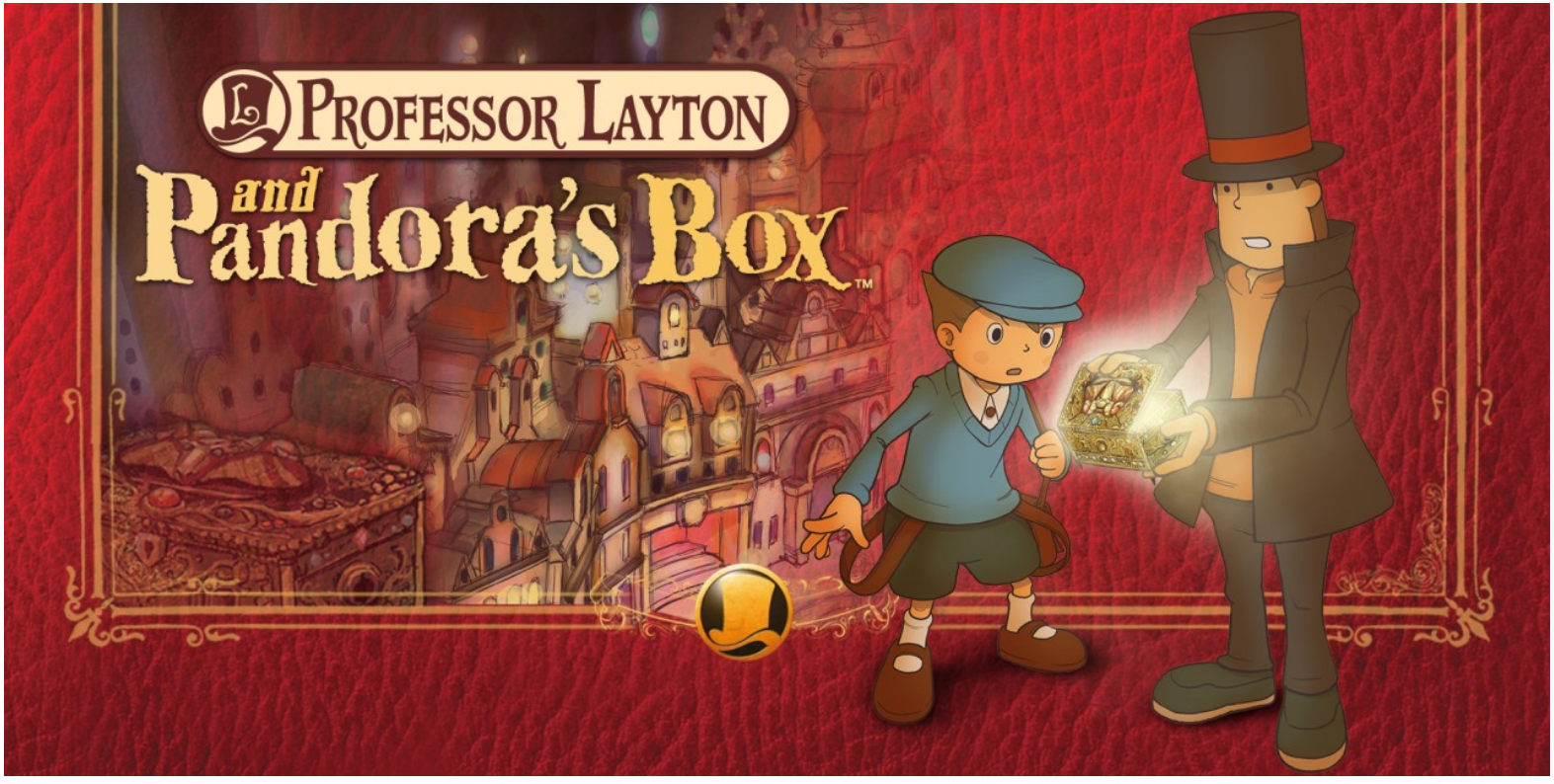
Algorithms from THE BOOK: Graphs

Alexander Feil

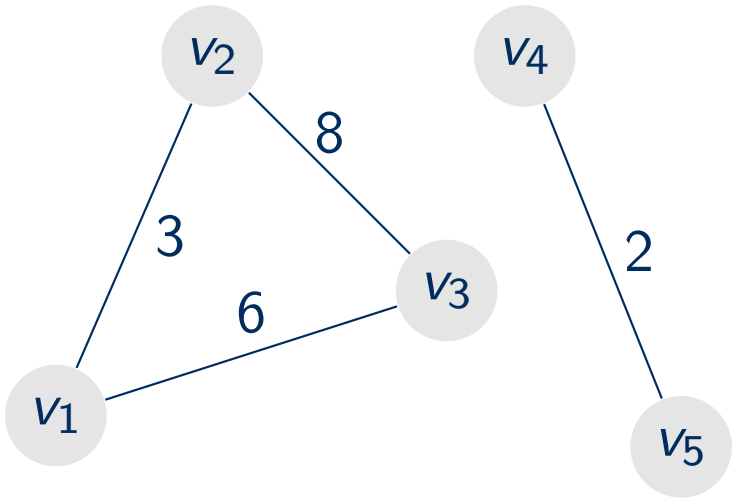
Technical University of Munich

Professor Layton

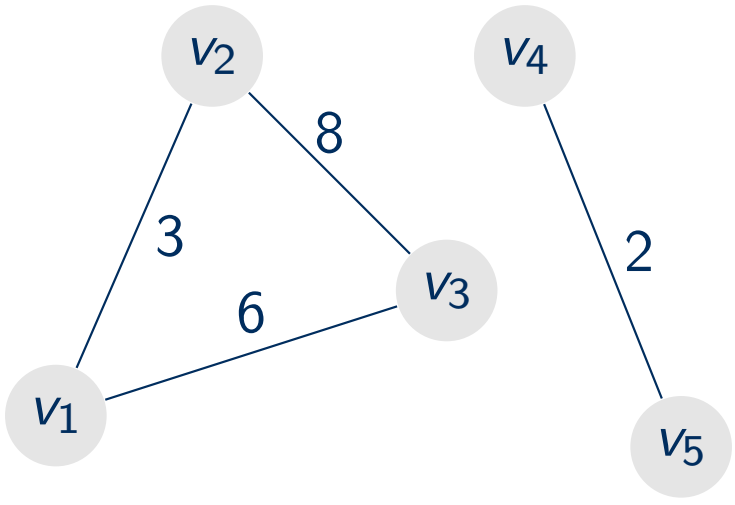


Professor Layton

Definitions

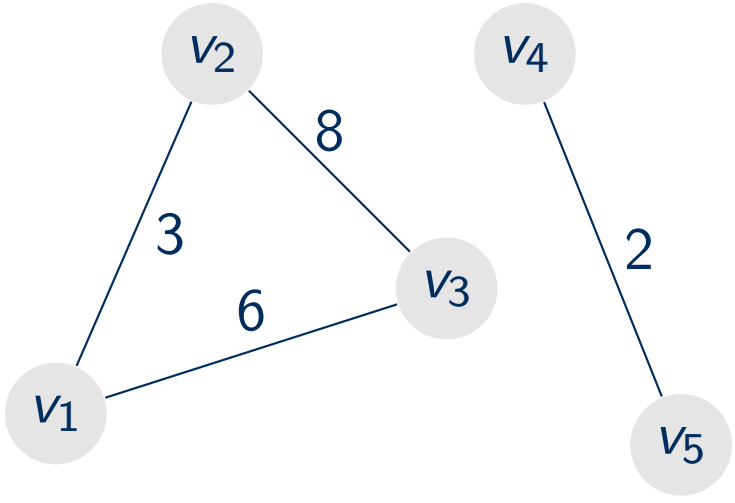


Definitions



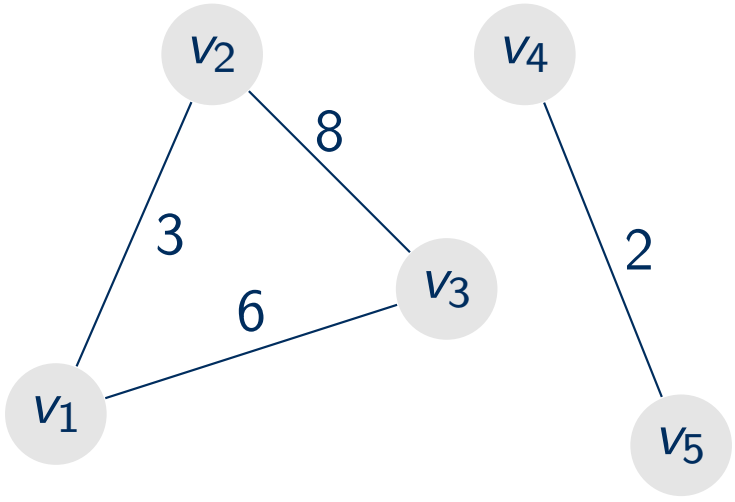
Node (Vertex): A point in the graph

Definitions



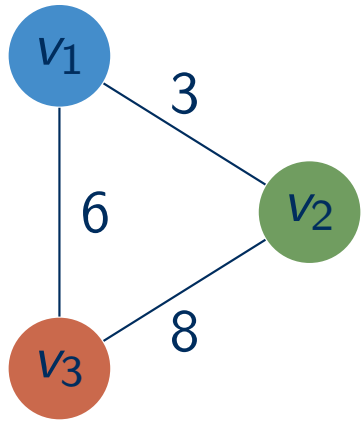
Node (Vertex): A point in the graph
Edges: Lines connecting vertices. $e_{m,k}$

Further Definitions



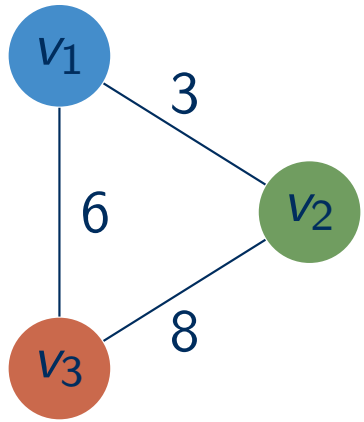
