Algorithms from THE BOOK

Chapter 3: Graph Theory

A Revision

Alexander Feil

SS2022

Contents

1 Introduction

The following document is a revision of the third chapter in the book "Algorithms from THE BOOK", written by Kenneth Lange [1]. It was composed for the seminar of the same name held by Prof. Dr. Oliver Junge at the Technichal University of Munich in the summer semester 2022. It will shortly refresh some basic definitions of graph theory and will then explain the algorithms presented in the book through an intuitive approach. The reader should be advised to have studied graph theory before reading the following. The definitions aren't of the upmost mathematical precision, but much rather serve as a reminder. Further, some programming knowledge is recommended. The used programming language will be Julia[1].

1.1 Refreshing Graph Knowledge

When I was younger, I always loved to play the nintendo© game "Professor Layton". It was a game full of puzzles you had to solve to advance in the storyline. Some of my favourite riddles were "use as few streetlights as possible to alight the city" or "how can Layton get through the forest as quickly as possible?". Now at universtiy, I have been introduced to the wonderous world of graph theory, which mathematically proves theorems about exactly those puzzles. But before we can prove theorems or even think about algorithms, we first have to define a few things[2]:

Definition 1.1 (Graphs, Digraphs)**.** *A **graph** is a system of nodes (also called vertices. think: points) and edges (think: lines connecting these points). A graph can be directed or not. However, we will only look at non-directed graphs here.*

Definition 1.2 (Edges, Neighbours)**.** *An **edge** is a 2-tuple, consisting of two vertices, e.g. $\{v_m, v_k\}$[3]. In the following, we will use the notation $e_{m,k} := \{v_m, v_k\}$.*

Definition 1.3 (Adjacency)**.** *We call two nodes **adjacent** or **neighbours**, if there is a edge connecting them. The number of a node's neighbours is called its **degree**.*

Definition 1.4 (Adjacency Matrix)**.** *We can keep track of adjacent nodes in a graph using the* adjacency matrix*. If v_m is adjacent to v_k, then $\texttt{A[m,k]} = 1$. If the edges are weighted, we define $\texttt{A[m,k]} := weight(e_{m,k}) =: w(e_{m,k})$*

Definition 1.5 (Paths)**.** *A path is a chain of nodes and edges, which can be written as $P{=}(v_1, e_{1,2}, v_2, ..., e_{n-1,n}, v_n)$, where no edge or node appears twice or more. Also note that the first and last entry are always vertices.*

[1]https://julialang.org/ . A programming language similar to MATLAB, developed at the Massachusetts Institute of Technology (MIT)

[2]These definitions are inspired from the book "Algorithms from THE BOOK" as well as the script to the lecture "Discrete Structures" held by Prof. Dr. Stefan Weltge at the TUM in the summer semester 2021 [2]

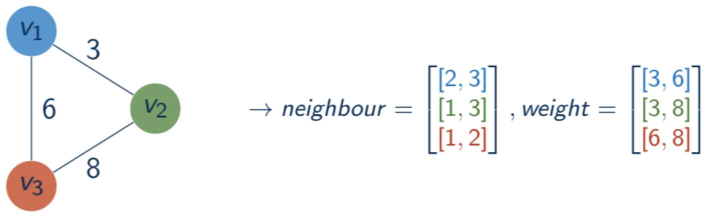
[3]The v comes from *vertex*, the singular form of *vertices*

Definition 1.6 (Components)**.** *A component is a set of nodes* $C := \{v_1, v_2, ..., v_n\}$*, where for every pair of nodes in* C *there is a path connecting them.*

Definition 1.7 (Tree)**.** *A tree is a graph without cycles.*

2 From Adjacency to Neighbourhood

With our first algorithm we want to extract neighbours of a node and the weights of the edges connecting them from the adjacency matrix A. The goal is to end up with two vectors, one for the neighbours and one for the corresponding weights. Here's an example:


v_1
v_2
v_3
3
6
8
$\rightarrow$ *neighbour* $= \begin{bmatrix}[2,3]\\ [1,3]\\ [1,2]\end{bmatrix}$, *weight* $= \begin{bmatrix}[3,6]\\ [3,8]\\ [6,8]\end{bmatrix}$

Graph 1

Notice how the entries in the vectors are vectors themselves, since a node can have a degree greater than 1. Therefore, one can get the degree of node v_m with `length`(`neighbour`[m]).

