \setminus \{v_i, v_{i+1}\}) + \{(v_k, v_i)\}$ (see Figure 7).

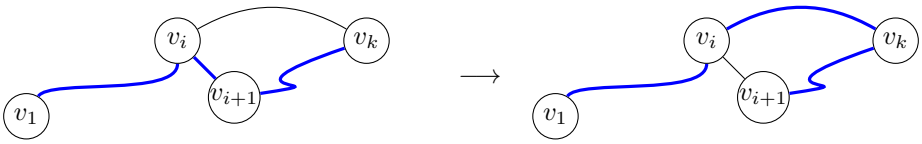


Figure 7: A path exchange.

Cycle exchange. In a similar way, let us suppose that we have a cycle $C = (v_1, \ldots, v_k)$ having two vertices v_i, v_j non consecutive in C, i.e., with $|i-j| > 1$ and such that both $\{v_i, v_j\}$ and $\{v_{i+1}, v_{j+1}\}$ are edges of the graph. Then we can obtain a new cycle

$$C' = (C \setminus \{\{v_i, v_{i+1}\}, \{v_j, v_{j+1}\}\}) + \{\{v_i, v_j\}, \{v_{i+1}, v_{j+1}\}\}$$

having the same length as C (see Figure 8).

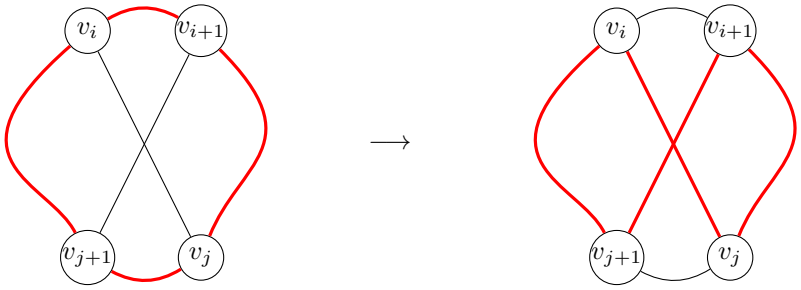


Figure 8: A path exchange.

There are several known sufficient conditions for the existence of a Hamilton cycle in a graph.

Every complete graph of order at least 3 is Hamiltonian, since a Hamilton cycle can be obtained by selecting all vertices one by one in an arbitrary order.

Example 7 The complete graph K_5 over set of vertices $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ has, e.g., the Hamilton cycle: $(\mathtt{b}, \mathtt{c}, \mathtt{a}, \mathtt{e}, \mathtt{d})$ (see Figure 9).

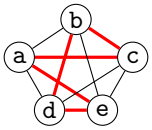


Figure 9: A Hamilton cycle in K_5.

When we have fewer edges, finding a Hamilton cycle is harder. Dirac proved what is the minimum degree that a graph must have to guarantee the existence of a Hamilton cycle.

Theorem 3 (Dirac (1952)) *Every simple graph G of order $n \geq 3$ such that $\delta(G) \geq \frac{n}{2}$ is Hamiltonian.*

Dirac's Theorem has the best possible bound on $\delta(G)$ to ensure the existence of a Hamilton cycle. We cannot, e.g., replace $\frac{n}{2}$ with $\lfloor \frac{n}{2} \rfloor$.

Example 8 Let n be an odd number and G be the union of two complete graphs $K_{\lceil \frac{n}{2} \rceil}$ meeting in one vertex w (see Figure 10, where $n = 7$). Then $\delta(G) = \lfloor \frac{n}{2} \rfloor$ but G cannot have a Hamilton cycle as it would have to go through w twice.

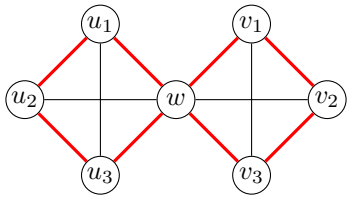


Figure 10: A closed walk visiting all vertices passing through w twice.

We can find other sufficient conditions.

Theorem 4 (Ore (1960)) *Let $G = (V, E)$ be a simple graph and let $u, v \in V$ such that $d_G(u) + d_G(v) \geq \#V$. Then G is Hamiltonian if and only if $G + \{u, v\}$ is Hamiltonian.*

Proof.

($\Rightarrow$) If G has a Hamilton cycle, then clearly the same cycle is also in $G + \{u, v\}$.

($\Leftarrow$) If $G + \{u, v\}$ has a Hamilton cycle C, then we can find a Hamilton cycle C' by using a cycle exchange (Exercise).

■

The *closure* of a (simple) graph $G = (V, E)$ is the graph obtained from G by recursively adding edges $\{u, v\}$ if $\{u, v\}$ is not an edge in the graph and the sum

of the two degrees is at least $\#V$, until all vertices of such form are adjacent. One can prove that the order in which the edges are added does not change the final result (why?).

Example 9 Let G be the graph of Example 2. The closure of G is the complete graph K_5 (see Figure 11).