$$A_2 = \begin{pmatrix} 0 & 3 & 6 & 0 & 0 \\ 3 & 0 & 8 & 0 & 0 \\ 6 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

Adjacency to Neighbourhood



Graph 1

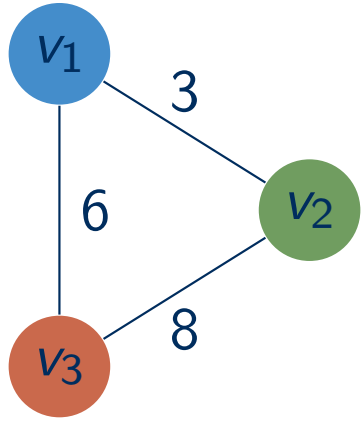
Adjacency to Neighbourhood



Graph 1

$$\rightarrow \textit{neighbour} = \begin{bmatrix} [2,3] \\ [1,3] \\ [1,2] \end{bmatrix},$$

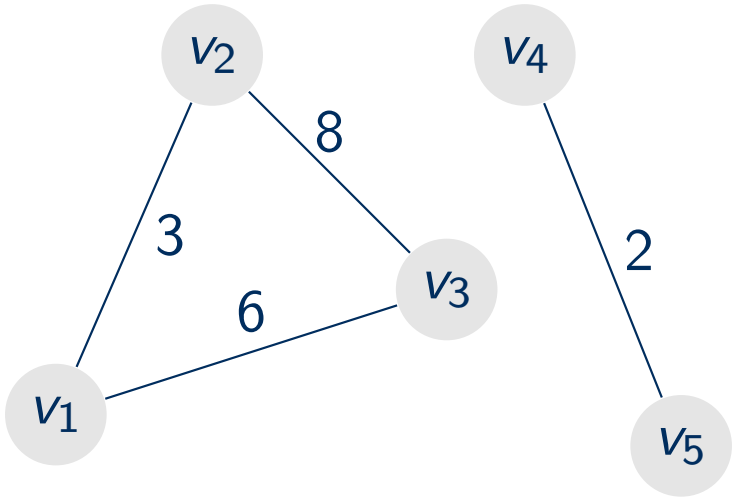
Adjacency to Neighbourhood



Graph 1

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

The basic idea



Graph 2

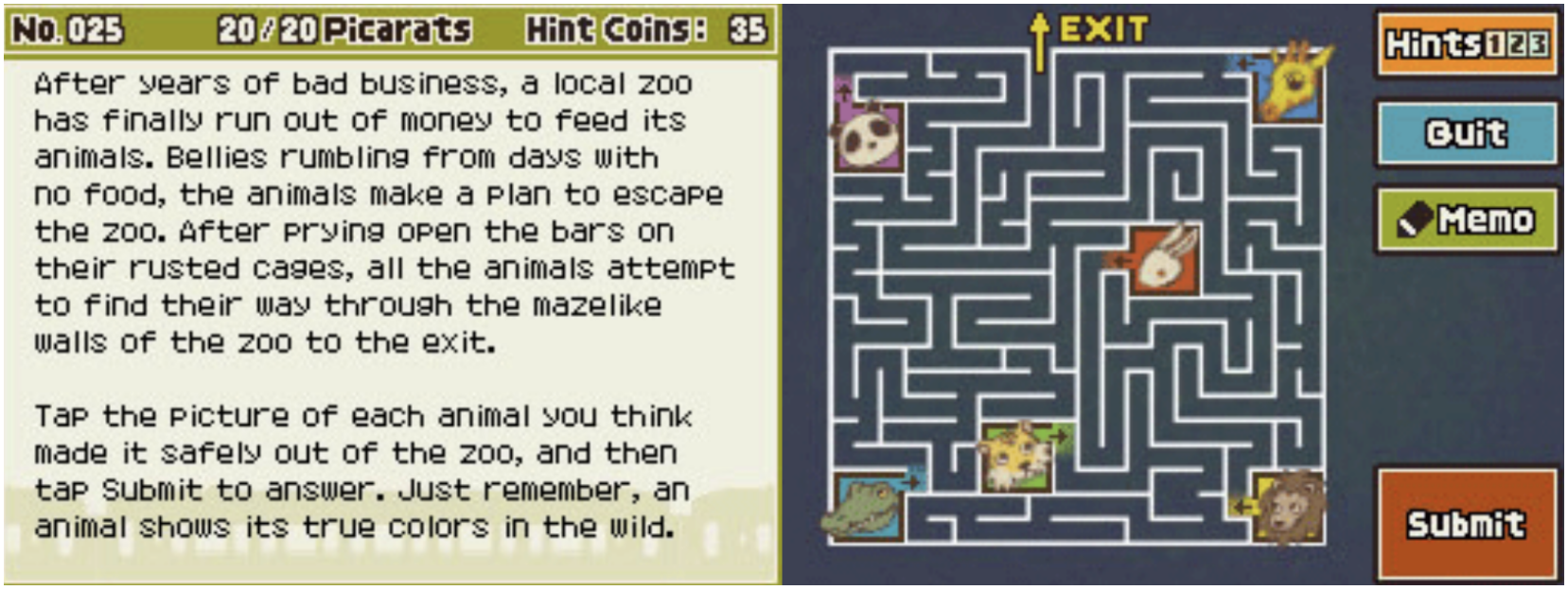
$$A_2 = \begin{pmatrix} 0 & 3 & 6 & 0 & 0 \\ 3 & 0 & 8 & 0 & 0 \\ 6 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