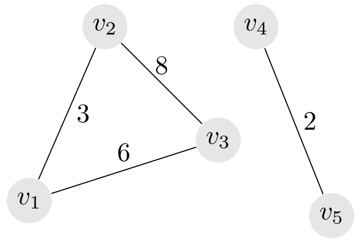
The Julia code to extract the *neighbour* and *weight* vectors from the adjacency matrix goes as follows:

```julia
 1 function AdjacencyToNeighbourhood(A::AbstractMatrix)
 2     (nodes,T) = (size(A,1), eltype(A))
 3     neighbour = [Vector{Int}() for m = 1:nodes]
 4     weight = [Vector{T}() for m=1:nodes]
 5     for m = 1:nodes
 6         for k = 1:nodes
 7             if A[m,k] != zero(T)
 8                 push!(neighbour[m], k)
 9                 push!(weight[m], A[m,k])
10             end
11         end
12     end
13     return (neighbour, weight)
14 end
```

The main idea of this code is the following: If an entry $a_{m,k} \neq 0$, there is a node connecting v_m to v_k. Therefore, the code only has to act, if $A[m,k]! = 0$ (see line 2). In this case, the algorithm pushes the number k (belonging to v_k) to the neighbour vector of v_m (line 9) and the weight of $e_{m,k}$ to the corresponding weight vector (line 10). The lines 3 to 5 are pure preparation. In lines 4 and 5 the neighbour and weight vectors are defined as vectors with vectors as entries.

2.1 Application of AdjacencyToNeighbourhood

For this application we will look at the following graph:



Graph 2

$$\text{The corresponding adjacency matrix is } A_1 := \begin{pmatrix} 0 & 3 & 6 & 0 & 0 \\ 3 & 0 & 8 & 0 & 0 \\ 6 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

In the GitHub repository[4] to this seminar you will find a jupyter notebook, where the "AdjacencyToNeighbourhood" algorithm and all other algorithms presented here are applied. The result of this code is:

```
[[2, 3], [1, 3], [1, 2], [5], [4]] #neighbour
[[3, 6], [3, 8], [6, 8], [2], [2]] #weight
```

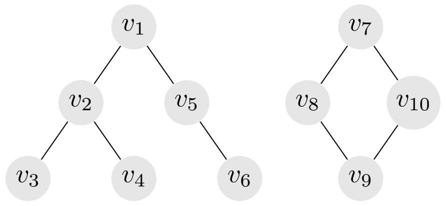
Looking at the first entries of the two vectors, one can see that v_1 is connected to v_2, v_3 and v_5 with the corresponding weights $3, 6$ and 3. This matches up with Graph 2.

[4]https://github.com/feilalexander/Seminar

3 Connected Components

In the introduction, we have spoken about components. Now, we want to develop an algorithm that will determine the number of components in a graph and find what component each node belongs to.

To explain the algorithm presented in "Algorithms from THE BOOK" we will look at the exemplary Graph 3:



Graph 3

Now, we want to find an algorithm that should return **the total number of components** and **a vector that lists the component for each node**. For Graph 3, that would be (**2**, **[1,1,1,1,1,1,2,2,2,2]**) (The first six nodes are in the first component, the last four in component 2). The only input for the algorithm should be the neighbour vector we saw in the Adjacency to Neighbourhood Algorithm.

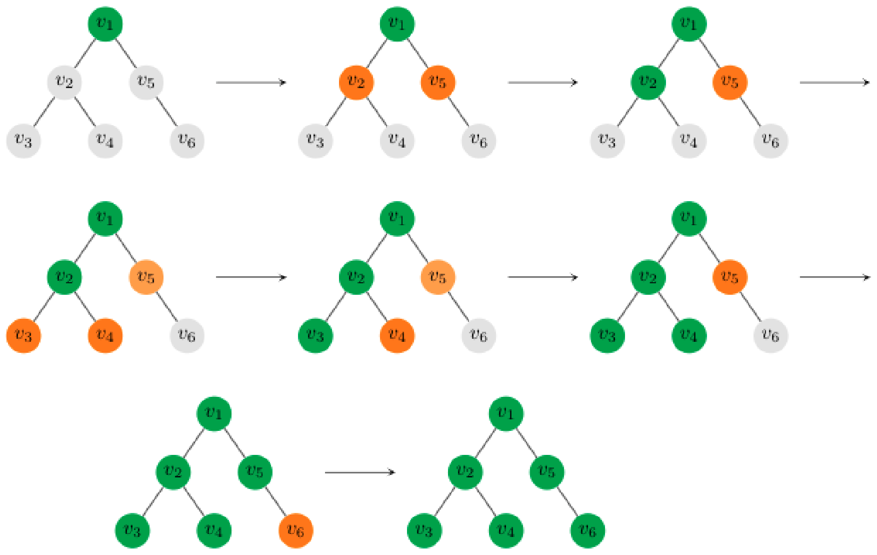
Before showing you the algorithm, I would like you to think about how you would implement such an algorithm. With the naked eye it is easy to see the components of course. But how would you teach a computer to find them? I left some space below for you to brainstorm. Feel free to use Graph 3 as a guide. Remember: you want to use the neighbour vector to find the components. [5]