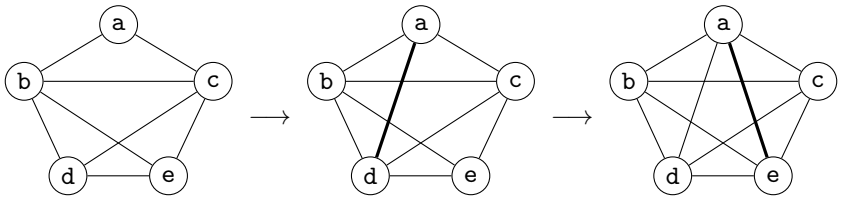


Figure 11: Closure of a graph.

A consequence of Theorem 4 is the following.

Corollary 1 *A simple graph is Hamiltonian if and only if its closure is Hamiltonian.*

There exist other sufficient condition for a graph to have a Hamilton cycle. Most of them requires the graph to have "enough" edges.

Theorem 5 (Pósa (1962)) *Let $G = (V, E)$ be a simple graph of order n realised by a graphic sequence $(d_i)_{i=1}^n$ with $d_1 \leq d_2 \leq \ldots \leq d_n$. If for every $k < \frac{n}{2}$ we have $d_k > k$, then G is Hamiltonian.*

Example 10 The Hamiltonian graph G of Example 2 is realised by the graphic sequence $(d_i)_{i=1}^5 = (2, 3, 3, 4, 4)$. For $k < \frac{5}{2}$, i.e., for $k = 1, 2$ we have $d_1 = 2 > 1$ and $d_2 = 3 > 2$.

Theorem 6 (Chvátal (1972)) *Let $G = (V, E)$ be a simple graph of order $n \geq 3$ realised by the graphic sequence $(d_i)_{i=1}^n$ with $d_1 \leq d_2 \leq \ldots \leq d_n$. If for every $k < \frac{n}{2}$ either $d_k > k$ or $d_{n-k} \geq n - k$, then H is Hamiltonian.*

Example 11 Let G be the graph shown in Figure 12. This graph has order 7 and it is realised by the graphic sequence $(d_i)_{i=1}^7 = (2, 3, 3, 5, 5, 5, 5)$. One has $d_3 = 3 \leq 3$ but $d_4 = 5 \geq 4$. Moreover, $d_1 = 2 > 1$ and $d_2 = 3 > 2$. So G satisfies Chvátal's condition but not Pósa's condition. A Hamilton cycle is shown in red in the same figure.

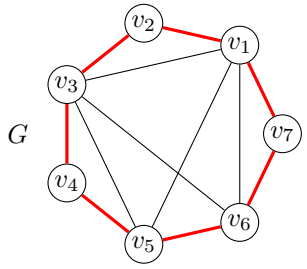


Figure 12: A Hamilton cycle in a graph.

Exercise. Let us consider the two graphs in Figure 13 (the one on the right is called *Petersen graph*).

1. Do the graphs have Eulerian trails?
2. Do the graphs have Eulerian tours?
3. Do the graphs have Hamilton paths?
4. Do they have Hamilton cycles?
5. Do they satisfy Dirac/Pósa/Chvátal conditions?

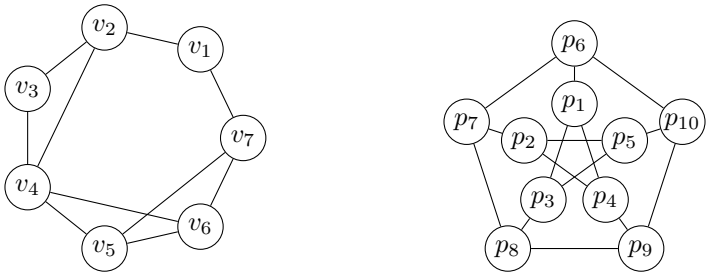


Figure 13: Two graphs (on the right the Petersen graph).

The *Travelling Salesperson Problem* (TSP) is a classical $\mathcal{NP}$-hard problem: given a weighted graph (G, w) find a minimum-weight Hamilton cycle in G.

Exercise.[Trick-or-Treat Spooky Problem (TSP)] It's Halloween Night and you decide to go around in your neighbourhood to collect as many candies as you can. The distance between the houses are represented in Table 1 and Figure 14.

Find the optimal route that allows you to leave and return to your home (H) collecting treats at every door, without having to knock at the same door twice.

	H	**b**	**c**	**d**	**e**
H	–	3	6	7	4
b	3	–	2	5	6
c	6	2	–	3	5
d	7	5	3	–	2
e	4	6	5	2	–

Table 1: Distance between houses in the neigbourhood.

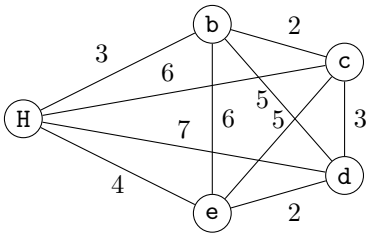


Figure 14: A plan of the neighbourhood.