

Problem-Sheet 2

My Solution

Michael Single

08-917-445

msingle@students.unibe.ch

Task 1:

Def(polyhedron): „a solid in three dimensions with flat faces, straight edges and sharp vertices.“ - From Wikipedia(See <http://en.wikipedia.org/wiki/Polyhedron>)

Def(Face): A triangle spanned by a triple of vertices, i.e. A complete graph K_3 .

Def(Platonic Solid): A regular, convex polyhedron with congruent faces of regular polygons with the same number of faces meeting at each of its vertices.

Claim: There exist only five platonic solids.

Proof(by contradiction):


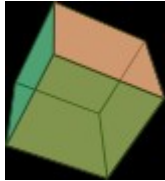



We either assume that:

- a) there are *less than five* platonic solids or
- b) there are *more than five* platonic solids.

If we can prove that neither **a)** nor **b)** holds true, we have shown that there exist exactly five solids. This kind of proof is sufficient **IFF** we can further show that platonic solids exist.

Thus, let us start by assuming **a)**

Since we know that there exist at least five solids (see **Table 1.1**) – namely *Tetrahedron*, *Hexahedron*, *Octahedron*, *Dodecahedron*, *Icosahedron* – we know, that platonic solids exist. Further we also know that assumption **a)** is a contradiction and thus cannot be true.

				
Tetrahedron	Hexahedron	Octahedron	Dodecahedron	Icosahedron
Table 1.1: The five platonic solids. (Images taken from http://en.wikipedia.org/wiki/Platonic_solid)				

Next, let us assume **b)**

Next, let us state some facts which will help us formulating the proof:

The following two statements hold true for any kind of solid:

1. At each of its vertices at least three faces meet.
2. The sum of all internal angles that meet at such a vertex must be smaller than 360 degrees.

Statement 1 must hold true since otherwise the solid would not be waterproof i.e. It would exhibit holes if there were vertices with fewer than 3 meeting faces.

Statement 2 must hold true since otherwise the shape would be flatten out.

From its definition we know that Platonic solid's faceses are all identical regular polygons.

For any *regular polygon* consisting of **n** vertices (let us denote such a polygon **n-gon**), the internal angle between two meeting edges (of such a n-gon) is equal to:

$$a(n) = ((n-2)*180^\circ) / n$$

whereat (n-2) is the number of internal, planar triangles of a n-gon.

Let us formulate a table which lists the total internal angles sum of regular **n**-gons for varying numbers of faces **m** meeting at any of its vertices. **Statement 1** tells us that we only have to consider m-values greater or equal to 3.

		m := Number of faces meeting at any vertex of the n-gon				
		3	4	5	6	...
n := Number of vertices of a considered regular n-gon	3	<u>180</u>	<u>240</u>	<u>300</u>	360	...
	4	<u>270</u>	360	450	540	...
	5	<u>324</u>	432	540	648	...
	6	360	480	600	720	...

Table 1.2: Total internal angle sum of a regular n-gon having m meeting faces at any vertex.						

A value of one particular cell (in yellow) in table **Table 1.2** is equal to **m*a(n)**

and indicates the sum of all internal angles that meet at any vertex of a considered n-gon.

Considering the table **Table 1.2** we can conclude that there exist exactly (and thus not more than) five solids fulfilling **Statement 2** of a solid (these are marked bold, underlined and in blue in table **Table 1.2**).

Hence, assumption **b)** is a contradiction.

Therefore, since we have shown that neither a) nor b) can hold true – i.e. By assuming either case we would end up with a contradiction – there must exist exactly five platonic solids. Q.E.D.

Task 2:

The source-code of **BFS** algorithm and used **Graph-Datastructures** can be found at the Appendix. My code has been implemented in Ruby and can be found at: <https://github.com/simplay/combo>

Currently, the starting position of my implementation is hard-coded – starting at vertex 1. I.e. A in the given example graph from this assignment.