[5] *Hint: Lange's Algorithm uses recursion*

During your brainstorm you might have found that two neighbours will always be in the same component. The algorithm uses exactly that property: Start with one node, find its neighbours. Two neighbours will always be in the same component. So we take these neighbours, find their neighbours and so forth, until we covered an entire component.

When thinking about this, we have to find a way to keep track of what nodes have been visited and which haven't. This is crucial, because otherwise we would never know when we're done. Hence, we define `component` $=$ `zeros(nodes)`. Now, when we visit node v_m, we set `component[m]` to the number of it's component. Hence, v_m hasn't been checked $\iff$ `component[m]` $=$ `0`. Now comes the tricky part. How do we make our way through a component?

Kenneth Lange chooses the following pattern: Take v_1, set `component[m]` $=$ `1` and find its neighbours. Take one of those neighbours (v_n) and set its entry to 1. Next you find a neighbour of v_n that hasn't been visited yet and repeat this process. If you have reached "the end of a branch" you go back to the last node with unchecked neighbours and continue there. In action, it would look like this:



Connected Components Algorithm in action. Green <> Visited, Orange <> To Check

This can be implemented with recursion. The recursive part presented in "Algorithms from THE BOOK" looks like this:

```julia
function visit!(neighbour::Array{Array {Int,1},1},
                component::Vector {Int}, i::{Int})
    for j in neighbour[i]
        if component[j] > 0 continue end #continue~false, end~true
        component[j] = component[i]
        visit!(neighbour, component, j)
    end
end
```

It basically goes through the neighbours of a node until it finds an unchecked one. Then it updates its component and runs the `visit!` recursion on that node. In a tree, this would mean that it follows one branch until the end before taking up a new one.

Now that we have understood this, we are ready for the entire code[6]:

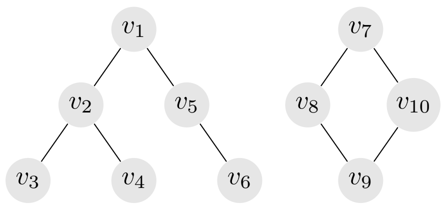
```julia
 1 function connect(neighbour::Array{Array{Int,1},1})
 2     nodes = length(neighbour)
 3     component = zeros(Int, nodes)
 4     componentcount = 0
 5     for i = 1:nodes
 6         if component[i] > 0 continue end #continue~false, end~true
 7         componentcount += 1
 8         component[i] = componentcount
 9         visit!(neighbour, component, i)
10     end
11     return (componentcount, component)
12 end
```

There are two main things to explain here. In line 4 we define the counter for the components. Once we have gone through one entire component, we add $+1$ in line 7 and go to the next component. You may think, that we could run into the problem of raising the componentcount by 1 before finishing a component. However this isn't the case, because the real work is happening in the visit! function. We are stuck in this recursion until an entire component has been visited. Then we go back to the for loop in line 5, which goes through the nodes until it finds one that hasn't been visited (i.e. is in a new component). Hence, `componentcount` always updates once we start to approach a new component.

[6]The original name for the variable "componentcount" used in the book is "components". I found this rather confusing with the variable component (no s!) and hence changed the name to a clearer term.

3.1 Application of Connected Components

For this application, we will once again look at Graph 3:



$$\text{with the adjacency Matrix } A_2 := \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

The result given by the algorithm you can find in the GitHub repository to this paper is $([\texttt{1}, \texttt{1}, \texttt{1}, \texttt{1}, \texttt{1}, \texttt{1}, \texttt{2}, \texttt{2}, \texttt{2}, \texttt{2}], \texttt{2})$. This checks out, since we have two components and all v_m with $m \in \{1, 2, 3, 4, 5, 6\}$ are in component 1 and all other nodes are in component 2.

