SIMULATION COMPONENTS

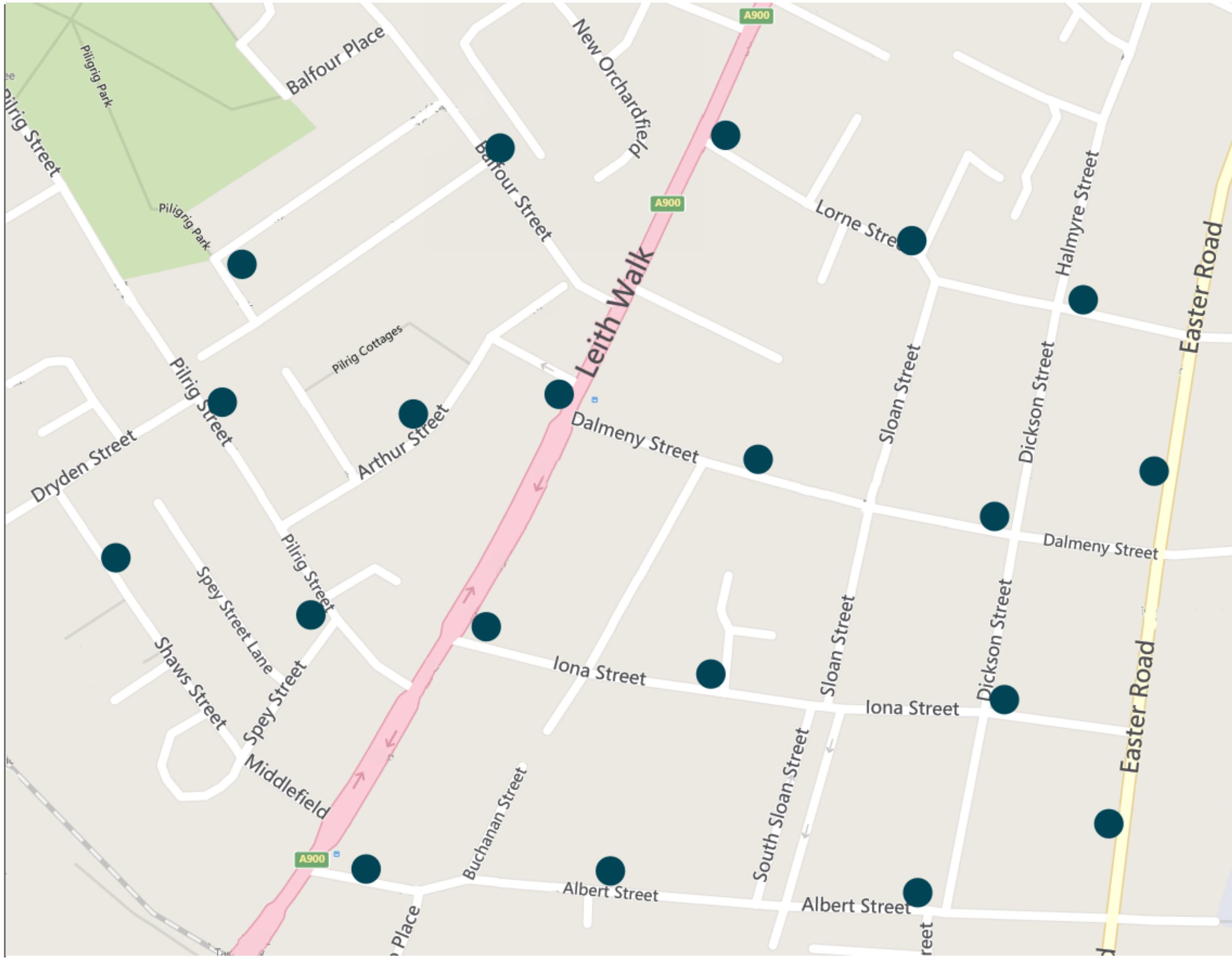
ROUTE PLANNING

SERVICE NETWORK

- We need an abstract representation of a street map and bus stop locations for the service network.
- We need to model the roads between different locations and the time require to travel these.
- We need to account for the fact that some streets only allow one way traffic.