Furthermore, instead of using letters im using intergers as vertex labels. Thus the graph from this example becomes (output of my implementation):

10 vertices:

1
2
3
4
5
6
7
8
9
10

15 edges:

1->2
1->4
2->3
2->6
3->1
3->4
3->5
4->5
6->3
7->6
7->8
8->6
8->10
9->8
10->9

Running the traversal algorithms will output the following solution:

starting position: 1

traversed path (DFS)

1 4 5 2 6 3

traversed path (BFS)

1 2 4 3 6 5

Please have a look at my repository (see link above) since I plan to allow users to specify the starting position. For defining a graph I have defined a simple data-format. Such a graph file is located in „data/“. The format's description can be found on my repository.

a) Basically we only have to change the datastructure Q from being a Queue to be a Stack in order to run a **DFS** instead of a **BFS**. Further, just for convenience purposes we rename Q to S (S like „Stack“).

Remember: **BFS** uses a **Queue**, **DFS** uses a **Stack**

Additionally, we can modify our traversal algorithm like the following:

```
Forall v in V set p_v = -1
S.push(r)
while(S notEmpty)
    Delete v from the front of S
    if p_v == -1
        Set p_v = v
        For vw in L_v
            S.push(w)
```

Algorithm 1: DFS Algorithm

Basically we have swapped the if statement with the for loop over L_v in **Algorithm 1**. This can be a performance boost for some cases. Since we will not iterate over the whole neighborhood of v if v was already visited. Keep in mind that this modification is not necessary required in order to form a **DFS** traversal from the previous **BFS** version.

b)

Without the suggested modification as formulated in **Algorithm 1** the memory complexity would be equal to $O(|V|+|E|)$. In general a **BFS** and **DFS** traversal algorithm have a **runtime-** and **memory complexity** of $O(|V|+|E|)$ (without applying further optimizations). When remembering which vertices in the graph already have been visited, then it is actually possible to reduce the memory complexity to $O(|V|)$ in both approaches. In my implementation the DFS and BFS traversal algorithms have a runtime- and memory complexity of $O(|V|+|E|)$.

Note that $O(|E|)$ can vary between $O(1)$ (no edges between vertices) and $O(|V|^2)$ (complete graph).

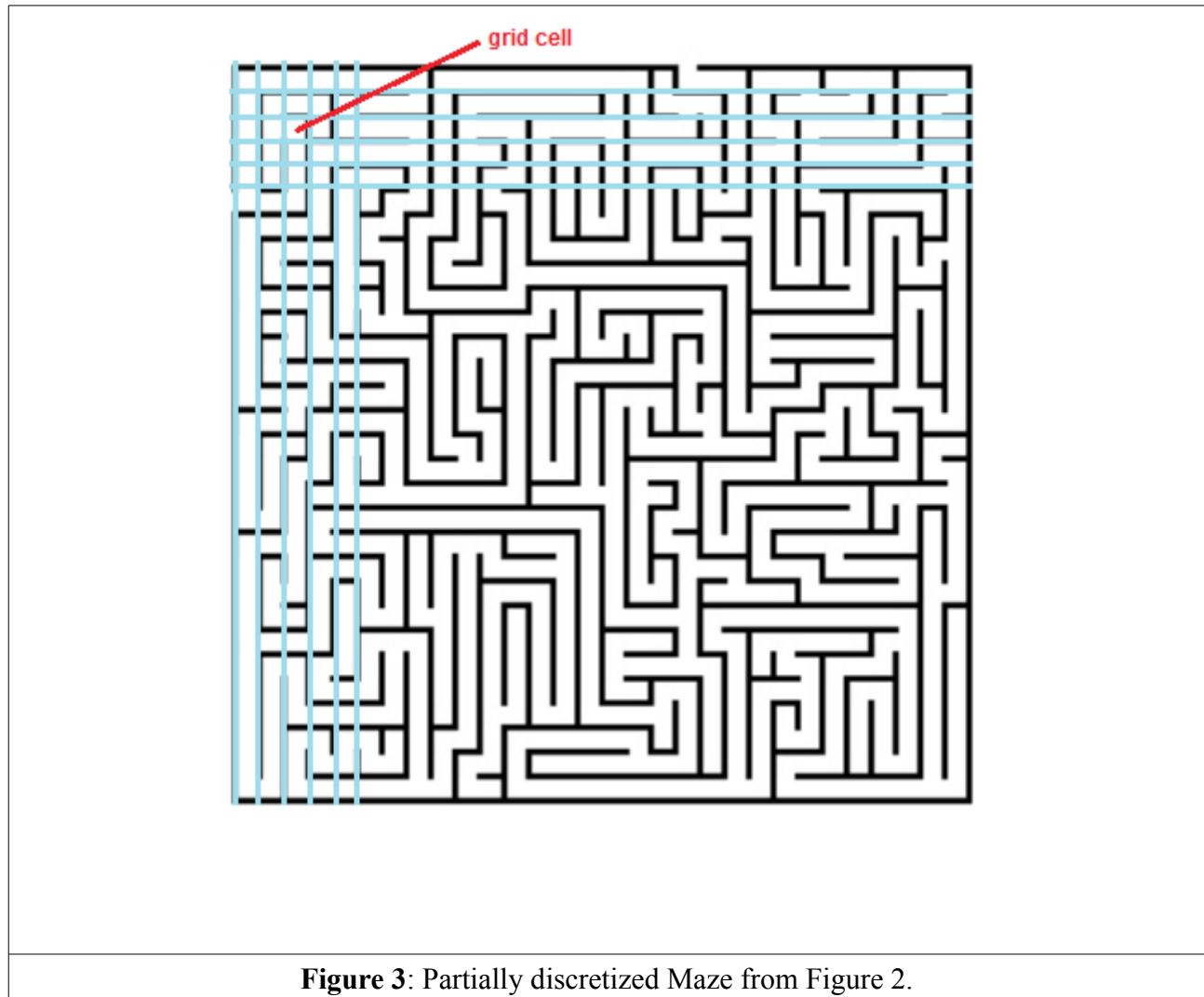
c)