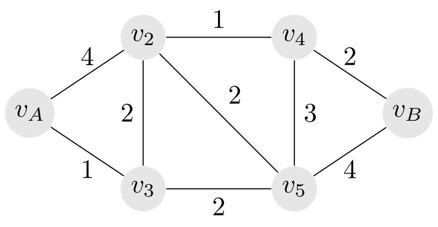
4 Dijkstra's Algorithm

In this chapter, we want to find an algorithm that will give us the shortest path from a given starting node v_A and an ending node v_B in a weighted graph G. This algorithm is significant in navigation and especially substantial for Professor Layton video games. Before getting to the algorithm, we first have to prove one proposition that will give us a fundamental argument in order to to understand the correctness of Dijkstra's Algorithm.

Proposition 4.1. *In a graph G, let there be a shortest path P from v_A to v_B. Then every subpath of this shortest path beginning at v_A is itself a shortest path.*

Proof. This can be shown through contradiction. Assume, there is a subpath from v_A to v_m, where v_m lies on the shortest path from v_A to v_B, which isn't the shortest path connecting v_A to v_m. Then there exists a shorter path connecting v_A to v_B (by rather using the shorter path to connect v_A to v_m), which contradicts the assumption that the original path was the shortest. □

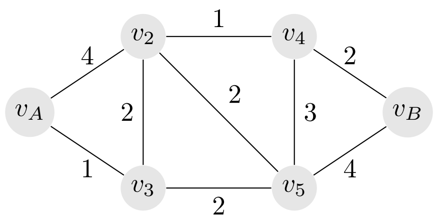
With this in mind we can now get to deriving the algorithm. But first, I would like you to brainstorm a bit. How would you go about teaching a computer how to find the shortest path in this graph?



Graph 4

One naive approach you might have chosen would be to start at v_A and always look for the next edge $e_{m,k}$ with the shortest weight $min\{w(e_{m,k}), e_{m,k} \in G\}$. However, this doesn't always work. Suppose $e_{A,2}$ in Graph 3 had weight 2. With our current method we would go $v_A - v_3 - v_2$. However, the direct route would be quicker! Therefore we rephrase our approach to "always look for the node with the next shortest path to v_A". Now, when we reach v_B, we found the shortest path. Such algorithms, that always look for the next best option, are called *greedy algorithms*.

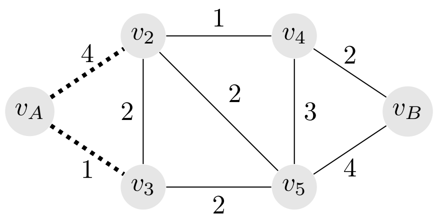
To implement this algorithm, we will need a way to track the candidates for the next shortest path. For this, the Julia package `DataStructures` offers the Priority Queue. It turns a zip array (a list of two-tuples) into a list that is ranked according to the second value in the tuple. Let's understand this by revising Graph 4:



In the very first step, we start with v_A (that has distance 0 to itself) and set the distance to all other nodes to infinity. Only once we create a connection from v_A to v_n we change it's distance. With this method, unreachable nodes will always end up with ∞. Hence, our first Priority Queue is

$$\mathtt{pq_0} = (\mathtt{v_A} => 0, \mathtt{v_2} => \infty, \mathtt{v_3} => \infty, \mathtt{v_4} => \infty, \mathtt{v_5} => \infty, \mathtt{v_B} => \infty)$$

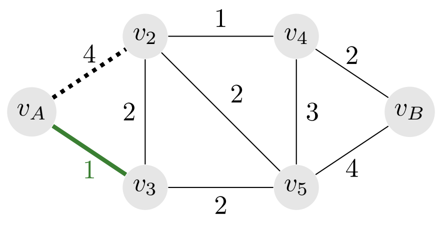
Next, we access the neighbour vector (See Graph 1) and find that v_2 and v_3 are neighbours of v_A



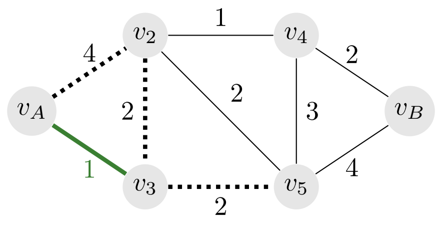
The weights of the connecting edges are smaller than ∞, so we update the Priority Queue to

$$\mathtt{pq_1} = (\mathtt{v_3} => 1, \mathtt{v_2} => 4, \mathtt{v_4} => \infty, \mathtt{v_5} => \infty, \mathtt{v_B} => \infty)$$

Notice how the Priority Queue automatically reorders itself according to the distance and v_A has been popped from the queue, since it's neighbours have been updated. Next, we look at the first candidate on the Priority Queue, which would be v_3. We note it's distance (as earler explained, we will not find a shorter path!) and mark it as visited.



