Problem-Sheet 1

My Solution

Michael Single

08-917-445

msingle@students.unibe.ch

Task 1:

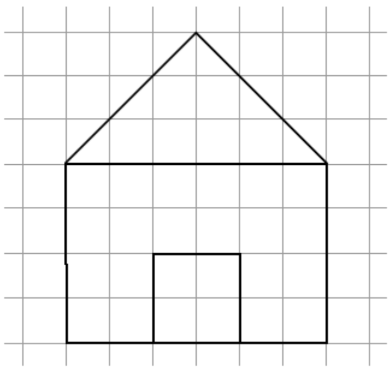	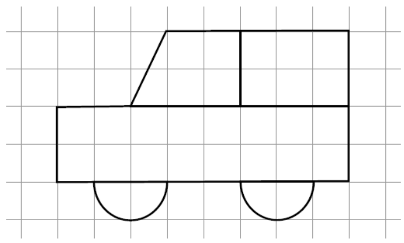	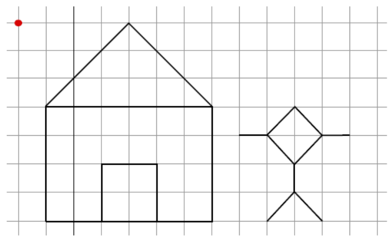
Figure 1.1	**Figure 1.2**	**Figure 1.3**

For a given *Plotter* that moves a pen in any direction compute the <u>minimal time</u> to plot the figures *figure 1.1*, *figure 1.2* and *figure 1.3*.

Assumptions:

1. Moving the pen by one grid unit takes **one second**.
2. **Time** used to **lift** the pen can be **neglected**.
3. After drawing a figure, the plotter has to **end** at the position **where** it **started** to plot initially.
4. We only draw an edge once. When revisting an already drawn edge we have to perform an pen.up() operation. This operation raises the pen so that it will not draw the edge once again. However, the plotter will then move along the edge but with an upraised pen. After moving along such an edge, the plotter's pen has to be lowered again by Pen.lower().

Note that **Assumption 3.** is not directly stated on the exercise sheet. However, this assumption was also made during the first class. Therefore, our task is to find the shortest path visiting all the edges of the fiven drawing just once and end up at the same position where we started initially. Such a path is called Euler-Tour.

Our first step is to label necessary vertices of the given figures f*igure 1.1*, f*igure 1.2* and f*igure 1.3*.

And then find a shortest path according to our assumptions. Lastely, we have to sum up the overall temporal cost to draw such a shortest path for each given figure. For this exercise I will provide figures showing such shortest paths and the corresponding temporal costs for each of the figures f*igure 1.1*, f*igure 1.2* and f*igure 1.3*.

NB1: A transition A → B indicated the movement of the plotter from node A to node B.

NB2: used formulas in order to compute the temporal costs of the transitions:

- **Circumference** of a circle with radius r: 2*r***π**
- **Diagonal** of a triangle: by using Pythagoras

NB3: *sqrt(x)* defines the square root of the number x.

An **Euler-Tour** in a Graph (see **def**.(**Graph**) in section Task 2 below) is a path which visits each edge exactly once and furthermore ends where it started.

Lemma(**Euler-Tour**): If and only if every vertex has an even degree then there exists an so called Euler-Tour .

A Plotter can perform two elementary operations:

- Draw, i.e. Moving while drawing a line.
- and moveUp(), i.e. Moving while its pen is raised – does not print but moves.

NB4: Every movement of the plotter (regardless which operation it performs – draw or moveUp) from a vertex to another corresponds to connecting two vertices by an edge.

Subtask Figure 1.1:

Labeling the vertices:

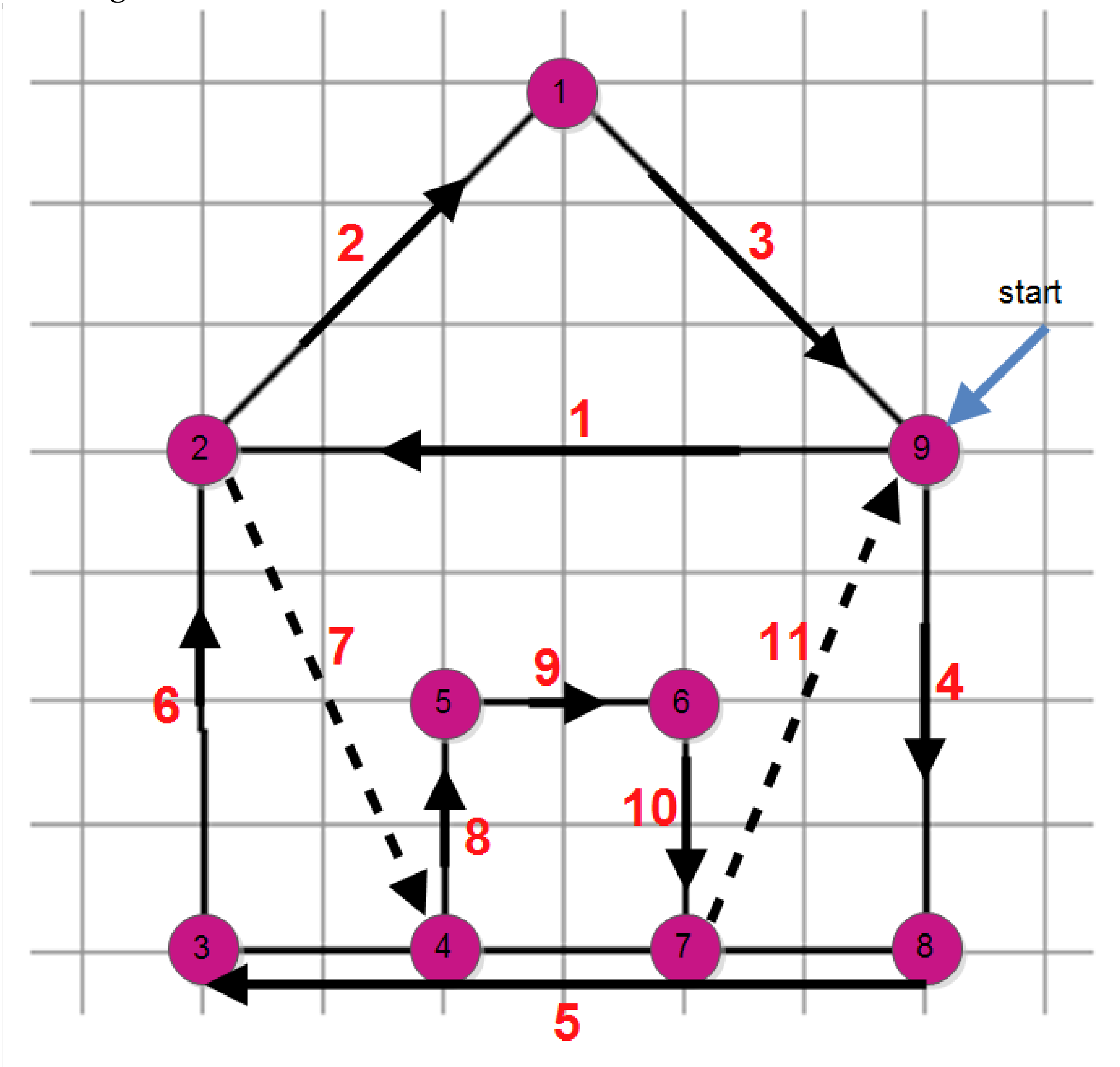

Figure 1.1.1: Labeled vertices (purple) and a solution path black arrows (the red number indicates the step number, also listed in below's table). An arrow indicates a the plotter movement from a vertex to another. Dotted arrows indicate penUp() plotter movements.

Initially, the vertices 2,4, 7 and 9 have an odd degree. Thus, in order to find an Euler-Tour, we have to add two additional – moved up – edges (the one from 2 to 4 and the one from 7 to 9).

Temporal path costs: The following table shows the temporal cost resulting for drawing **figure 1.1**. accoding to my solution from **figure 1.1.1** (red enumerated black arrows):

Step Number	Action	Transition	Temporal Cost [s]
1	draw	$9 \rightarrow 2$	6
2	draw	$2 \rightarrow 1$	3*sqrt(2)
3	draw	$1 \rightarrow 9$	3*sqrt(2)
4	draw	$9 \rightarrow 8$	4
5	draw	$8 \rightarrow 3$	6
6	draw	$3 \rightarrow 2$	4
7	Up movement	$2 \rightarrow 4$	2*sqrt(5)
8	draw	$4 \rightarrow 5$	2
9	draw	$5 \rightarrow 6$	2
10	draw	$6 \rightarrow 7$	2
11	Up movement	$7 \rightarrow 9$	2*sqrt(5)
		Total ∑	26+6*sqrt(2) + 4*sqrt(5) = *43.430 seconds*

According to my solution it takes about 43.43 seconds in order to plot **figure 1.1**. Note that there is just an additional cost of *2*2*sqrt(5) seconds* required (*penUp()* movements, indicated by the black dotted arrows in **figure 1.1.1**)