In general a maze illustrated as in **Figure 2**, can be discretized by a grid of equidistant cells. **Figure 3** shows exemplarily a partial discretization (light blue lines) of the maze from **Figure 2**. One particular grid cell of this discretization scheme is indicated by a red pointing line. Note that we will have to discretize the whole maze and not just some part as shown in **Figure 3**. As a matter of simplification – to keep things understandable and simple – I only have drawn a fraction of the grid into Figure 3.

In a naive approach each grid cell represents one vertex in our graph. Then, an edge between two vertices indicates that there is a direct connection between the two considered cells (i.e. There is no blocking wall between the two cells / we can walk from the first cell to the other cell). Thus, if there is a direct connection between two cells, we connect the two corresponding vertices by an edge in the graph. We repeat this procedure for all vertices/grid cells. We can model it as an undirected graph. Just mark every visited vertex in order to find out of the maze.

Then, we simply can apply the BFS algorithm and **stop IF we have found the exit** (a special labeled vertex). This forces us to apply to introduce a stopping conditional in the BFS algorithm – *hasExitFound?*. Note that we have to keep track of the already visited vertices in the maze in order to find a path through the maze (i.e. backtrack).

Notice: The BFS algorithm first considers all paths through the graph of length one, then length two and so on until it reaches a „leaf“ vertex (an end, no outgoing direction from from this vertex).



d)

In general it depends which traversal approach we should favor. If the solution is deep from the starting vertex then a BFS traversal will be beneficial. However, if the solution-vertex is close to the starting position (i.e. The pathlength from start-node to target-node is relatively small) then a **DFS** may behave better. For this case a DFS algorithm would traverse the graph a path in the graph all the way down until a „leaf“ or dead-end has been reached. It only will continue if the leaf is not equal to the target-vertex and hence would proceed with another neighbor. In short: **DFS** favors child-vertices whereat **BFS** favors the neighborhood vertices. For the given maze from **Figure 2** problem, we usually are better off using a **DFS** traversal approach.

The reason for using a DFS traversal approach:

The starting vertex in a maze game is usually rather „far away“ from the maze's exit. Simply

speaking, this means that the length of the path from the start to the exit position is rather long, since this maximized (subjective impression) the fun of a maze-solve-player (i.e. Solving the maze becomes harder, assuming there are many other possible paths among the solution path).