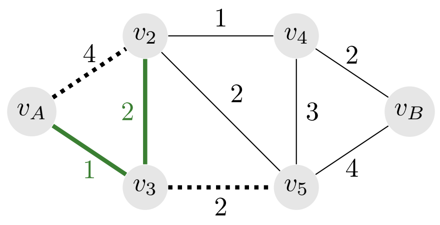
Following this, we look at the neighbours of v_3 and check, whether there might be a shorter path to a neighbour through v_3.



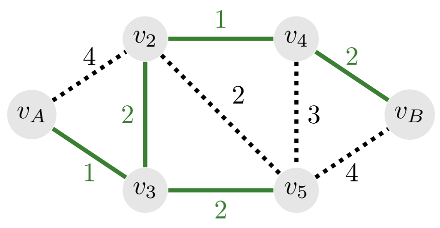
This would be the case for v_2, so we update it's minimum distance to 3. Additionally, v_5 now gets a smallest distance that isn't ∞, leaving us with

$$\mathtt{pq_2} = (\mathtt{v_2} => 3, \mathtt{v_5} => 3, \mathtt{v_4} => \infty, \mathtt{v_B} => \infty)$$

and (choosing v_2)



Following this pattern, Dijkstra's Algorithm returns



Now it's time to introduce Dijkstra's algorithm:

```
 1 using DataStructures
 2 function dijkstra(neighbour::Array{Array{Int,1},1},
 3                   weight::Array{Array{T,1},1}, source::Int),
 4                   where T<:Number
 5     nodes = length(neighbour)
 6     predecessor = zeros(Int, nodes)
 7     visited = falses(nodes)
 8     node = collect(1:nodes)
 9     distance = fill(Inf, nodes)
10     distance[source] = 0.0
11     pq = PriorityQueue(zip(node, distance))
12     while !isempty(pq)
13         (i,d) = peek(pq)
14         distance[i] = d
15         visited[i] = true
16         dequeue!(pq)
17         for k in 1:length(neighbour[i])
18             j = neighbour[i][k]
19             if !visited[j]
20                 dij = d + weight[i][k]
21                 if  dij < pq[j]
22                     predecessor[j] = i
23                     pq[j] = dij
24                 end
25             end
26         end
27     end
28     return (distance, predecessor)
29 end
```

The first few lines are very similar to our previous algorithms. In line 6 we define a vector, with which one can reconstruct a shortest path. For Graph 4, this would have been $[0, 2, 1, 2, 3, 4]$. In lines 9 and 10 we define the distance vector, which will store the length of the shortest path from each node to v_A [7]

In line 13 the `peek` function extracts the node on the first place in the priority list and thereby also its distance to v_A. As previously explained, this is the node's final shortest distance and can hence be saved.

In line 16 we go through the neighbours of this node and update their shortest distances if necessary. Note that it is crucial to use `for k in 1 : length`(...) instead of `for k in neighbour`[i], since we need the index k to access the right weight in line 19.

Lastly, to explain the if loop between lines 20-23: If the new path in G_m is shorter than the one that was possible in G_{m-1}, we need to note that. This is done here.

This being said, it remains to prove that this algorithm actually terminates with the correct answer. This will be proven in the Proposition on the next page:

[7] It should be mentioned that I have altered the original code in line 9 to make it shorter.

Proposition 4.2. *For a graph with m nodes, Dijkstra's algorithm terminates after m steps with the final distance.*

Before going about this proof, I have to introduce a new way of thinking about Dijkstra's Algorithm. In the code, we start with node v_A (established with Int) and add the node with the next shortest path and the corresponding edge in each step. You could regard this as starting with $G_0 := \{v_A\} \subseteq G$. In our previous example, we would have added v_3 and $e_{A,3}$ to make $G_1 := \{v_A, e_{A,3}, v_3\}$.