The Algorithm

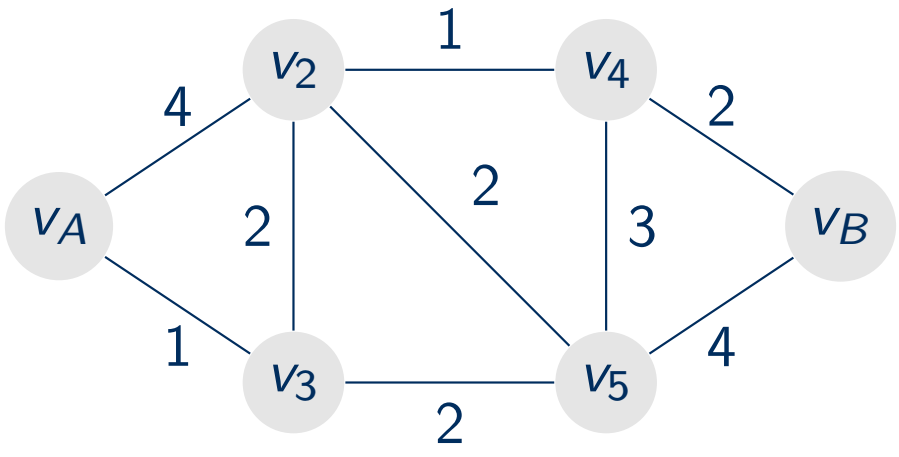
```
1  function AdjacencyToNeighbourhood(A::AbstractMatrix)
2     (nodes,T) = (size(A,1), eltype(A))
3      neighbour = [Vector{Int}() for m = 1:nodes]
4      weight = [Vector{T}() for m=1:nodes]
5      for m = 1:nodes
6          for k = 1:nodes
7              if A[m,k] != zero(T)
8                 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

Any Questions?

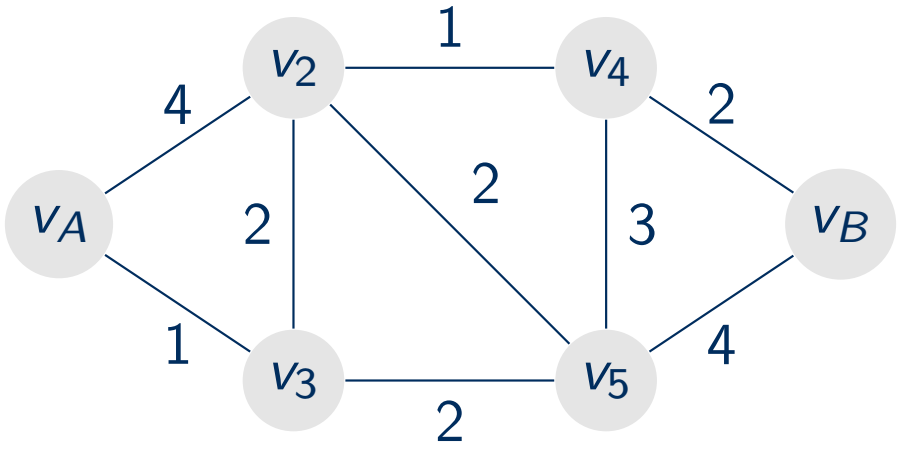
Layton Puzzle



Even Further Definitions

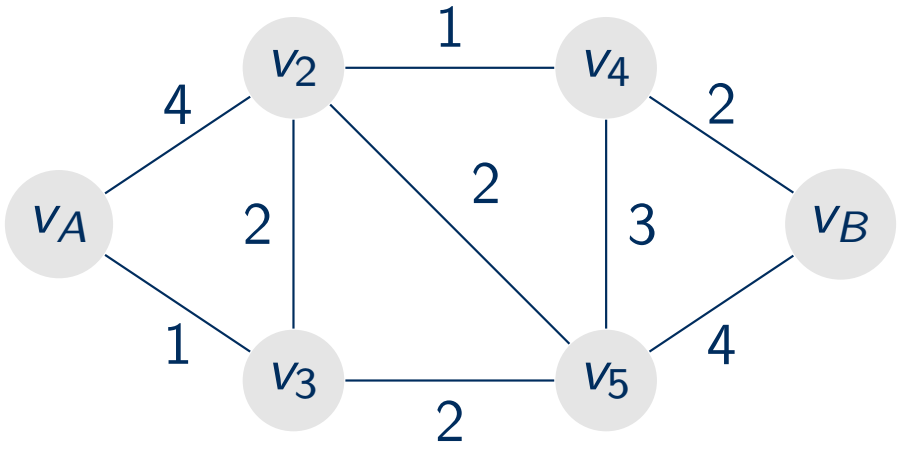


Even Further Definitions



Path: Chain of nodes and edges, no node appears twice

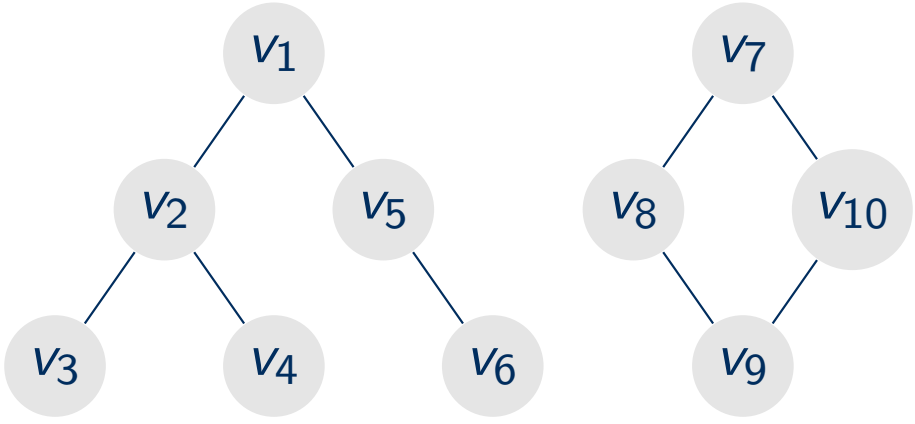
Even Further Definitions



Path: Chain of nodes and edges, no node appears twice

Components: Set of nodes with paths between each

Intuition



Graph 3

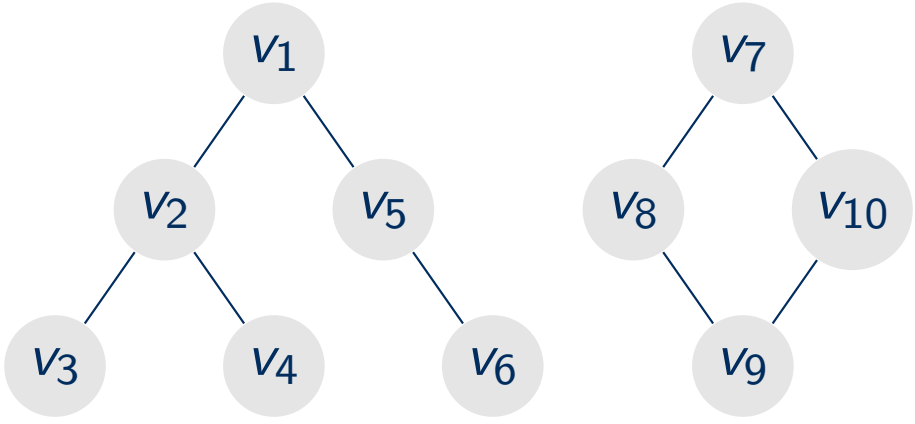
The Input

$$A_3 := \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

The Input

$$A_3 := \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \rightarrow \begin{bmatrix} [2,5] \\ [1,3,4] \\ [2] \\ [2] \\ [1,6] \\ [5] \\ [8,10] \\ [7,9] \\ [8,10] \\ [7,9] \end{bmatrix}$$

Intuition



Graph 3

function visit!

```
1 function visit!(neighbour::Array{Array {Int,1},1},
2                 component::Vector {Int}, i::{Int})
3     for j in neighbour[i]
4         if component[j] > 0 continue end #false true
5         component[j] = component[i]
6         visit!(neighbour, component, j)
7     end
8 end
```

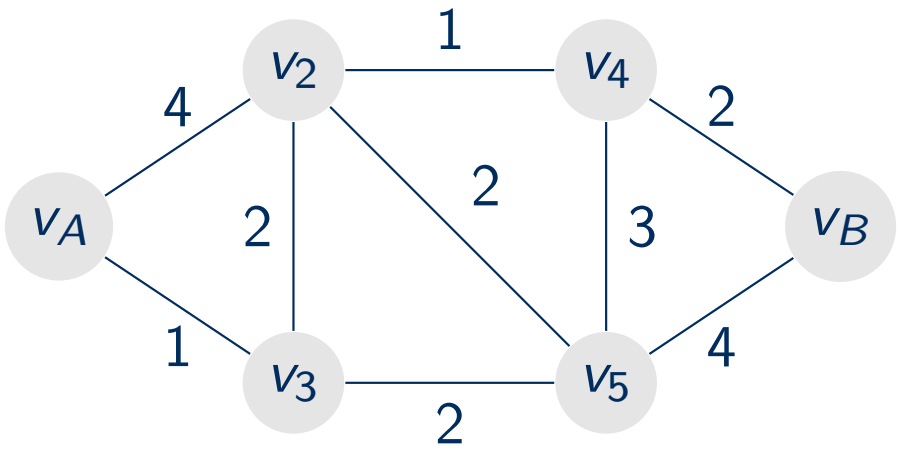
function connect