Subtask Figure 1.2:

Labeling the vertices:

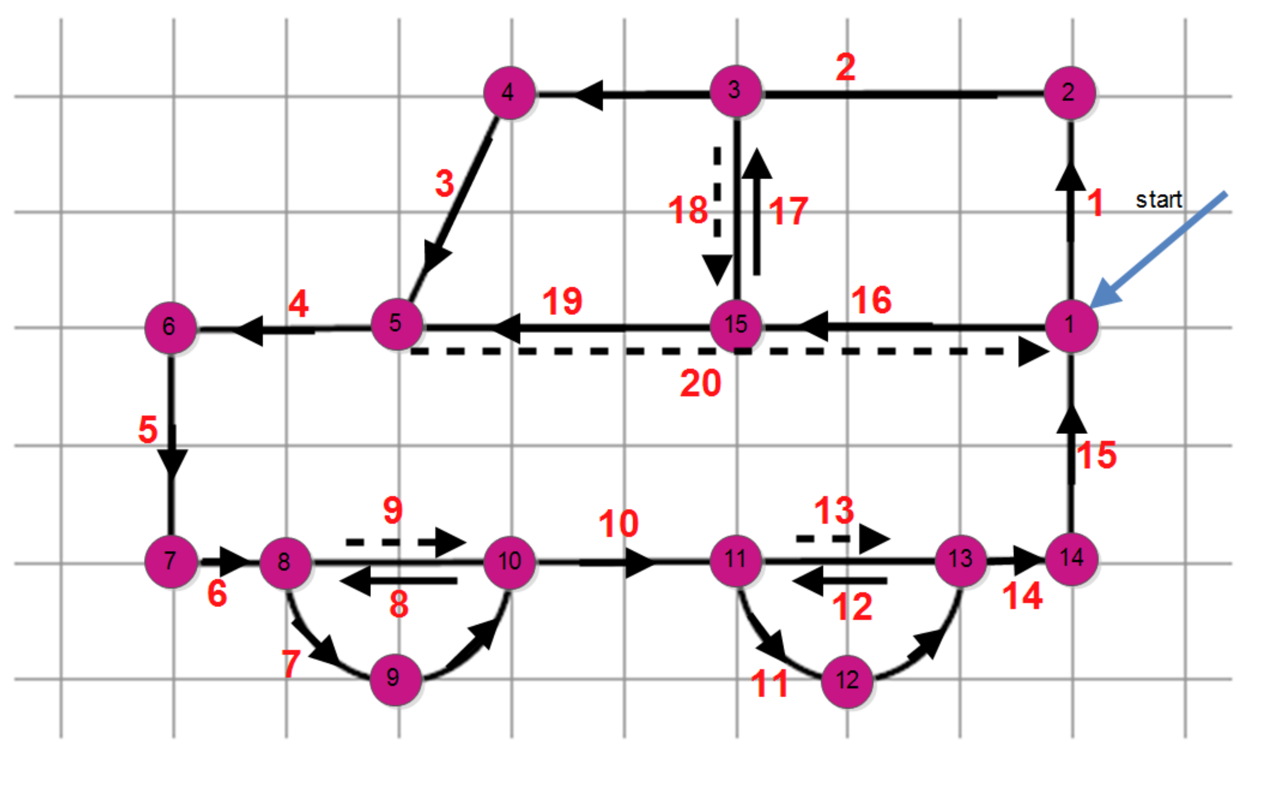

Figure 1.2.1 Labeled vertices (purple) and a solution path black arrows (the red number indicates the step number, also listed in below's table). An arrow indicates a the plotter movement from a vertex to another. Dotted arrows indicate penUp() plotter movements.

Initially, the vertices 1, 3,5, 8, 10, 11, 13, and 15 have an odd degree. Thus, in order to find an Euler-Tour, we have to add two additional – moved up – edges.