EXAMPLE

Leith Walk area in Edinburgh; 20 imagined stop locations

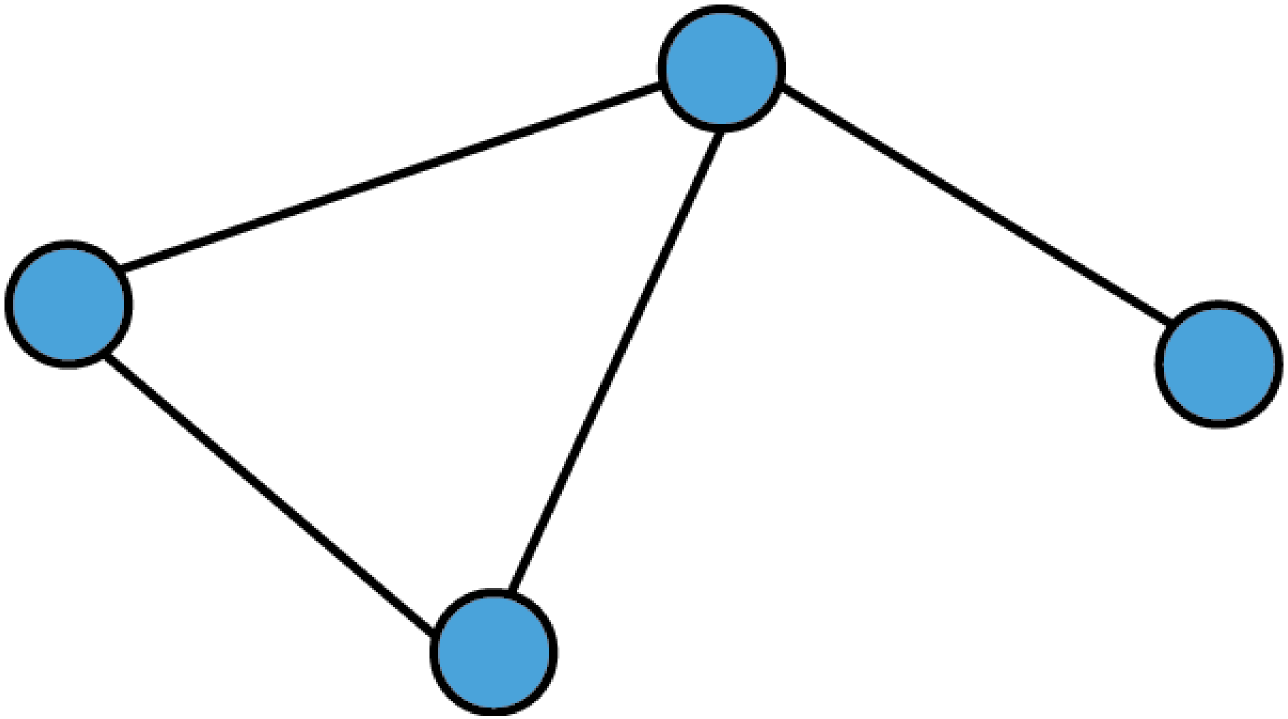


Map source: bing.com

GRAPH REPRESENTATION

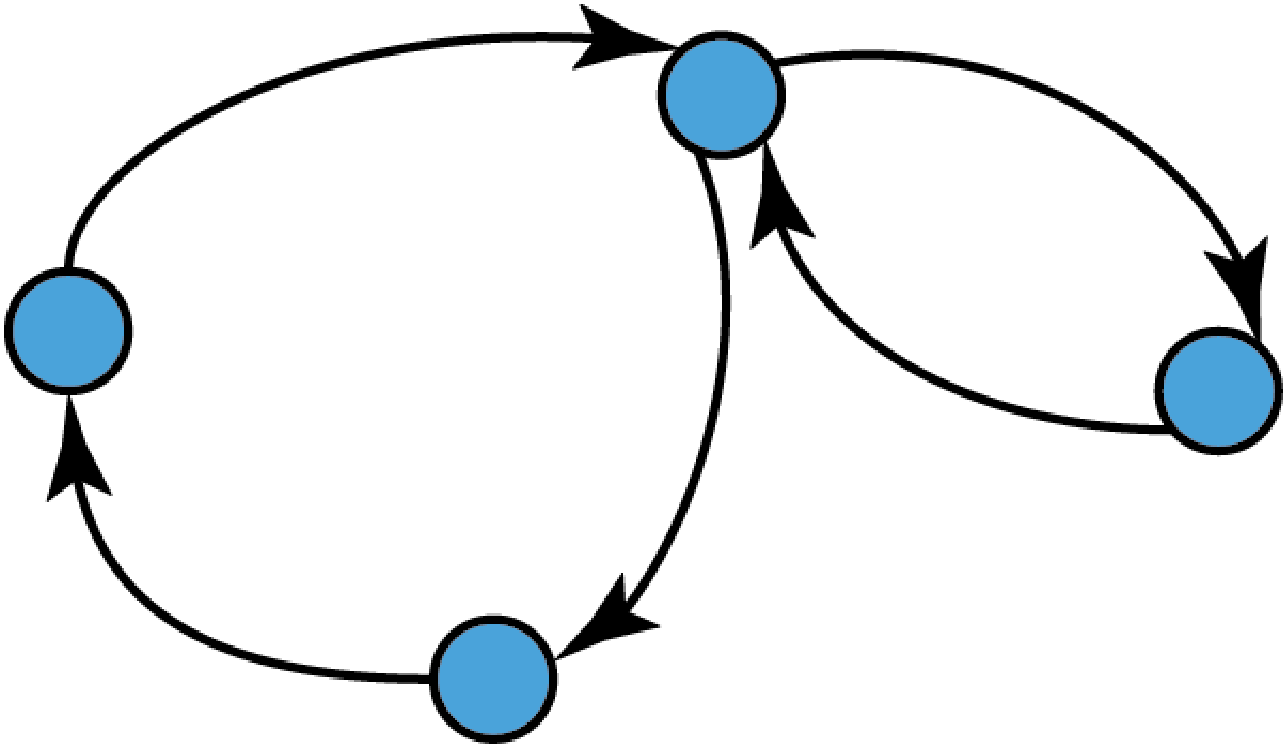
- In mathematical terms such a collection of bus stops interconnected with street segments can be represented through a graph.
- A graph $G = (V,E)$ comprises a set of vertices V that represent objects (bus stops) and E edges that connect different pairs of vertices (links/street segments).
- Graphs can be *directed* or *undirected*.

UNDIRECTED GRAPHS



- Edges have no orientation, i.e. they are unordered pairs of vertices. That is there is a symmetry relation between nodes and thus *(a,b) = (b,a)*.

DIRECTED GRAPHS



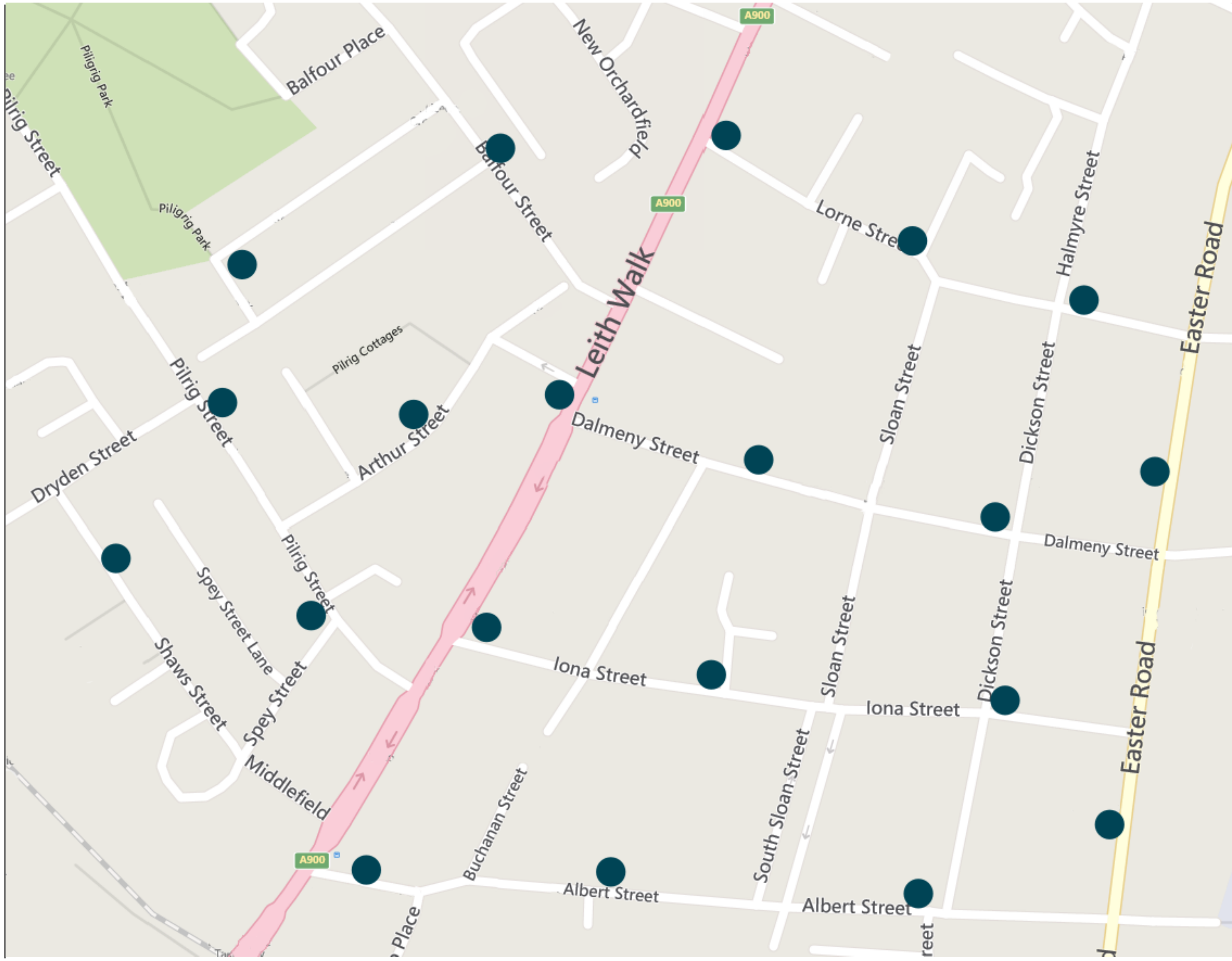
- Edges have a direction associated with them and they are called *arcs* or directed edges.
- Formally, they are *ordered* pairs of vertices, i.e. $(a,b) \neq (b,a)$ if $a \neq b$.