```
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 #false true
7          componentcount += 1
8          component[i] = componentcount
9          visit!(neighbour, component, i)
10     end
11     return (componentcount, component)
12 end
```

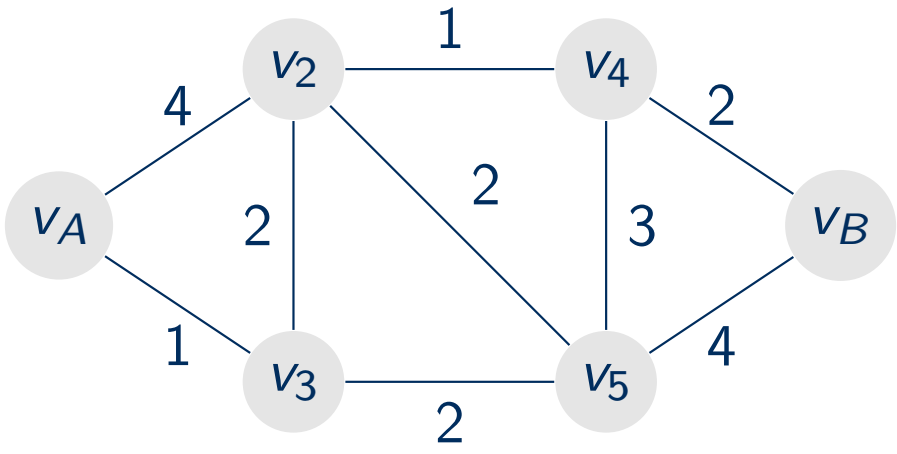
Any Questions?

A Layton Original

Preparation

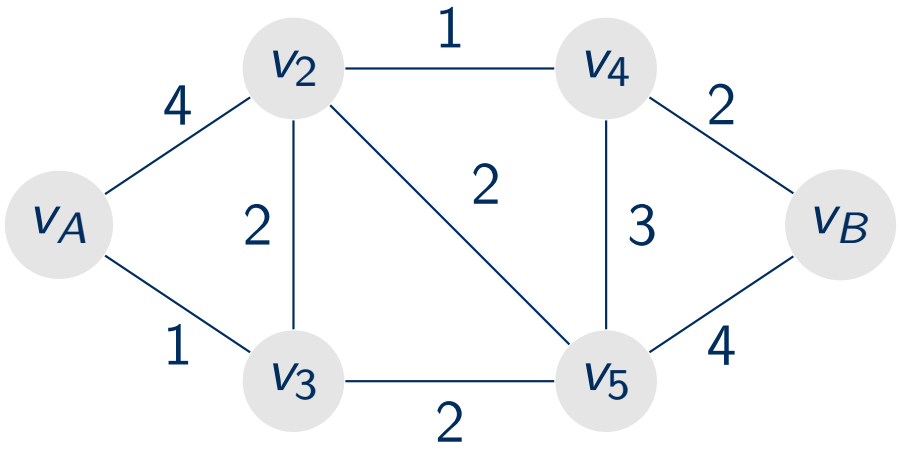


Preparation

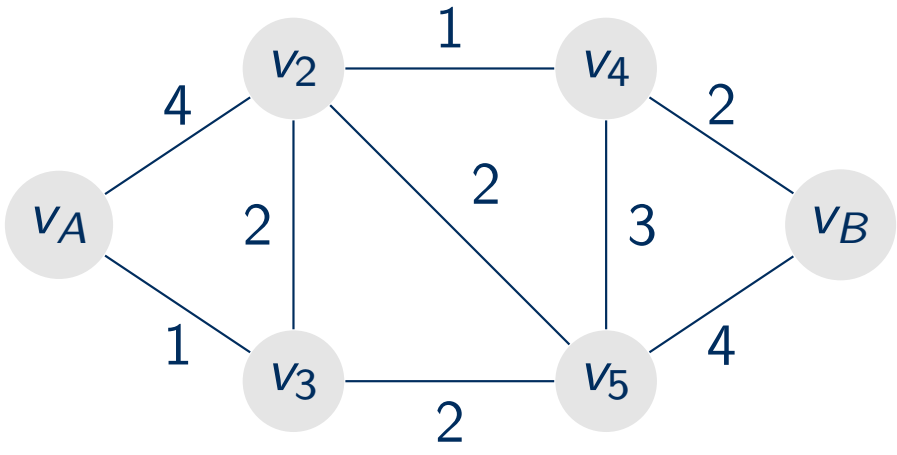


Proposition 4.1 *In a graph G, let there be a shortest path P from v_A to v_B. Then every subpath of this shortest path beginning at v_A is itself a shortest path.*

Intuition

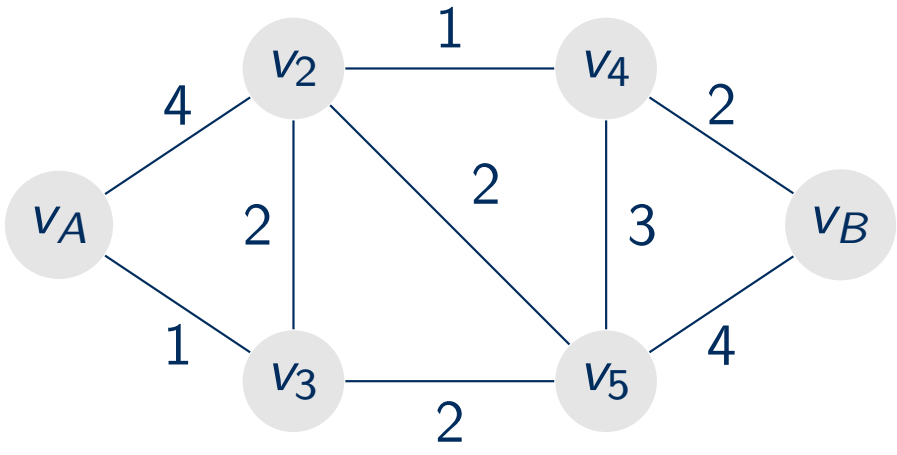


Intuition



$$\mathtt{pq}_0 = (\mathtt{v}_\mathtt{A} => 0, \mathtt{v}_2 => \infty, \mathtt{v}_3 => \infty, \mathtt{v}_4 => \infty, \mathtt{v}_5 => \infty, \mathtt{v}_\mathtt{B} => \infty)$$

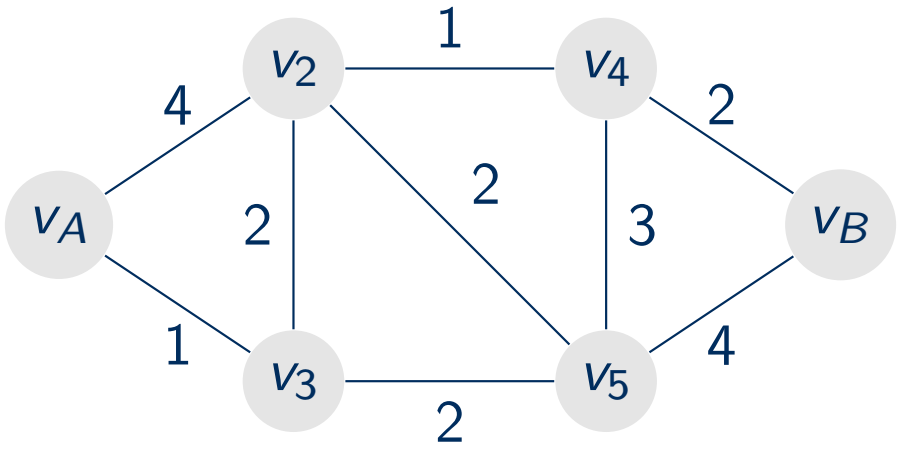
Intuition



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

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

Intuition



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

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

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

Dijkstra Pseudocode

```
1  function DijkstraPseudo(neighbour, weight, source)
2      pq = priority queue
3      while priority queue isn't empty
4          m =  first entry on pq
5          note distance of m
6          mark m as visited
7          pop m off pq
8          for not visited neighbours of m
9              if path through m is quicker
10                 update on pq
11                 note m as new predecessor
12             end
13         end
14     end
15     return distance, predecessor
16 end
```

Dijkstra's Algorithm (1)