Temporal path costs: The following table shows the temporal cost resulting for drawing **figure 1.2.** accoding to my solution from **figure 1.2.1** (red enumerated black arrows):

Temporal path costs:

Step Number	Action	Transition	Temporal Cost [s]
1	draw	1 → 2	2
2	draw	2 → 4	5
3	draw	4 →5	sqrt(5)
4	draw	5 →6	2
5	draw	6 →7	2
6	draw	7 →8	1
7	draw	8 →9 →10	**π**
8	draw	10 →8	2
9	Up movement	8 →10	2
10	draw	10 →11	1
11	draw	11 →12 → 13	**π**
12	draw	13 →11	2
13	Up movement	11 → 13	2
14	draw	13 →14	1
15	draw	14 → 1	2
16	draw	1 → 15	3
17	draw	15 → 3	2
18	Up movement	3 → 15	2
19	draw	15 →5	3
20	Up movement	5 →1	6
		Total ∑	40+2***π** + sqrt(5) = *48.520 seconds*

According to my solution it takes about 48.520 seconds in order to plot **figure 1.2**. Note that there is just an additional cost of *12 seconds* required (*penUp()* movements, indicated by the black dotted arrows in **figure 1.2.1**)

Subtask Figure 1.3:

Labeling the vertices:

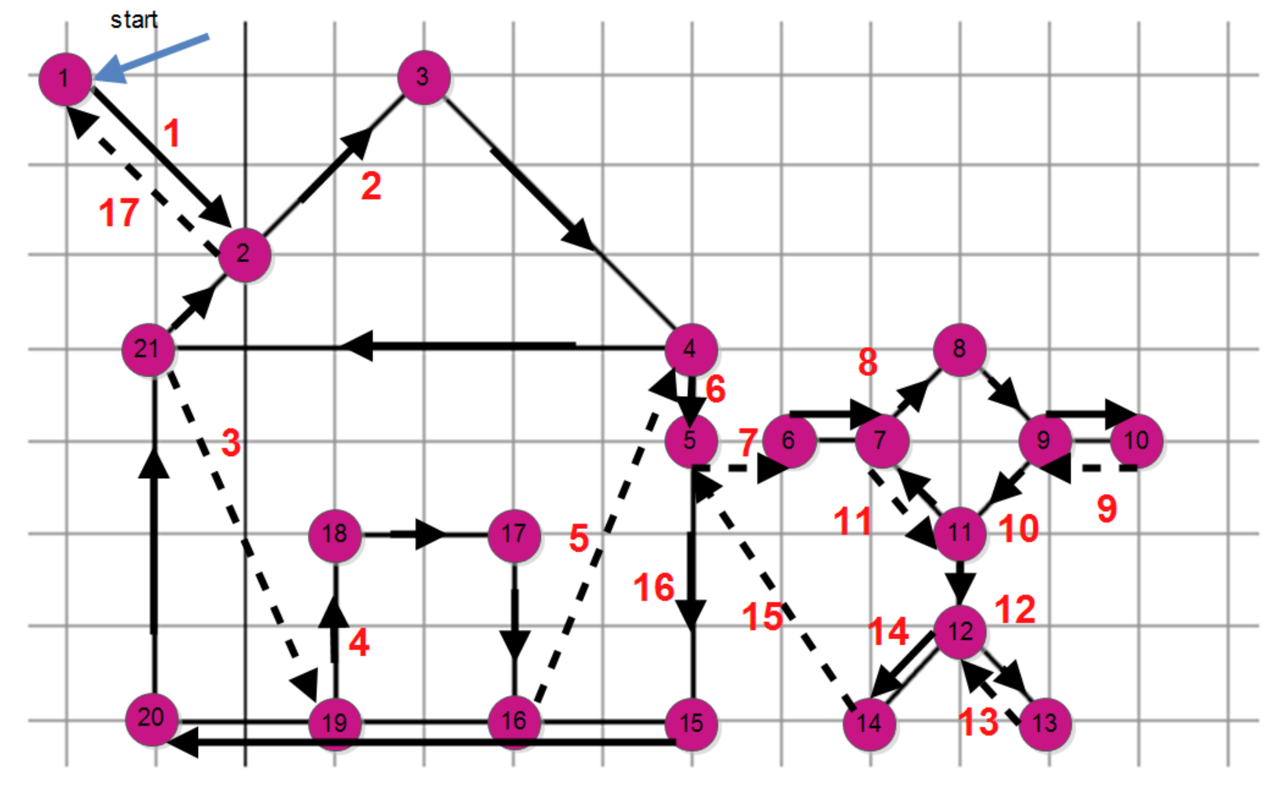

Figure 1.3.1 Labeled vertices (purple) and a solution path black arrows (the red number indicates the step number, also listed in below's table). An arrow indicates a the plotter movement from a vertex to another. Dotted arrows indicate penUp() plotter movements.