Appendix:

The main traversal logic can be found in the file **graph.rb** within my repository:

<https://github.com/simplay/combo>

The graph.rb file: At Commit **4a7dbe809a6ff49c1b6f6f38dde811fb36a7640** (versionized)

```
require_relative 'vertex.rb'
require_relative 'edge.rb'
require_relative 'my_stack.rb'
require_relative 'my_queue.rb'

require 'pry'

class Graph

  attr_reader :vertices, :edges, :is_directed

  def initialize(file_name)
    @vertices = Set.new
    @edges = Set.new
    load_file(file_name)
    to_s
  end

  def dfs_traversal_at(r)
    stack = MyStack.new
    stack.push(r)
    path = []
    while(!stack.empty?)
      vertex = stack.remove
      if vertex.p == -1
        vertex.p = 1337
        vertex.neighbor_vertices.each do |neighbor|
          stack.push(neighbor)
        end
        path << vertex
      end
    end
    traversal_method = "DFS"

    puts "traversed path (#{traversal_method})"
    path.each {|v| print(v.to_s + ' ')}
    puts "\n"
  end
end
```

```

def bfs_traversal_at(r)
  queue = MyQueue.new
  r.p = 1337
  queue.push(r)
  path = []
  while(!queue.empty?)
    vertex = queue.remove
    vertex.neighbor_vertices.each do |neighbor|
      if neighbor.p == -1
        neighbor.p = vertex.p
        queue.push(neighbor)
      end
    end
    path << vertex
  end
  traversal_method = "BFS"
  puts "traversed path (#{traversal_method})"
  path.each {|v| print(v.to_s + ' ')}
  puts "\n"
end

# @param in_bfs
def demo_graph_traversals
  r = vertices.first
  puts "starting position: #{r.to_s}"
  dfs_traversal_at(r)
  unmark_vertices
  bfs_traversal_at(r)
end

# print components of graph
# i.e. its vertices and edges.
def to_s
  puts "#{@vertices.count} vertices:"
  @vertices.each {|v| puts v.to_s}
  puts
  puts "#{@edges.count} edges:"
  @edges.each {|v| puts v.to_s}
  puts
end

private

def unmark_vertices
  @vertices.each &:unmark
end

# load given file
def load_file(file_name)
  state = nil
  file = File.new(file_name, "r")
  while (line = file.gets)
    state = state_of(line) if (line =~ /#/)
    process(line, state)
  end
  file.close
end

```

```

# @param line String line of file
# @param state what type of line are we currently reading
def process(line, state)
  line_content = line.split(/\s/)
  if line_content.length > 1
    case state
      when 'm'
        @is_directed = line_content.last.eql?('1')
      when 'v'
        build_vertex(line_content)
      when 'e'
        build_edge(line_content)
      else
        puts "undefined state"
      end
    end
  end
end

# a vertex builder using data from current read line.
# @param line [String] currently read line
def build_vertex(line)
  vertex_keys = [:id, :p]
  vertex_arguments = {}
  line.each_with_index do |item, idx|
    vertex_arguments[vertex_keys[idx]] = item
  end
  @vertices.add(Vertex.new(vertex_arguments))
end

# a vertex builder using data from current read line.
# @param line [String] currently read line
def build_edge(line)

  vertices = [0,1].map do |idx|
    @vertices.select{|vertex| line[idx].eql?(vertex.id)}.first
  end

  vertices = sort_id_ascending_of(vertices) unless @is_directed
  @edges.add(Edge.new(vertices.first, vertices.last, @is_directed))
end

# get state from graph file
# telling us what kind of line we are dealing with
# in order to determine how we have to proceed this line.
# Note that white-spaces are chomped
# valid states:
#   'm' meta information
#   'v' vertex data
#   'e' edge data
# @param line [String] currently read line
def state_of(line)
  (line.split(/\s/).map &:chomp).last
end

```



```
# sort vertex ids ascending if we have an undirected graph
def sort_id_ascending_of(vertices)
  first_v = vertices.first
  last_v = vertices.last
  if first_v.id.to_i > last_v.id.to_i
    vertices[1] = first_v
    vertices[0] = last_v
  end
  vertices
end

end
```