Proof. We will prove this through induction on the number of visited nodes k. Let G_k be the subgraph of G after k nodes have been visited

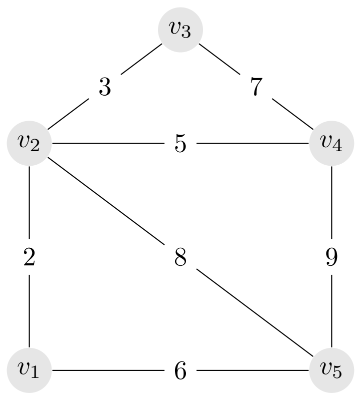
We want to show that all nodes in G_k have their final and correct distance to v_A noted in `distance`.

For the base case G_0, this is true by definition.

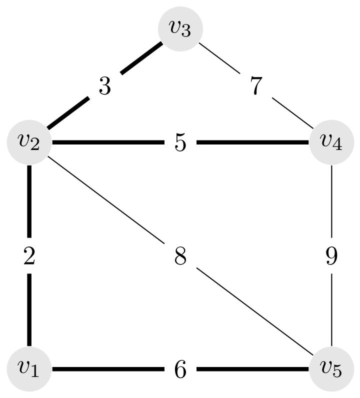
So let G_k be the subgraph after the k^{th} node has been visited. Our induction hypothesis is that we already know the final distance for all nodes in G_k. To get G_{k+1} we add the next node on the priority list. Due to the construction of the priority queue, this is v_k's final distance. $\square$

5 Prim's Algorithm

With Prim's Algorithm we want to find the minimum spanning tree (MST) of a graph. If you wonder what that is, let me explain it to you with the example presented by Kenneth Lange. Take a look at this graph. Can you find a subgraph $T_i \subseteq G$, where all nodes in G are also in T_i and $\sum_{e_{ij} \in T_i} w(e_{ij})$ is at the minimum?



You might have found that it would be clever to start with the smallest weight, which is w_{12}. Then you may have tried to find the next "lightest" edge and continue your path. Hopefully, you came to this solution:



This greedy way of working yourself through the graph is similar to Dijkstra's algorithm. In fact, Prim shares the credit for this algorithm with Dijkstra and the Czechoslovak mathematician Janík. Prim's Algorithm also works with the priority queue, but instead of using the shortest path to sort (as Dijkstra does) it prioritises the next shortest edge. The chain of steps is the following:

Step 1: start with node $v_i = v_1$

Step 2: find the neighbours of v_i and the weights of the edges connecting them to the node

Step 3: add these neighbours and the corresponding weights to the priority queue.

Step 4: Take the top entry *(think: the lightest edge)* of the Priority Queue

Step 5: Check, whether the start/end points of the edge, lets say v_m and v_l, are already part of the spanning Tree T_i. One of them will surely be. *(Exercise to you: Why?)*[8]

> **Step 5.1:** If the second, lets say v_l, isn't part of the tree yet, add the edge $e_{m,l}$ to the spanning tree T_i to make T_{i+1}. Now set $v_i = v_l$
>
> **Step 5.2:** If both nodes were part of the tree already, we cannot add the edge because it would make a cycle. So nothing happens

Step 7: Pop the edge off the queue. We won't need it any longer.

It is crucial to understand that we will go through every edge, even if it's incident nodes are already on the tree. This will make no harm, since we can disregard them with an if statement. A pseudo code would look like this:

[8]In order for the edge to be added to the Priority Queue, one of the incident nodes must have been visited.

```
function prims(neighbour, weight)
    i = 1
    while spanning tree < number of nodes
        if node i hasn’t been checked
            add its incident edges to the priority queue
            note that i has been checked
        end
        (k,l) = lightest edge
        if k hasn’t been checked
            add (k,l) to the spanning tree
            i = k
        elseif l hasn’t been chkecked
            add (k,l) to the spanning tree
            i = l
        end
        pop (k,l) off the priority queue
    end
    return the tree
end
```