Temporal path costs:

Step Number	Action	Transition	Temporal Cost [s]
1	draw	1 → 2	2*sqrt(2)
2	draw	2-→ 3→ 4→ 21	5*sqrt(2)+6
3	Up movement	21→ 19	2*sqrt(5)
4	draw	19→ 18→ 17→ 16	6
5	Up movement	16→ 4	2*sqrt(5)
6	draw	4 → 5	1
7	Up movement	5→ 6	1
8	draw	6→ 7→ 8→ 9→ 10	2*sqrt(2)+2
9	Up movement	10→ 9	1
10	draw	9→ 11→ 7	2*sqrt(2)
11	Up movement	7→ 11	sqrt(2)

12	draw	11→ 12→ 13	1+sqrt(2)
13	Up movement	13→ 12	sqrt(2)
14	draw	12→ 14	sqrt(2)
15	Up movement	14→ 5	sqrt(13)
16	draw	5→ 15→ 20→ 21→ 2	13+sqrt(2)
17	Up movement	2→ 1	2*sqrt(2)
		Total ∑	31+18*sqrt(2)+4*sqrt(5)+sqrt(13) = *69.010 seconds*

According to my solution it takes about *69.010* seconds in order to plot **figure 1.3**. Note that there is just an additional cost of (2+4*sqrt(5)+4*sqrt(2)+sqrt(13) = 20.210) *seconds* required (*penUp()* movements, indicated by the black dotted arrows in **figure 1.3.1**)

What makes this problem muchg more difficult than the previous two examples?
>> Figure 1.3 consists of two unconnected fugures. Furthermore, the starting position of the plotter is not on a figure initially. When we were plotting **figure 1.1** and **figure 1.2** the plotter basically could stay on the same figure. There were some pinUp() operations necessary here and there but that's it to find the best Euler-Tour. This is completely different for plotting **figure 1.3**. The optimal path could be plotting partially the house-subfigure and then partially the right subfigure and so on. Greedely plotting one figure after the other would not lead to the best path. Therefore the path-solution-space is much bigger than for plotting **figure 1.**1 and **figure 1.2** (similar to the helocopter platform landing path-problem as introduced in the first class.

Task 2:

Graph **G** = (**V**, **E**) is the set of its vertices **V** and edges **E**.

A vertex **v** is a container structure consisting of a *position* and an identifier called *index*.

An edge **e_ij** = (**v_i**, **v_j**) is represents a connection between two vertices **v_i**, **v_j.** Thus, the set of edges **E** contains all connections between the vertices in **V** of the given Graph **G.**

NB: If two vertices **v** and **w** are connected with each other, i.e. **e_vw** = (**v**, **w**), then **vw** denotes the edge pointing from **v** to **w**.

A special type of graph is the so called *complete graph **G_c** = (**V_c**, **E_c**)*. In this graph, every vertex in **V_c** is connected with any other vertex in **V_c**. Therefore, for a complete graph the following holds true:

If **G_c** has |**V_c**| = n vertices, every such vertex is connected with (n-1) other vertices.

Notation conventions:

- **L_v** denotes the set of arcs leaving **v**, i.e. It represents the set of all edges pointing from a vertex **v** to its neighbor vertices.
- |**L_v**| denotes the number of neighbor vertex of **v**.
- **p_w** denotes an associated value for the vertex **w**, i.e. The label of the vertex.

Observations:

- Initially, the p-value for every vertex vertex v in V is set to -1, i.e. Forall v in V: p_v = -1 i.e. It is is labeled by -1.
- The most outer loop runs until Q is the empty set (i.e. Q is empty).
- The conditional in the most inner loop holds true if and only if the p value of a considered vertex w is equal to -1, i.e. p_w is equal -1.
- Every most outer iteration selects the front most element in Q and iterates over its neighbor vertices. If a neighbor is labeled by -1 we append it to Q and overwrite its label by the index of the current vertex in iteration. We visit each vertex at most once, since we append a vertex to Q only if its label is equal to -1 and we relabel such a vertex' label after appending it. We visit each vertex at least once (assuming each vertex is at least with any other vertex in the graph connected), since we visit the whole neighborhood of every vertex in Q and append such a vertex IF it has a label equal -1. We apply this procedure to each element in Q. Therfore, since we visit each vertex at least and at most once, we visit each vertex in G once.
- In the most inner loop, ve visit each neighbor of every element in Q.

This **concludes** to the following: the complexity of this algorithm is in general in

O(∑_{v in V} |L_v|) which is the *sum of the number of neighbors |**L_v**| of every vertex **v** in V.*

For a complete graph with *n* vertices this is in **O**(n(n-1) since every vertex has *(n-1)* connection. Note that **O**(n(n-1)) is the same as **O**(n^2). So this algorithm has a quadratic complexity in terms of the vertex count (for the complete graph).