GRAPH REPRESENTATION IN YOUR SIMULATORS

- For our simulations we will consider *directed* graph representations of the service network.
- This will increase complexity, but is more realistic.

BACK TO THE EXAMPLE

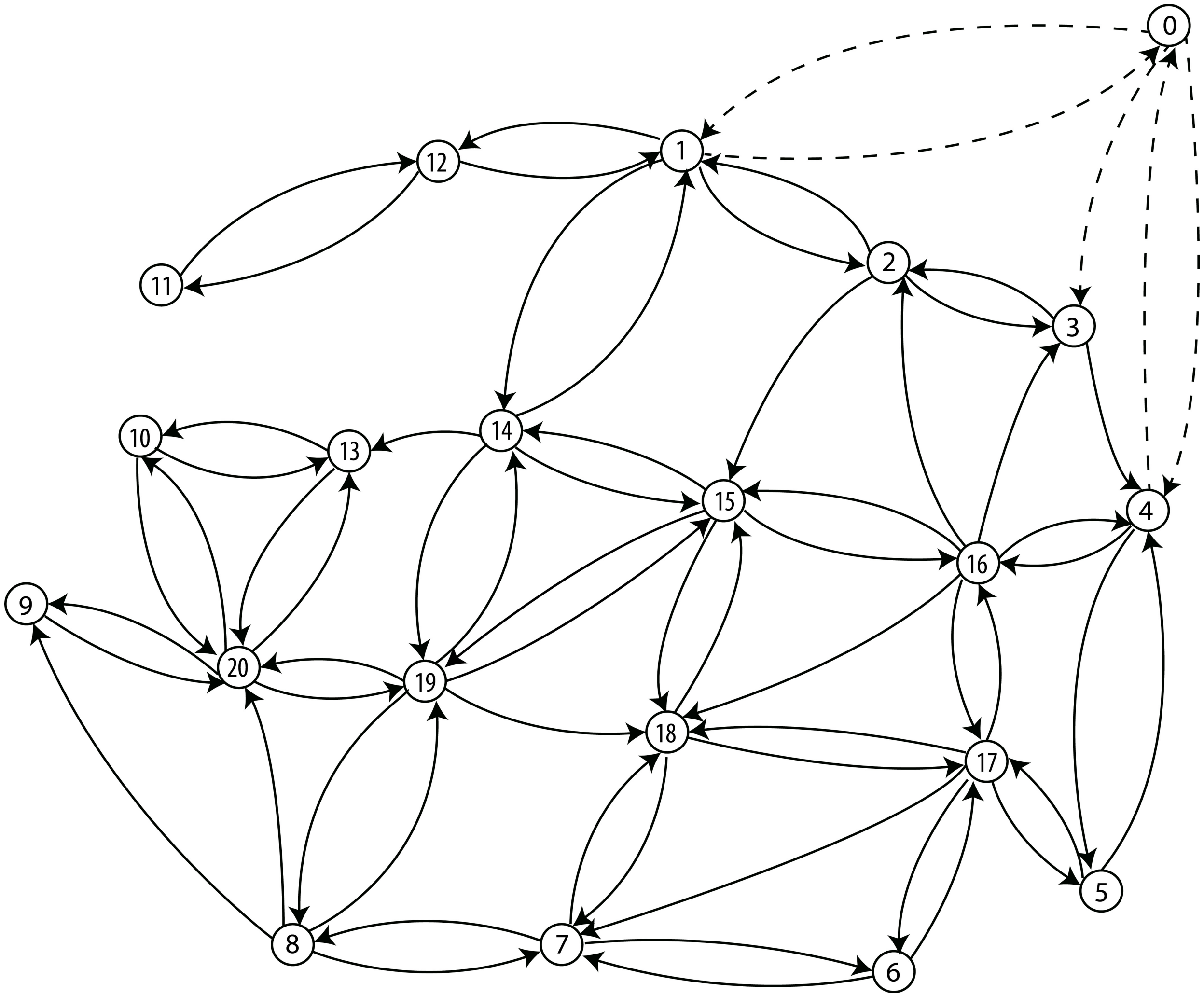
This area…



Map source: bing.com

CORRESPONDING GRAPH

...can be represented by



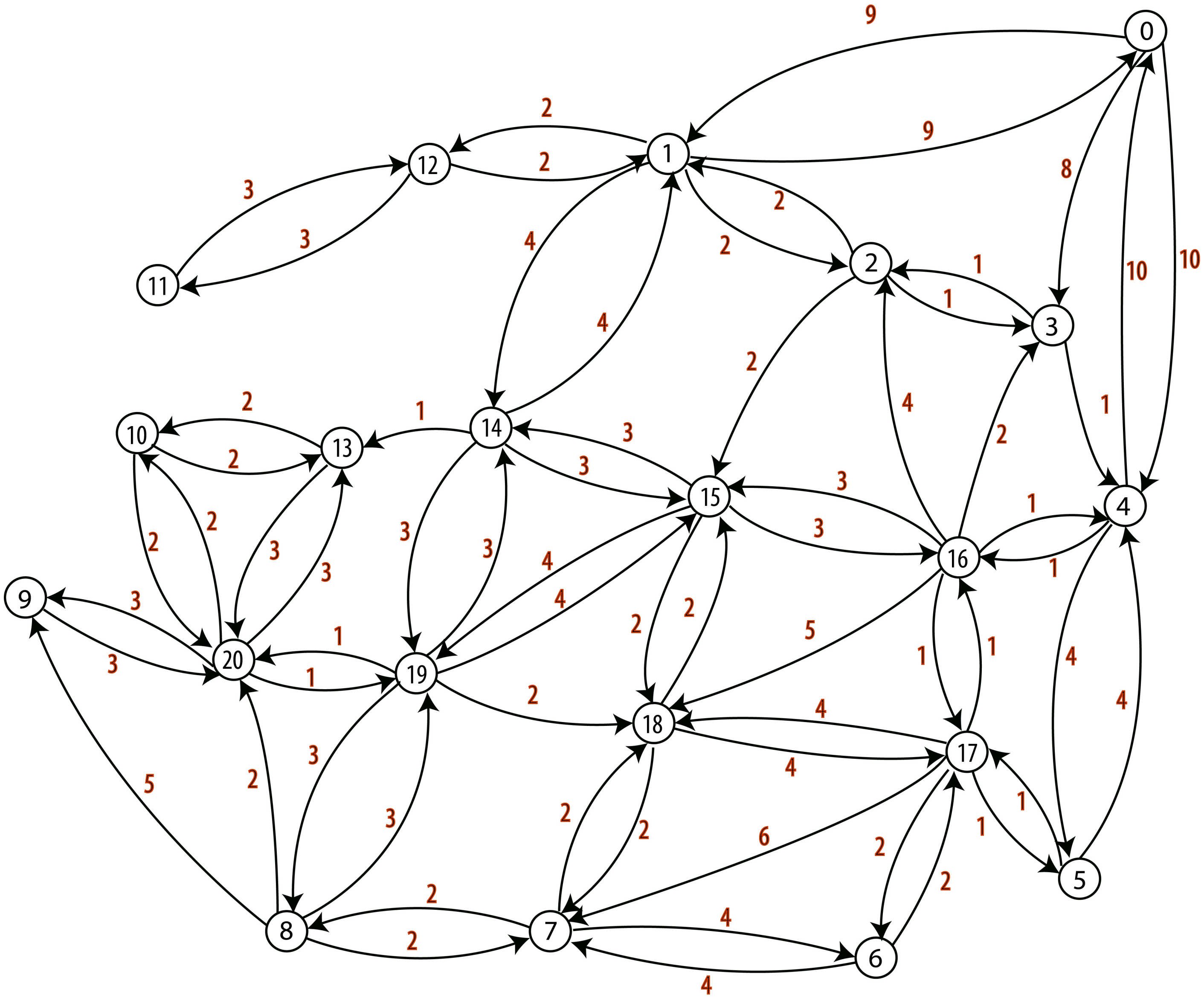
We numbered vertices & added node '0' for the garage.

WEIGHTED GRAPH

- We also need to model the distances between stop locations.
- We will use a *weighted graph* representation, where a number (weight) is associated to each arc.
- In our case weights will represent the average travel duration between two stops (vertices) in one direction, expressed in minutes.

WEIGHTED GRAPH

For our example, this may be



INPUT SCRIPT

- Graph representation of the bus stop locations and distances between them will be given in the input script in *matrix* form.
- We will consider the garage as bus stop 0. For a service network with *N* stops a *N x N* matrix will be specified.
- The `map` keyword will precede the matrix.
- Where there is no arc in the graph between two vertices we will use a -1 value in the matrix.

FOR THE PREVIOUS EXAMPLE

	0	1	2	3	4	5	...	19	20