Now I will present the complete code to you. Don’t be scared by the long preamble, they are simply necessary definitions. I will explain tricky lines after presenting the code. Without further a do, here it is:

```
function prim(neighbour::Array{Array{Int, 1}, 1},
              weight::Array{Array{T, 1}, 1} where T <: Number
    nodes = length(neighbour)
    visited = falses(nodes)
    (mst_nodes, i) = (1, 1)
    key = Array{Tuple{Int, Int}, 1}(undef, 0)
    priority = Array{Float64, 1}(undef, 0)
    pq = PriorityQueue(zip(key, priority))
    mst = Array{Tuple{Int, Int}, 1}(undef, 0) #The actual MST
    while mst_nodes < nodes
        if !visited[i]
            for k = 1:length(neighbour[i])
                j = neighbour[i][k]
                pq[(i,j)] = weight[i][k]
            end
            visited[i] = true
        end
        ((k,l), val) = peek(pq) #val is a spacetaker, no further use
        if !visited[k]
            push!(mst, (k,l))
            mst_nodes += 1
            i = k
        end
        if !visited[l]
            push!(mst, (k,l))
            mst_nodes += 1
            i = l
        end
        dequeue!(pq)
    end
    return mst
end
```

One main thing I did not understand on my fist glance at this algorithm were lines 6-9. However, they actually only define the priority queue and of what type it's entries should be.

The act of not regarding an edge whose incident nodes have been visited before happens in lines 18-30. If nothing happens in the last two if-enviromnents, then it will solely be popped off the priority queue in line 29.

Similarly to Dijkstra's algorithm, we will prove the correctness of Prim's algorithm and that it terminates after a finite amount of steps. We will do this in two steps and start with a helpful proposition:

Proposition 5.1. *Adding an edge outside of a spanning tree T to the tree makes a cycle. Deleting any edge from that cycle makes a new spanning tree.*

Proof. The first statement is rather clear. Assume, we add the edge $e_{m,k}$ connecting v_m and v_k. Since v_m and v_k are part of T, there exists one distinct path in T connecting the two. Adding e to P would make a cycle.

Now $T \cup e_{m,k}$ contains one cycle. Removing any edge e' from that cycle would destroy the cycle and hence $T \cup e_{m,k} \setminus e' =: T'$ would have no more cycles. By definition T' is a tree. □

Proposition 5.2. *For a weighted connected graph, Prim's Algorithm terminates after a finite number of steps with a minimal spanning tree.*

Proof. **Part 1** First we prove that the algorithm actually returns a tree T_n. We do this through induction. For T_1, only consisting of v_1, this is clear. Let T_k be a tree. The algorithm only adds an edge to the tree if either one of the incident nodes has not been visited yet (lines 20 and 25). In other words, is not part of tree T_k. The other node will have to be part of the tree in order for the edge to be regarded on the priority queue (line 14). Hence, the edge e_{k+1} we are about to add has one incident node inside of and one outside of the tree. Hence T_{k+1} is a tree as well.

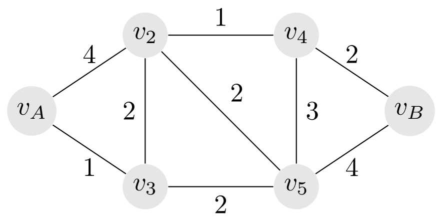
Part 2 To complete the proof, we will now show that T_n is minimal. To do so, we assume that T_n isn't minimal. Let S be an actual minimum tree that shares as many edges with T_n as possible. Since T_0 is a subgraph of S but T_n isn't, there must be a stage at which we add e_{k+1} to the tree T_k so that $T_{k+1} \subsetneq S$ for the first time.

Since S is a spanning tree of graph G and $e_{k+1} \notin S$, adding e_{k+1} to S would create a cycle (look at the previous example to convince yourself). Now, let e_{k+1} connect $v_k \in T_k$ to $v_{k+1} \in T_{k+1} \setminus T_k$. Since the cycle leaves the tree T_k with this edge, it also has to re-enter to complete the cycle by reaching v_k. Let this edge, where the cycle re-enters T_k (in other words, where one end is in T_k and the other is outside), be e'. Note that $e' \notin T_k$.