```
1  using DataStructures
2  function dijkstra(neighbour::Array{Array{Int,1},1},
3                    weight::Array{Array{T,1},1},
4                    source::Int),
5                    where T<:Number
6      nodes = length(neighbour)
7      predecessor = zeros(Int, nodes)
8      visited = falses(nodes)
9      node = collect(1:nodes)
10     distance = fill(Inf, nodes)
11     distance[source] = 0.0
12     pq = PriorityQueue(zip(node, distance))
13
14     ...
15
16     return (distance, predecessor)
17 end
```

Dijkstra's Algorithm (2)

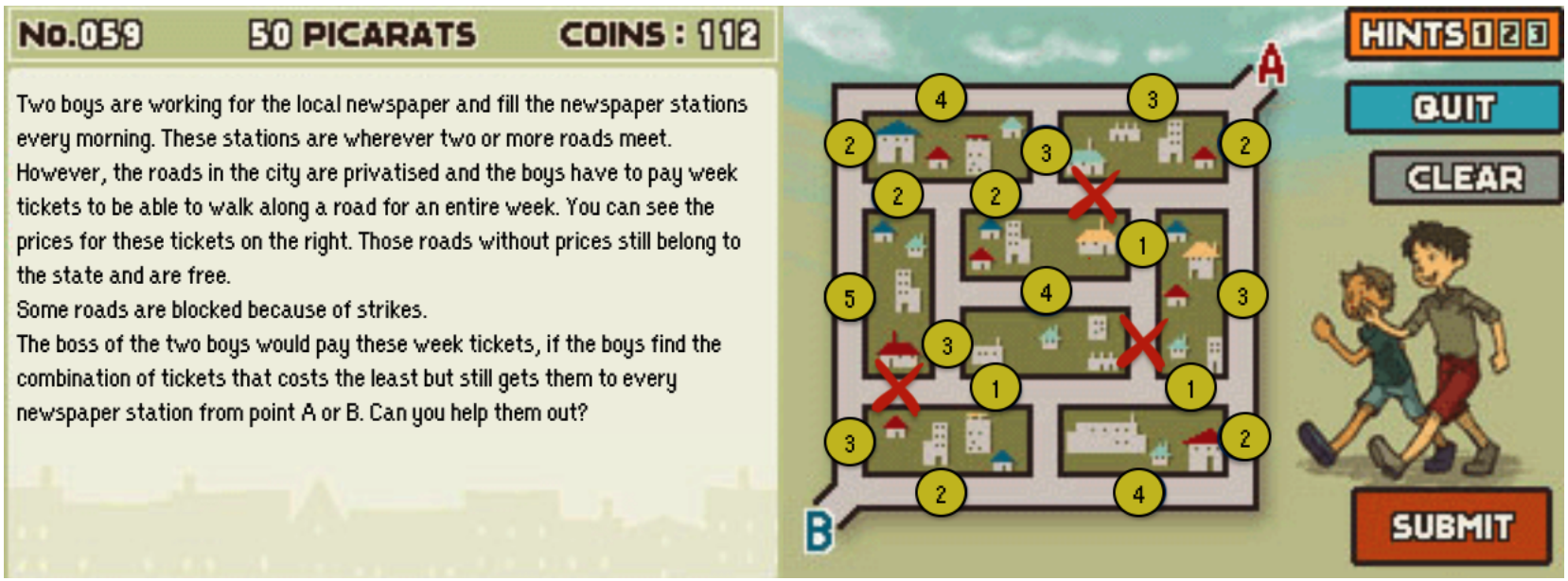
```
 1 function dijkstra(...)
 2
 3     while !isempty(pq)
 4         (i,d) = peek(pq)
 5         distance[i] = d
 6         visited[i] = true
 7         dequeue!(pq)
 8         for k in 1:length(neighbour[i])
 9             j = neighbour[i][k]
10             if !visited[j]
11                 dij = d + weight[i][k]
12                 if dij < pq[j]
13                     predecessor[j] = i
14                     pq[j] = dij
15                 end
16             end
17         end
18     end
19     return (distance, predecessor)
20 end
```

Proving Dijkstra's Algorithm