0	0	9	-1	**8**	10	-1	...	-1	-1
1	9	0	2	-1	-1	-1	...	-1	-1
2	-1	2	0	1	-1	-1	...	-1	-1
3	**-1**	-1	1	0	1	-1	...	-1	-1
4	10	-1	-1	1	0	4	...	-1	-1
5	-1	-1	-1	-1	4	0	...	-1	-1
.	.	.	.	.	.	.		.	.
.	.	.	.	.	.	.		.	.
.	.	.	.	.	.	.		.	.
19	-1	-1	-1	-1	-1	-1	...	0	1
20	-1	-1	-1	-1	-1	-1	...	1	0

*Note that the matrix is not symmetric.

ROUTE PLANNING

- Minibuses may be scheduled depending on different parameters:
  1. The maximum time a user is willing to wait (`maxDelay`).
  2. The difference between desired departure time of a user (related to `pickupInterval`) and the time of the request (related to `requestRate`).
- Based on a set of requests, you must compute the shortest routes that pick up and drop off the largest possible number of passengers that intend to take similar journeys.

ROUTE PLANNING

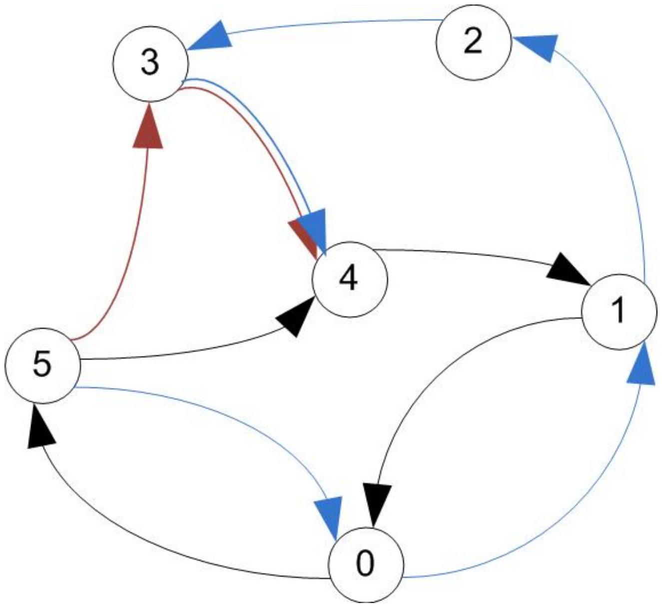
- There may not be passengers boarding/disembarking at all the bus stops along a route.
- Thus it may be appropriate to work with an equivalent graph where vertices that do not require to be visited are isolated and equivalent arc weights are introduced.
- Sometimes it may be more efficient to travel multiple times through the same location, even if the route previously serviced passengers who had placed requests there.