Since Prim's Algorithm added e_{k+1} to the tree instead of e', the priority queue implies that $w(e_{k+1}) \leq w(e')$. Because of Proposition 5.1a, taking e' out of S and adding e_{k+1} instead makes another spanning tree. This contradicts the assumption, that S was a MST that shared as many edges with T_n as possible. □

5.1 Application of Prim's Algorithm

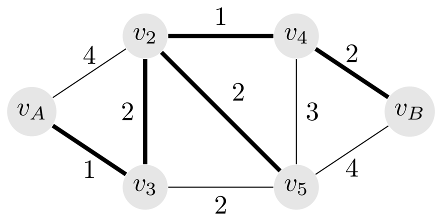
For this application we will run Prim's Algorithm on Graph 4:



Once again, you can find the complete code execution on the GitHub repository. It returns:

$$(\texttt{1},\texttt{3}),(\texttt{3},\texttt{2}),(\texttt{2},\texttt{4}),(\texttt{2},\texttt{5}),(\texttt{4},\texttt{6})$$

This corresponds to



In fact, it could have chosen $e_{3,5}$ instead of $e_{2,5}$. Then we would have received the exact same subgraph as we have seen in Dijkstra's Algorithm. You can take as the last exercise of this paper to think whether this is a coincidence or not. Is it provable that every MST is also a solution to Dijkstra's Algorithm?

References

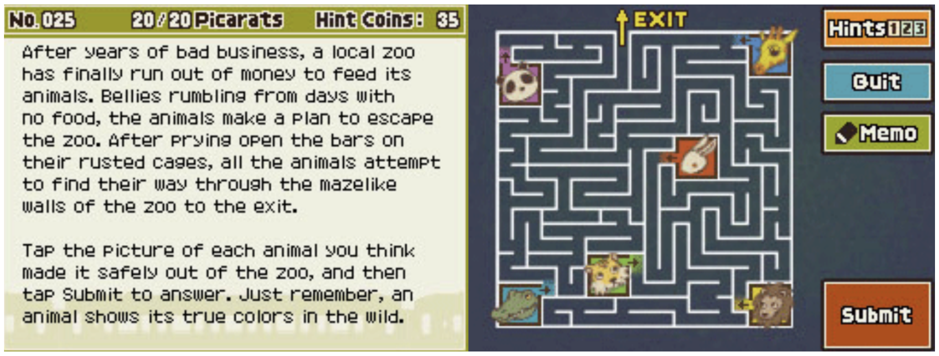
[1] Kenneth Lange. *Algorithms from THE BOOK.* SIAM, 2020.

[2] Prof. Dr. Stefan Weltge. *Diskrete Strukturen Skript.* 2021.

6 Two puzzles

Here are two riddles from the Professor Layton series[9] to test your knowledge. What algorithm did you use for each?

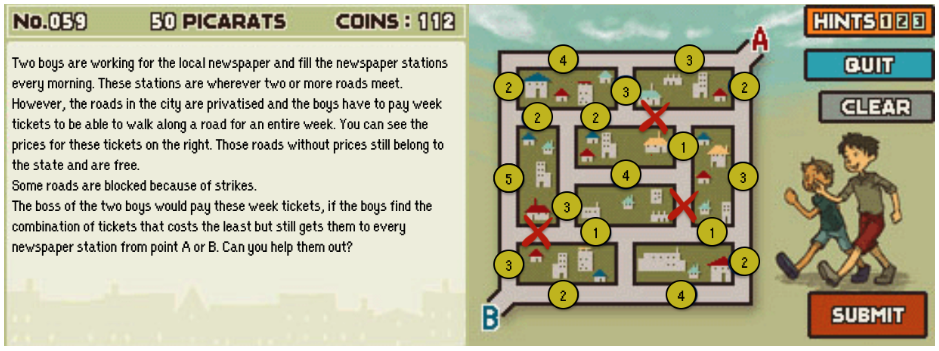
Diabolical Box Riddle 25: Surviving in the Wild



After years of bad business, a local zoo has finally run out of money to feed its animals. Bellies rumbling from days with no food, the animals make a plan to escape the zoo. After prying open the bars on their rusted cages, all the animals attempt to find their way through the mazelike walls of the zoo to the exit.

Tap the picture of each animal you think made it safely out of the zoo, and then tap Submit to answer. Just remember, an animal shows its true colors in the wild.

Curious Village Riddle 59: The longest path (altered)



Two boys are working for the local newspaper and fill the newspaper stations every morning. These stations are wherever two or more roads meet. However, the roads in the city are privatised and the boys have to pay week tickets to be able to walk along a road for an entire week. You can see the prices for these tickets on the right. Those roads without prices still belong to the state and are free.
Some roads are blocked because of strikes.
The boss of the two boys would pay these week tickets, if the boys find the combination of tickets that costs the least but still gets them to every newspaper station from point A or B. Can you help them out?

[9]https://www.laytonseries.com/naen/