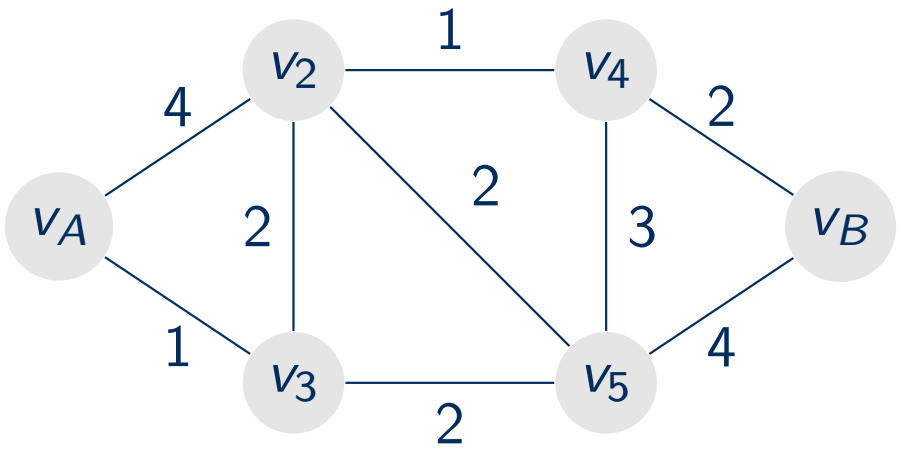
Proposition 4.2 *For a graph with m nodes, Dijkstra's algorithm terminates after m steps with the final distance.*

Any Questions?

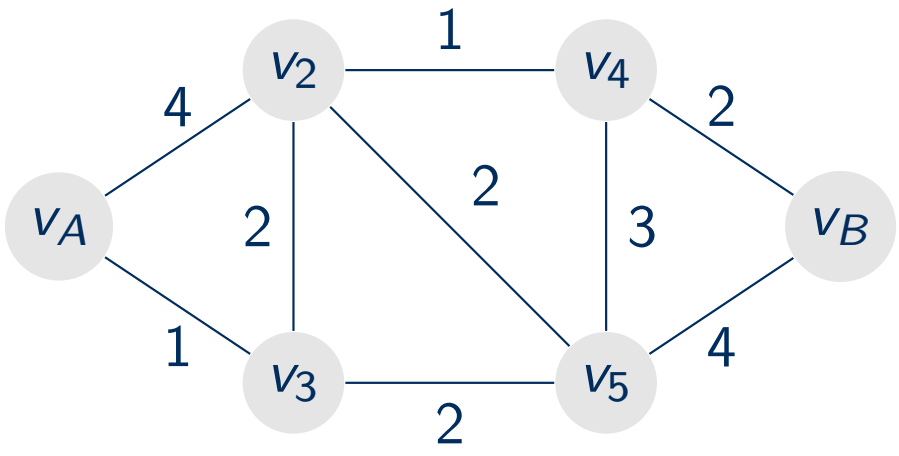
Another Layton Puzzle



Last Round of Definitions

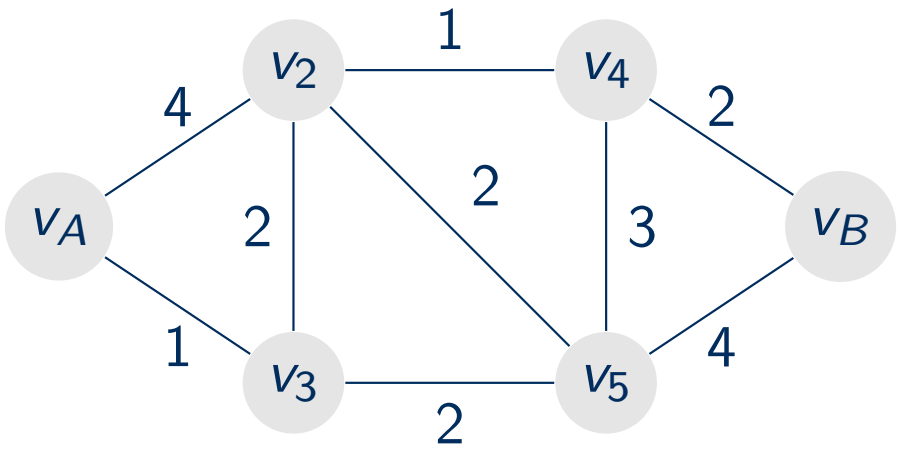


Last Round of Definitions



Cycle: A path that begins and ends at the same node

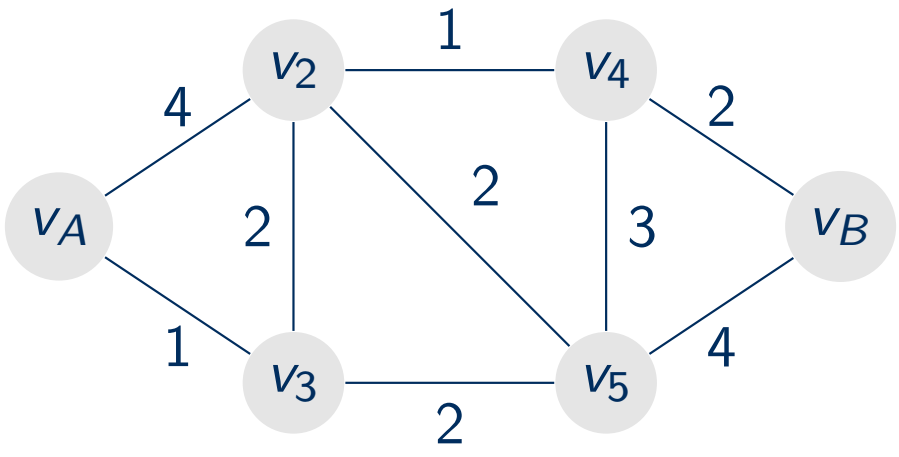
Last Round of Definitions



Cycle: A path that begins and ends at the same node

Tree: A cycleless graph

Last Round of Definitions

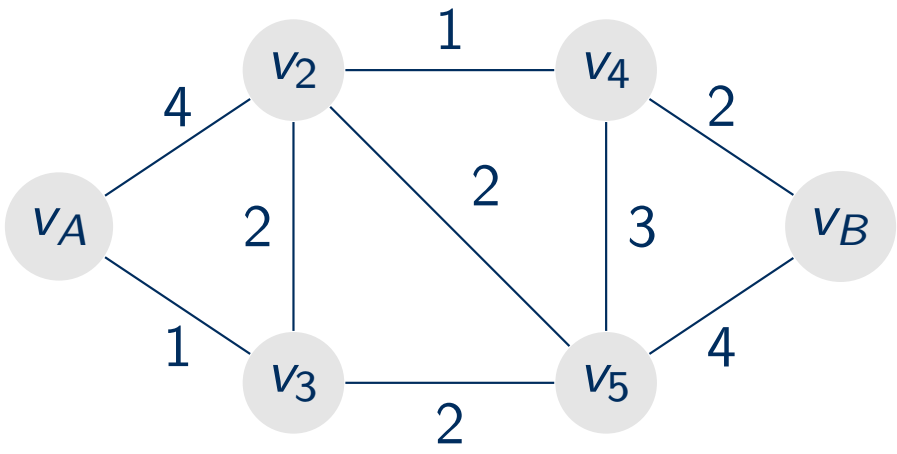


Cycle: A path that begins and ends at the same node

Tree: A cycleless graph

Spanning Tree *(ST)*: A subgraph $T \subseteq G$ where T is a tree and all nodes in G lie in T.

Last Round of Definitions



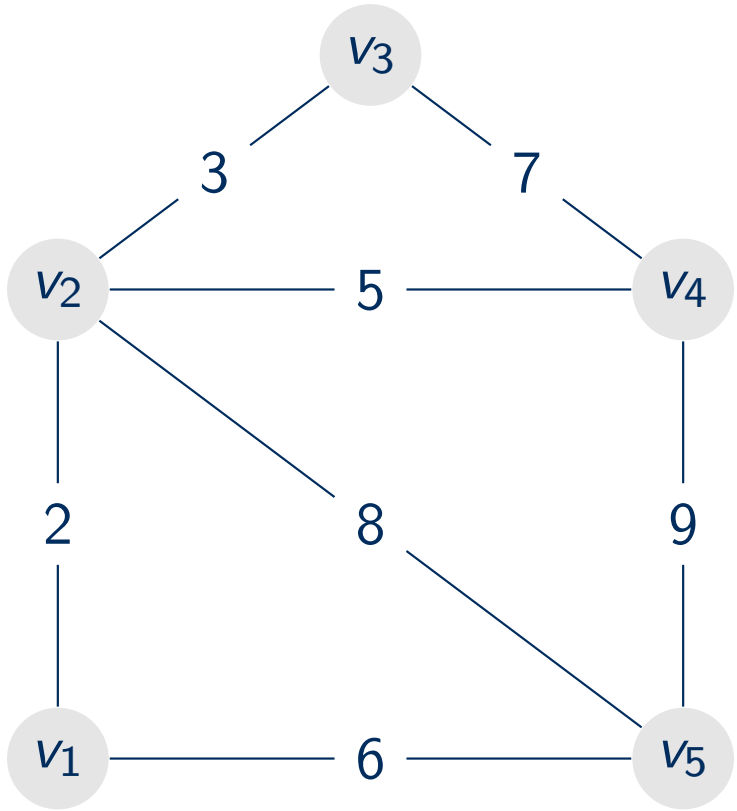
Cycle: A path that begins and ends at the same node

Tree: A cycleless graph

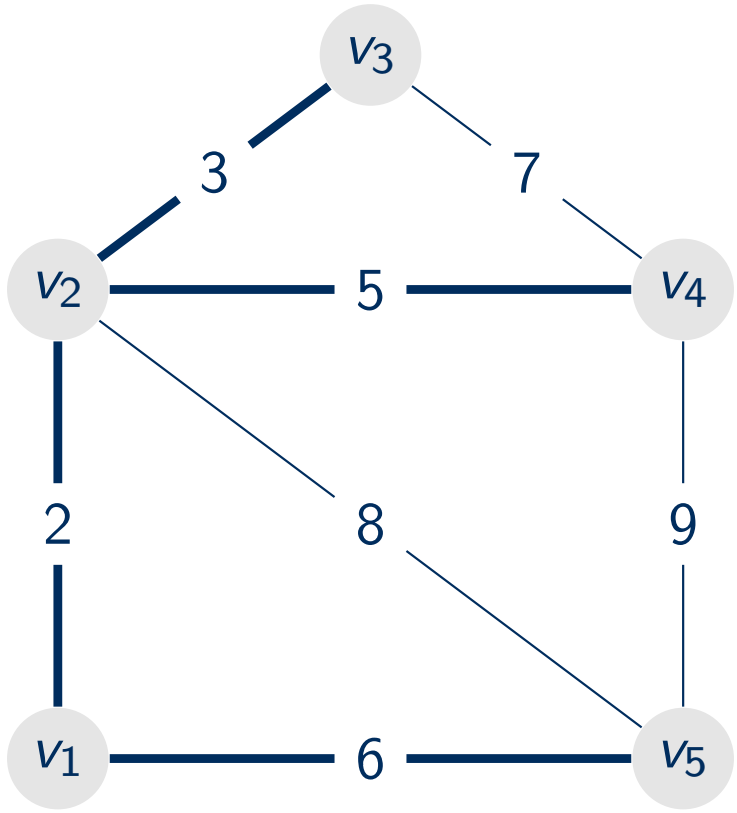
Spanning Tree *(ST)*: A subgraph $T \subseteq G$ where T is a tree and all nodes in G lie in T.

Minimum Spanning Tree: A ST with minimal total weight

Intuition



Intuition



Pseudo Code