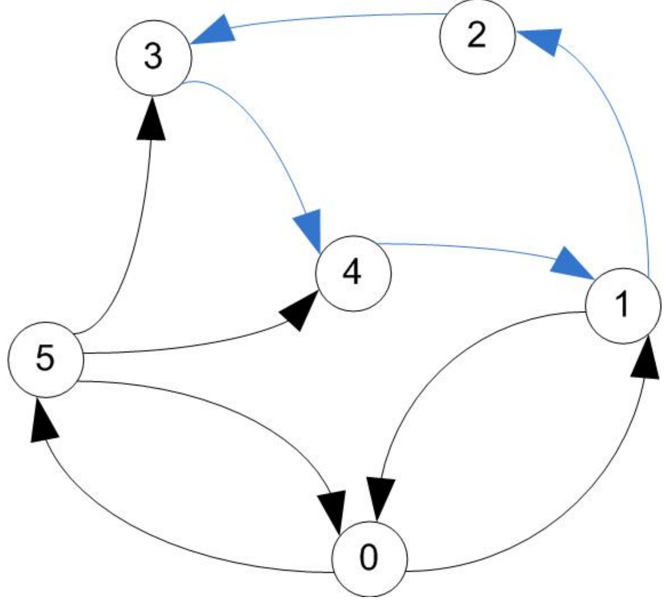
THE (MORE) CHALLENGING PART

- Let's refer to the graph of all bus stops where service is required at a given time by "service graph".
- How to partition the service graph and find (almost) optimal routes that visit all vertices in the service graph with minimum cost?
- This is entirely up to you, but I will discuss some useful aspects next.
- You must justify your choice in the final report and comment appropriately the simulator code.
- You may wish to implement more than one algorithm.

USEFUL TERMINOLOGY

- A *walk* is a sequence of arcs connecting a sequence of vertices in a graph.
- A *directed path* is a walk that does not include any vertex twice, with all arcs in the same direction.
- A *cycle* is a path that starts & ends at the same vertex.

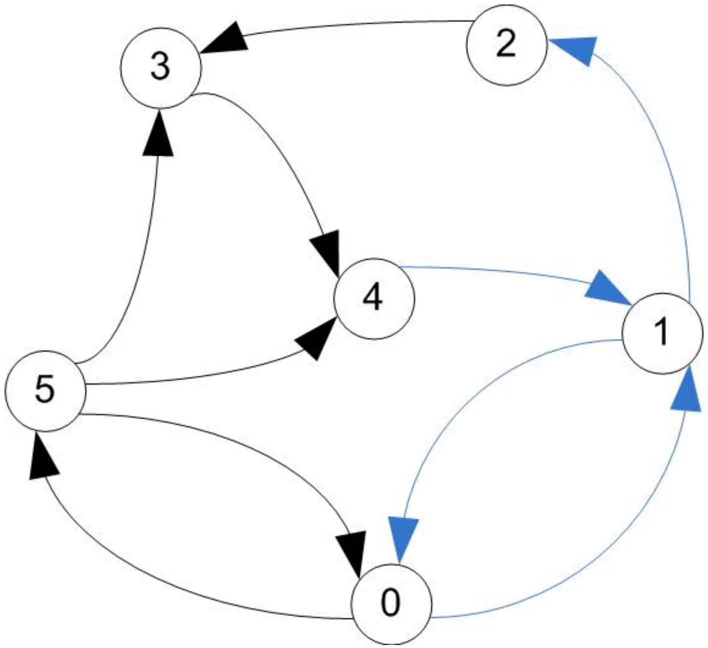


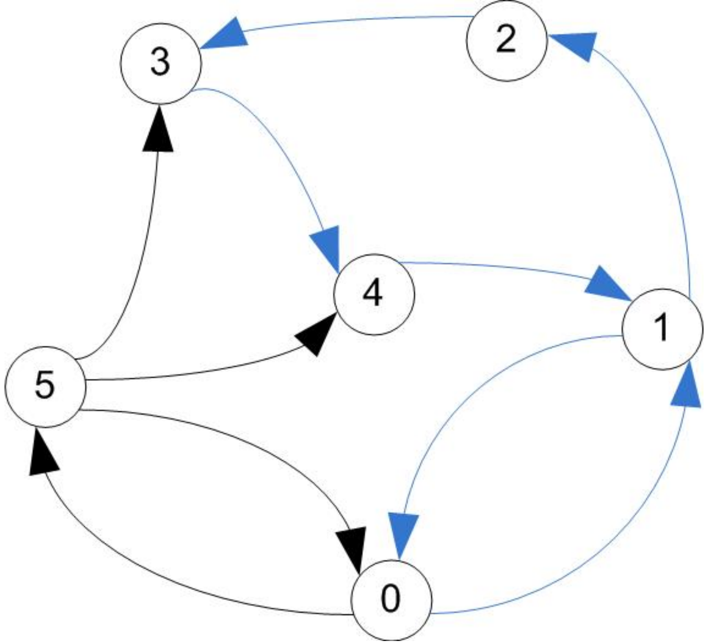


directed paths / cycle

USEFUL TERMINOLOGY

- A *trail* is a walk that does not include any arc twice.
- A trail may include a vertex twice, as long as it comes and leaves on different arcs.
- A *circuit* is a trail that starts & ends at the same vertex

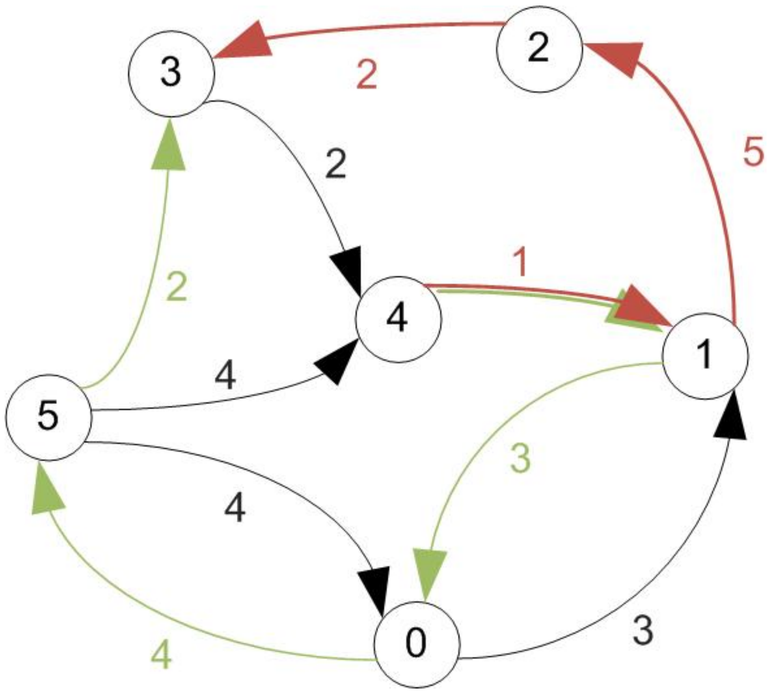




trail / circuit (tour)

SHORTEST PATHS

- There may be multiple paths that connect two vertices in a directed graph.
- In a *weighted* graph the shortest path between two vertices is that for which the sum of the arc costs (weights) is the smallest.