```
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 checked
            add (k,l) to the spanning tree
            i = l
        end
        pop (k,l) off the priority queue
    end
    return tree
end
```

THE ALGORITHM (1)

```
1  function prim(neighbour::Array{Array{Int, 1}, 1},
2                weight::Array{Array{T, 1}, 1} where T <:
       Number
3      nodes = length(neighbour)
4      visited = falses(nodes)
5      (mst_nodes, i) = (1, 1)
6      key = Array{Tuple{Int, Int}, 1}(undef, 0)
7      priority = Array{Float64, 1}(undef, 0)
8      pq = PriorityQueue(zip(key, priority))
9      mst = Array{Tuple{Int, Int}, 1}(undef, 0) #The
       actual MST
10
11     (...)
12
13     return mst
14 end
```

THE ALGORITHM (2)

```
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)
        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
```

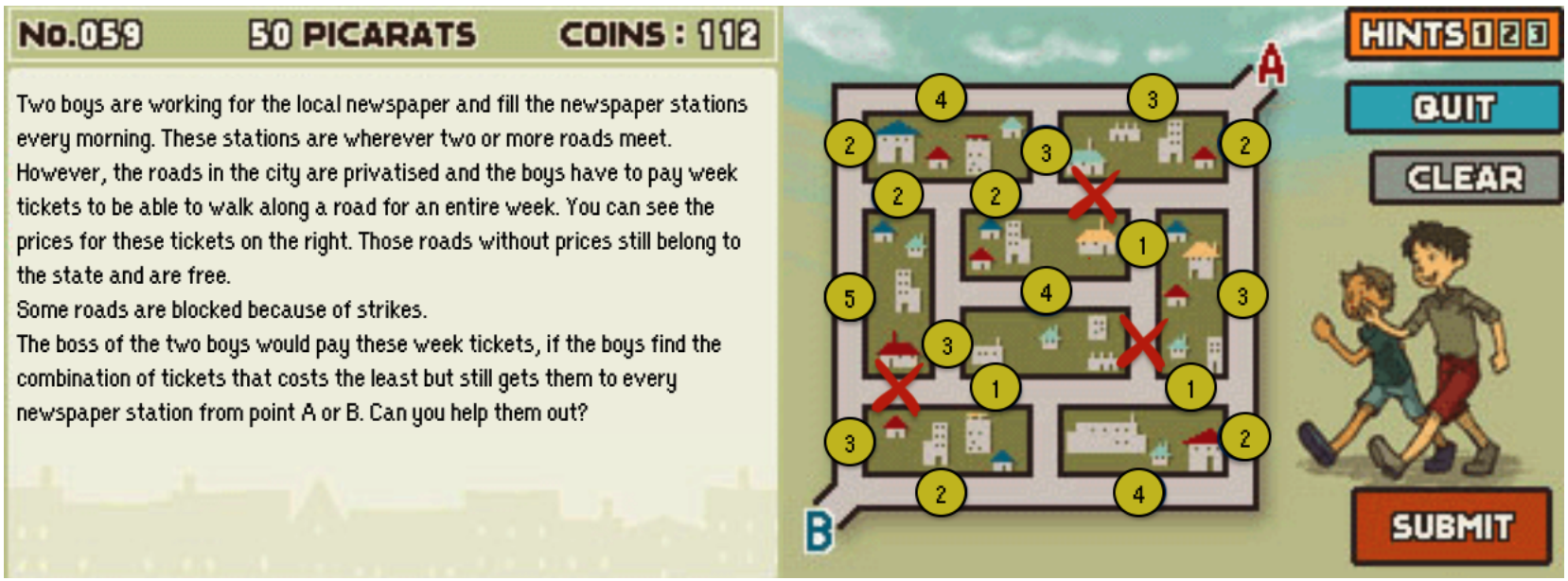
Helpful Proposition

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

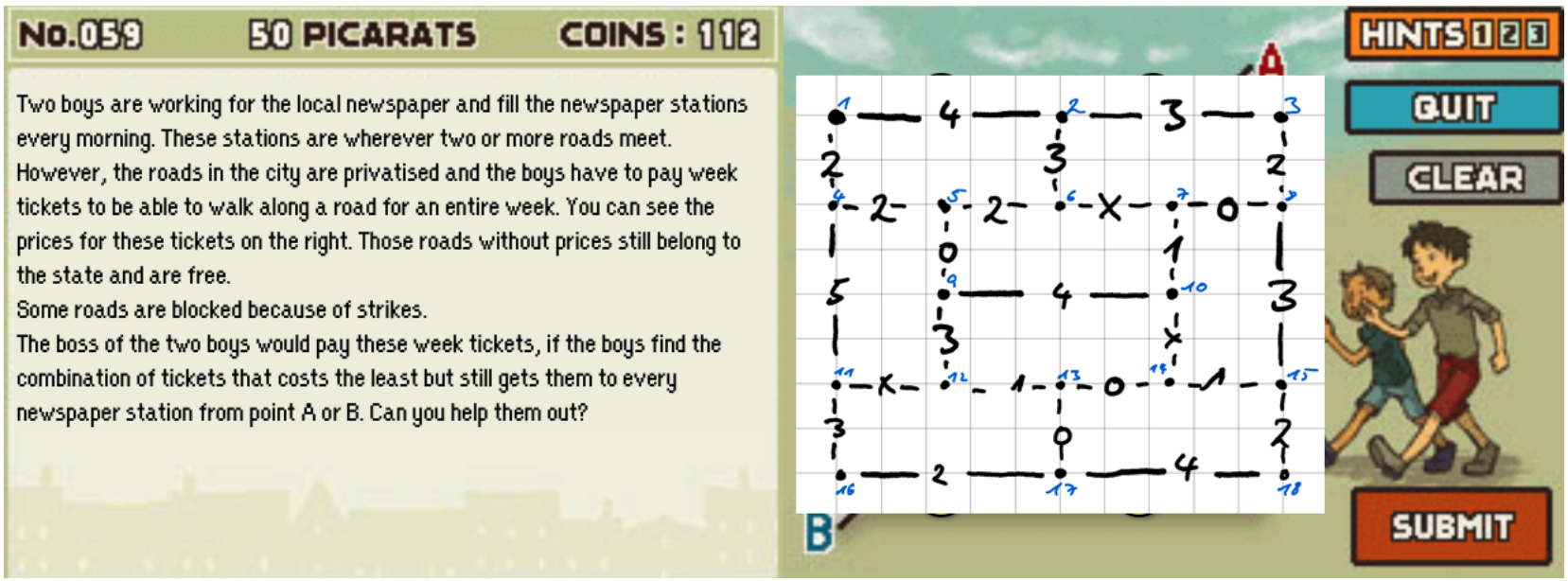
Proof of Prim's Algorithm

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

Application of Prim's Algorithm



Application of Prim's Algorithm



Application of Prim's Algorithm

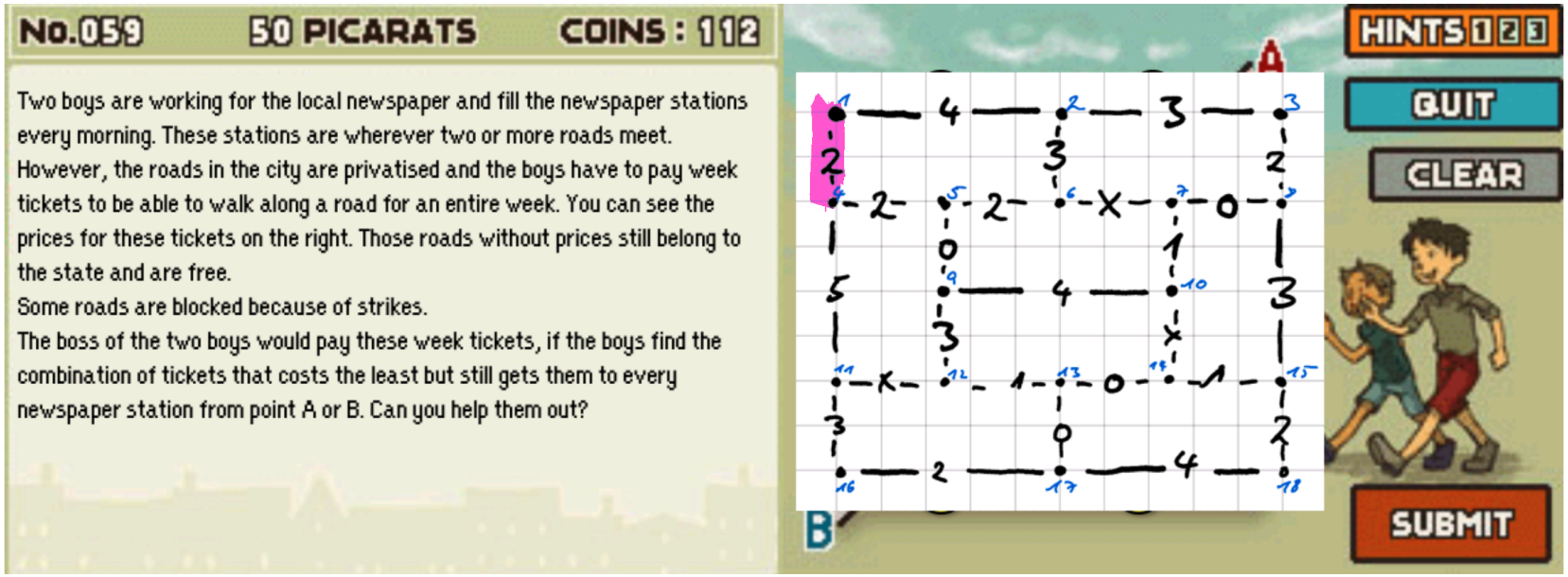
```
e = 1/100
                #   1 2 3 4 5 6 7 8 9 101112131415161718
LaytonMatrix₁ = [0 4 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
                 0 0 3 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0;
                 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0;
                 0 0 0 0 2 0 0 0 0 0 5 0 0 0 0 0 0 0;
                 0 0 0 0 0 2 0 0 e 0 0 0 0 0 0 0 0 0;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
                 0 0 0 0 0 0 0 e 0 1 0 0 0 0 0 0 0 0;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0;
                 0 0 0 0 0 0 0 0 0 4 0 3 0 0 0 0 0 0;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0;
                 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 e 0 0 e 0;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4;
                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
LaytonMatrix = LaytonMatrix₁ + LaytonMatrix₁'