SHORTEST PATHS

- There are several algorithms you can use to find the shortest paths on a given service network.
- A non-exhaustive list includes
  - Dijkstra's algorithm (single source),
  - Floyd-Warshall algorithm (all pairs),
  - Bellman-Ford algorithm (single source).
- Each of these have different complexities, which depend on the number of vertices and/or arcs.
- The size and structure of the graph will impact on the execution time.

FLOYD–WARSHALL ALGORITHM

- A single execution finds the lengths of the shortest paths between **all** pairs of vertices.
- The standard version does not record the sequence of vertices on each shortest path.
- The reason for this is the memory cost associated with large graphs.
- We will see however that paths can be reconstructed with simple modifications, without storing the end-to-end vertex sequences.

FLOYD–WARSHALL ALGORITHM

- Complexity is $O(N^3)$, where N is the number of vertices in the graph.

  The core idea:

- Consider $d_{i,j,k}$ to be the shortest path from i to j obtained using intermediary vertices only from a set {1,2,...,k}.
- Next, find $d_{i,j,k+1}$ (i.e. with nodes in {1,2,...k+1}.
  - This could be $d_{i,j,k+1} = d_{i,j,k}$ or
  - A path from vertex i to k+1 concatenated with a path from vertex k+1 to j.

FLOYD–WARSHALL ALGORITHM

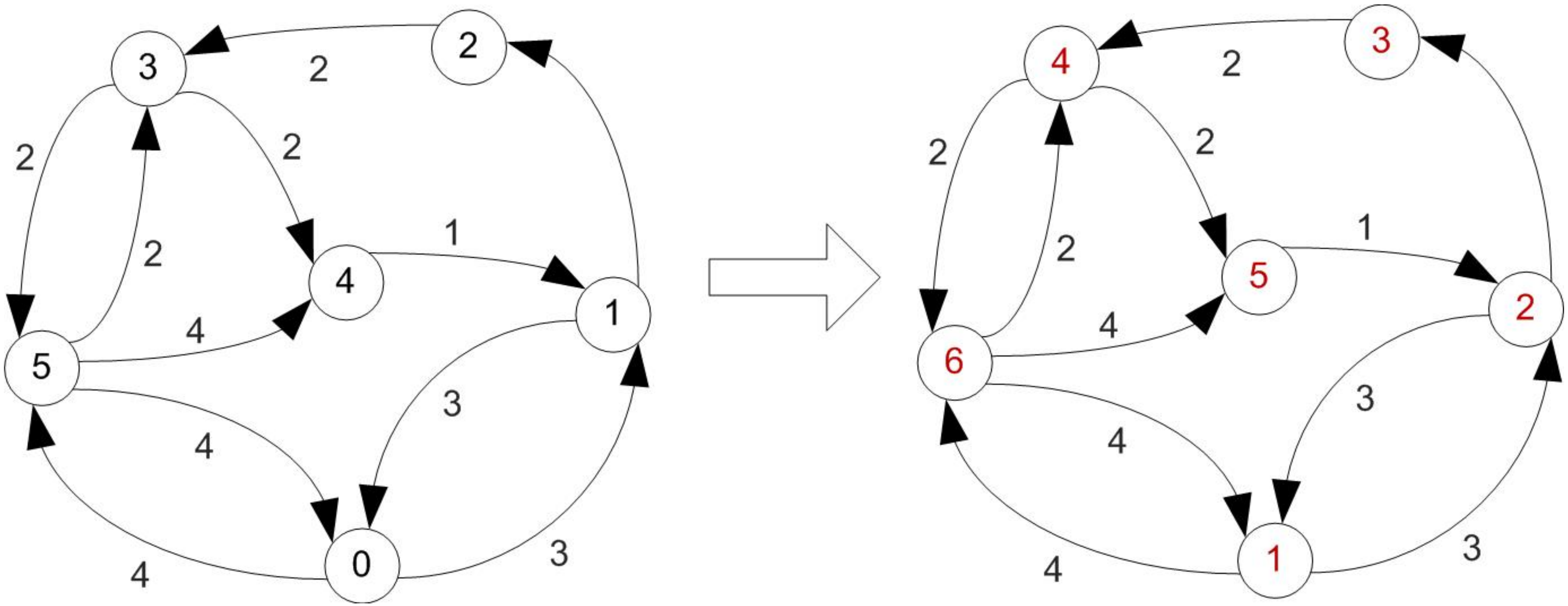
- Then we can compute all the shortest paths recursively as

$$d_{i,j,k+1} = \min(d_{i,j,k}, d_{i,k+1,k} + d_{k+1,j,k}).$$

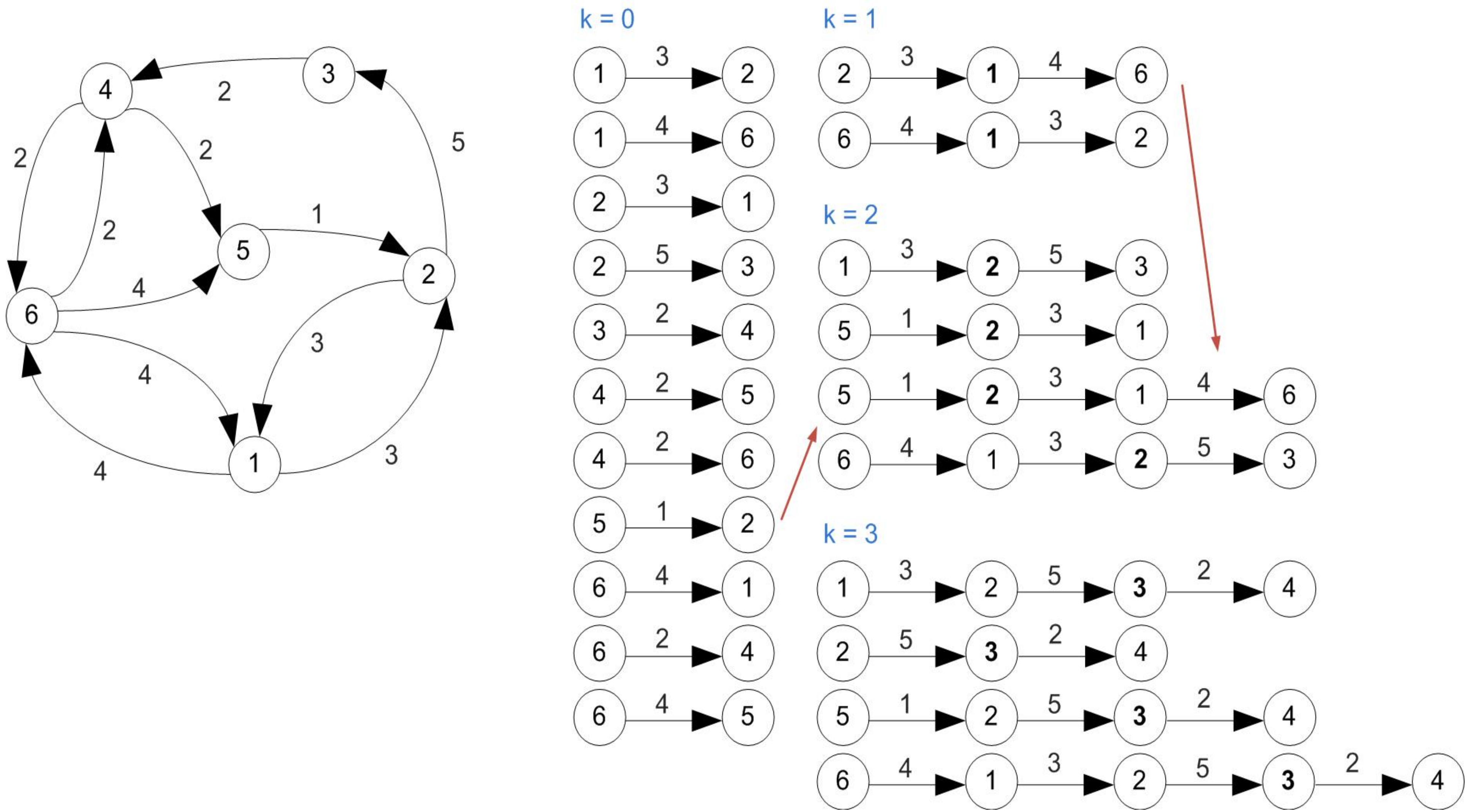
- Initialise $d_{i,j,0} = w_{i,j}$ (i.e. start form arc costs).
- Remember that in your case the absence of an arc between vertices is represented as a -1 value, so you will need to pay attention when you comput the minimum.

EXAMPLE

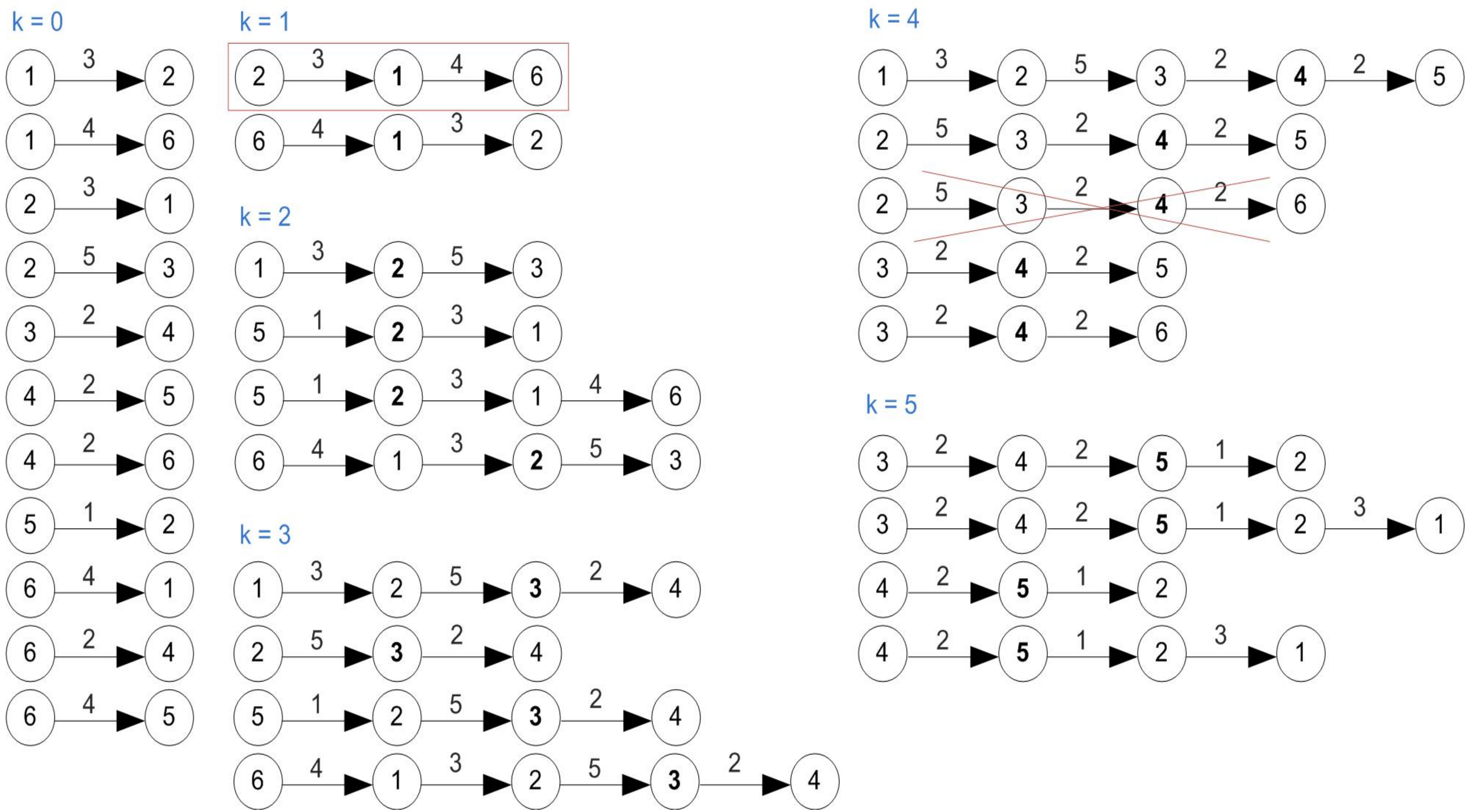
First let's increase vertex indexes by one, since we were starting at 0.