prim(AdjacencyToNeighbourhood(LaytonMatrix)[1], AdjacencyToNeighbourhood(LaytonMatrix)[2])
```

✓ 0.9s

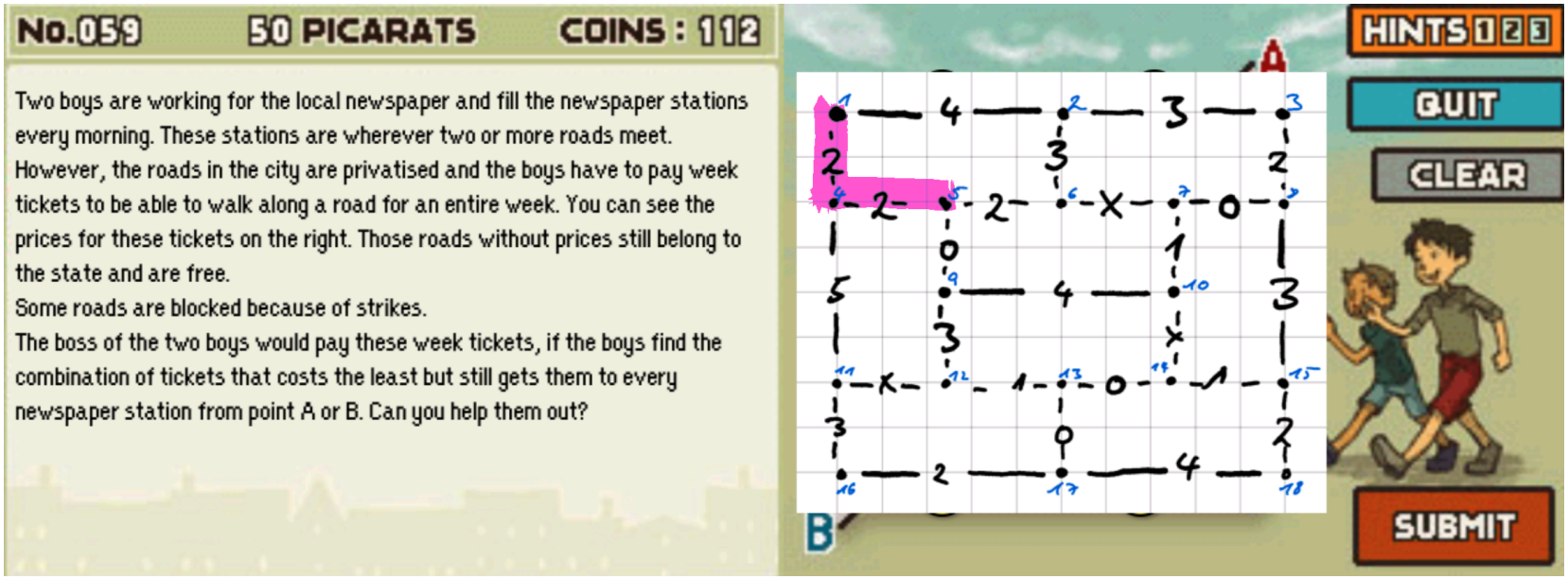
(1, 4), (4, 5), (5, 9), (5, 6), (6, 2),(9, 12), (12, 13), (13, 14), (13, 17), (14, 15), (15, 18), (17, 16), (16, 11), (2, 3), (3, 8), (8, 7), (7, 10)

Application of Prim's Algorithm



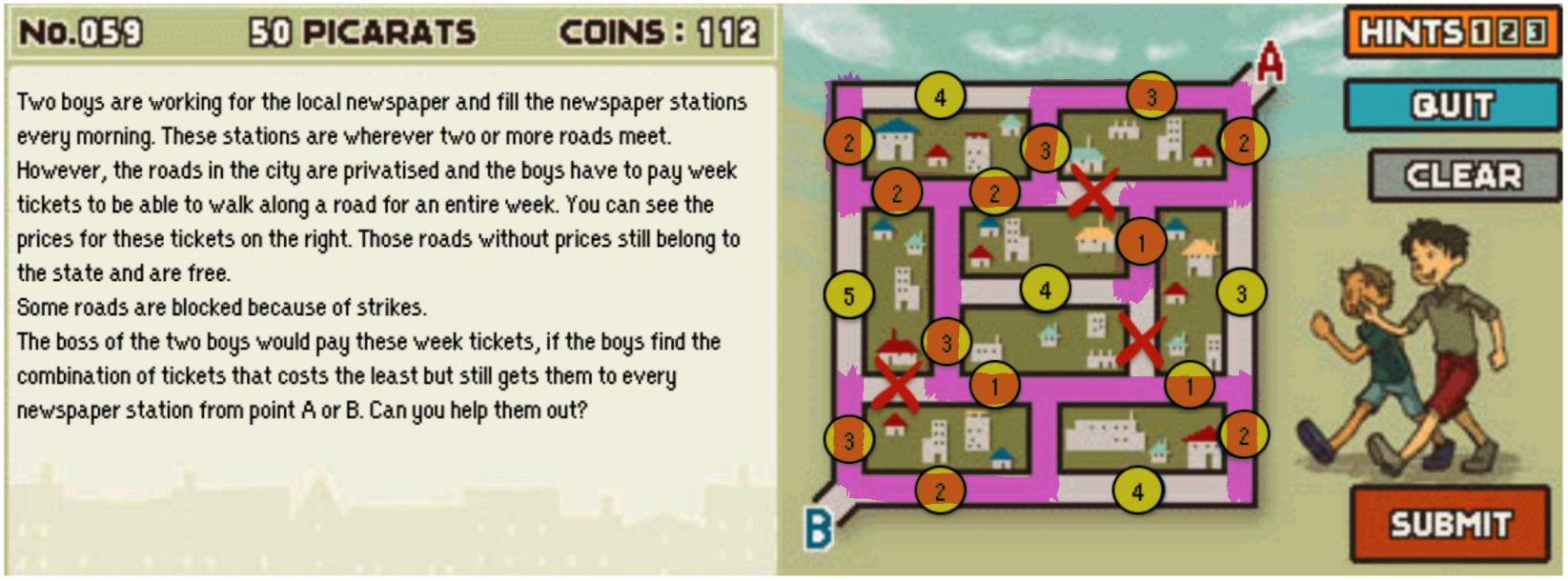
(1, 4), (4, 5), (5, 9), (5, 6), (6, 2),(9, 12), (12, 13), (13, 14), (13, 17), (14, 15), (15, 18), (17, 16), (16, 11), (2, 3), (3, 8), (8, 7), (7, 10)

Application of Prim's Algorithm



(1, 4), (4, 5), (5, 9), (5, 6), (6, 2),(9, 12), (12, 13), (13, 14), (13, 17), (14, 15), (15, 18), (17, 16), (16, 11), (2, 3), (3, 8), (8, 7), (7, 10)

Application of Prim's Algorithm



(1, 4), (4, 5), (5, 9), (5, 6), (6, 2),(9, 12), (12, 13), (13, 14), (13, 17), (14, 15), (15, 18), (17, 16), (16, 11), (2, 3), (3, 8), (8, 7), (7, 10)

Thank you!