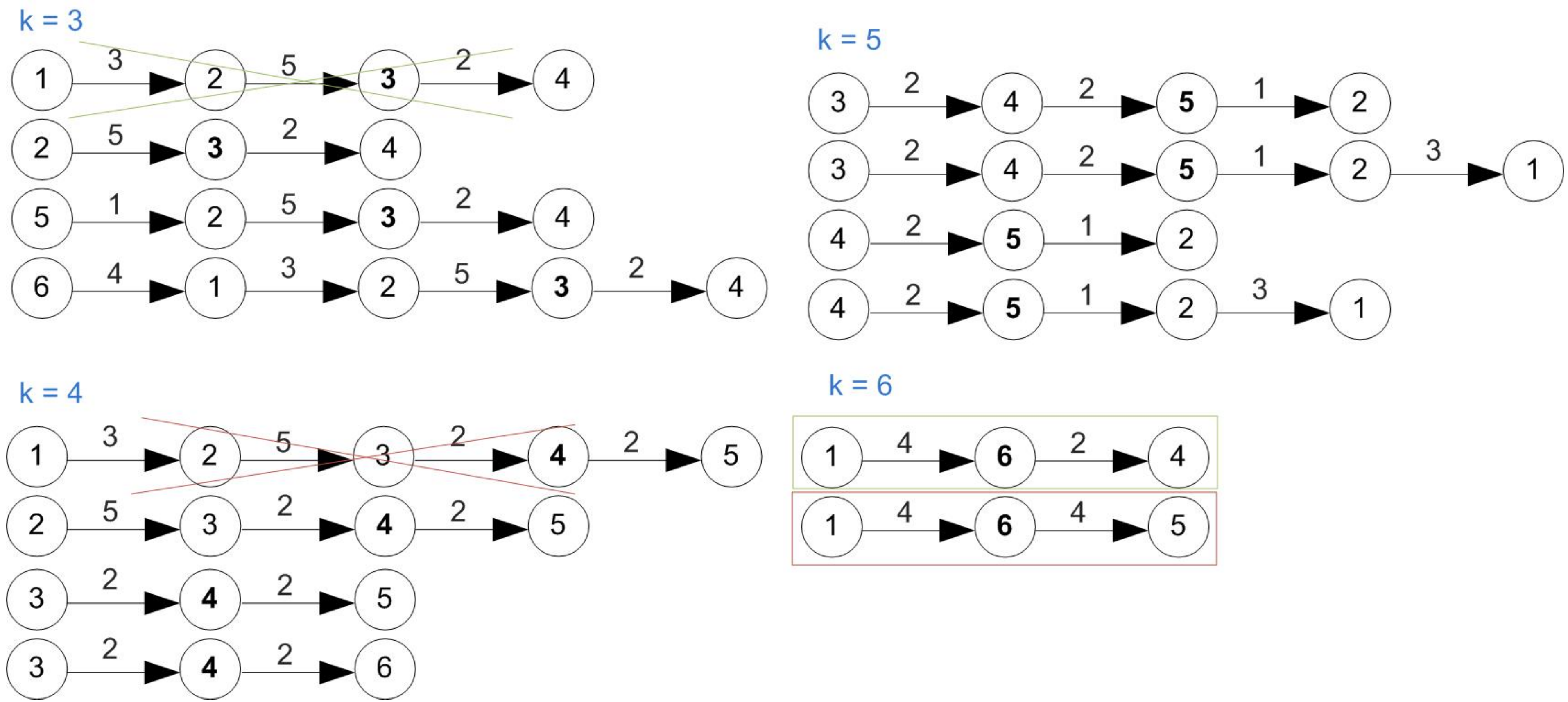
EXAMPLE



EXAMPLE (CONT'D)



EXAMPLE (CONT'D)



All shortest paths found at this step.

PSEUDOCODE

```
Denote d the N × N array of shortest path lengths.
Initialise all elements in d with inf.

For i = 1 to N
   For j = 1 to N
     d[i][j] ← w[i][j]   // assign weights of existing arcs;

For k = 1 to N
   For i = 1 to N
       For j = 1 to N
         If d[i][j] > d[i][k] + d[k][j]
             d[i][j] ← d[i][k] + d[k][j]
         End If
```

FLOYD–WARSHALL ALGORITHM

- This will give you the lengths of the shortest paths between each pair of vertices, but not the entire path.
- You do not actually need to store all the paths, but you would want to be able to reconstruct them easily.
- The standard approach is to compute the shortest path tree for each node, i.e the spanning trees rooted at each vertex and having the minimal distance to each other node.

PSEUDOCODE

```
Denote d, nh the N × N arrays of shortest path lengths and
respectively the next hop of each vertex.

For i = 1 to N
   For j = 1 to N
     d[i][j] ← w[i][j]   // assign weights of existing arcs;
     nh[i][j] ← j

For k = 1 to N
   For i = 1 to N
       For j = 1 to N
         If d[i][j] > d[i][k] + d[k][j]
             d[i][j] ← d[i][k] + d[k][j]
             nh[i][j] ← nh[j][k]
         End If
```

RECONSTRUCTING THE PATHS

To retrieve the sequence of vertices on the shortest path between nodes *i* and *j*, simply run a routine like the following.

```
path ← i
While i != j
  i ← nh[i][j]
  append i to path
EndWhile
```

FINDING OPTIMAL ROUTES GIVEN A SET OF USER REQUIREMENTS

- Finding shortest paths between different bus stops is only one component of route planning.
- The problem you are trying to solve is a flavour of the Vehicle Routing Problem (VRP). This is a known *NP-hard* problem.
- Simply put, an optimal solution may not be found in polynomial time and the complexity increases significantly with the number of vertices.

HEURISTIC ALGORITHMS

- Heuristics work well for finding solutions to hard problems in many cases.
- Solutions may not be always optimal, but good enough.
- Work relatively fast.
- When the number of vertices is small, a 'brute force' approach could be feasible.
- Guaranteed to find a solution (if there exists one), and this will be optimal.

CHOOSING ROUTE PLANNING ALGORITHMS

- You have complete freedom to choose what heuristic you implement, but
- make sure you document your choice and discuss its implication on system's performance in your report.
- It is likely that you will need to compute shortest paths.
- Again, you can choose any algorithm for this task, e.g. Floyd-Warshall, Dijkstra, etc., but explain your choice.
- You can implement multiple solutions, as some may not work for any graph o will perform poorly.