

Case study 31

Path finders, rovers and Ariadne's thread

Objective: This chapter tackles a widespread optimization problem: computing the shortest path between two cities. The solving technique is based on Dijkstra's algorithm. It also is applied to two similar applications domains, namely maze solving and controlling a rover on a hostile planet. A general purpose, application independent, solving tool is developed.

Keywords: optimization, shortest path, Dijkstra's algorithm, maze solving, rover control

31.1 Shortest paths between two cities

Let us consider a set of cities between which roads have been built (Figure 31.1). We associate a *distance* with each road. For instance, the road from city **A** to city **B** is **85** km long. We will write: $d_{AB} = 85$. To compute the distance between cities **A** and **F** we add distances **85** and **80**, that is, $d_{AF} = 165$ km.

Computing d_{AJ} , the distance between **A** and **J**, is a bit more complex. Indeed, the figure shows that three different paths with three different lengths connect city **A** to city **J**, namely:

- **A.B.F.I.J** with $d_{AJ} = 499$ km
- **A.C.H.J** with $d_{AJ} = 487$ km

– **A.E.J** with $d_{AJ} = 675$ km

Hence the concept of *shortest path* between two cities. Computing efficiently this shortest path is the main objective of this chapter.

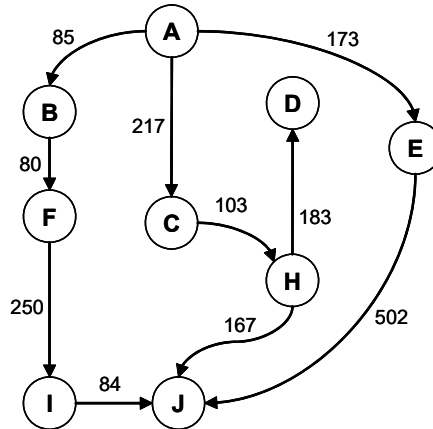


Figure 31.1 - A map showing cities and roads between them

31.2 Dijkstra's Algorithm

Dijkstra's algorithm is the most popular technique to find the *shortest path* between two nodes of a graph¹.

Each edge of the graph is given a *cost* that measures the *effort* required for traveling from the source node to the target node. This cost (a non-negative number) can be the distance or more generally some resource (such as money, time, energy or gasoline), depending on the nature of the problem.

We will develop one of the variants of this algorithm, that computes the shortest paths between a definite node, the *starting node*, and each of the other nodes of the graph (actually, one of the shortest paths, since more than one path with the same length may exist), a structure called the *shortest-path tree*. By querying this tree, we can extract the shortest path between the starting node and any target node.

Let **A** be the starting node of the graph. We assign to each node a property that specifies its distance from this starting node. Before applying the algorithm, we set this property to **0** for node **A** (the distance between **A** and **A** is zero) and **infinite** for all the other nodes (Figure 31.2). The goal of the algorithm is to adjust this distance until we get its minimal value.

1. See https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm for example

We distinguish in this graph two subsets of nodes, the nodes for which the minimal distance has already been computed and the other nodes, for which the minimal distance has yet to be evaluated. Let us call the first category *internal nodes* (they belong to the solution) and the others *external nodes*. Before applying the algorithm, the set of internal nodes is empty. Internal nodes are drawn in dark gray and the external nodes in white.

We will examine each of the external nodes, one by one. When we select a node to examine, we color it in light gray.

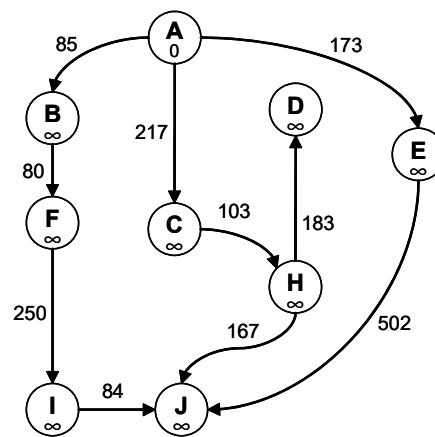


Figure 31.2 - Initial state of the graph

Quite naturally, we examine starting node **A** first, which is then colored in light gray (Figure 31.3/left). The distance of **B** from **A**, d_B , is set to d_{AB} . More precisely, $d_B = d_A + d_{AB}$. The same modification is applied to the other neighbor nodes **C** and **E**. The result is shown in Figure 31.3/left. Node **A** is (trivially) an internal node and is colored in dark gray (Figure 31.3/right).

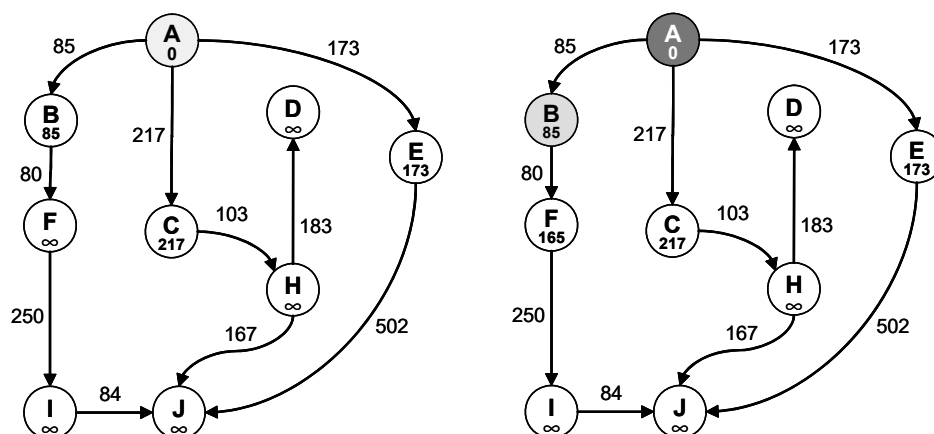


Figure 31.3 - Processing cities **A** and **B**

Here comes the smart part of the algorithm.

Among the external subset, we select the node with the lowest distance and we examine it. At this stage, this node is **B**, with distance $d_B = 85$. Its unique neighbor is **F**, for which we compute new distance $d_F = d_B + d_{BF} = 85 + 80 = 165$. Since this new value is lower than the former one (∞), it is assigned to **F**. The current state of the graph is shown in Figure 31.3/right.

What is the status of node **B**? Could we find, later, a better path from **A** to it? No, we couldn't since this hypothetical path should include at least one external node and all of them have a higher distance than that of **B**. So, d_B really is minimal so that **B** can be declared an internal node, which therefore is colored in dark gray (Figure 31.4/left).

Once again, we select the external node with the lowest distance and we examine it. This node is **F**, with distance **165**. It is colored in light gray and its neighbors are updated. Its unique neighbor is **I**, the new distance of which is $d_I = d_F + d_{FI} = 165 + 250 = 415$. Since $415 < \infty$, the distance of **I** is updated (Figure 31.4/left).

The reasoning about the status of **B** can be applied to **F**, which becomes internal and is colored in dark gray (Figure 31.4/right).

In the next iteration, we select and examine external node **E**, which now has the lowest distance (**173**). It is colored in light gray and its unique neighbor, **J**, is updated: $d_J = d_E + d_{EJ} = 173 + 502 = 675$. Since $675 < \infty$, the distance of **J** is updated (Figure 31.4/right).

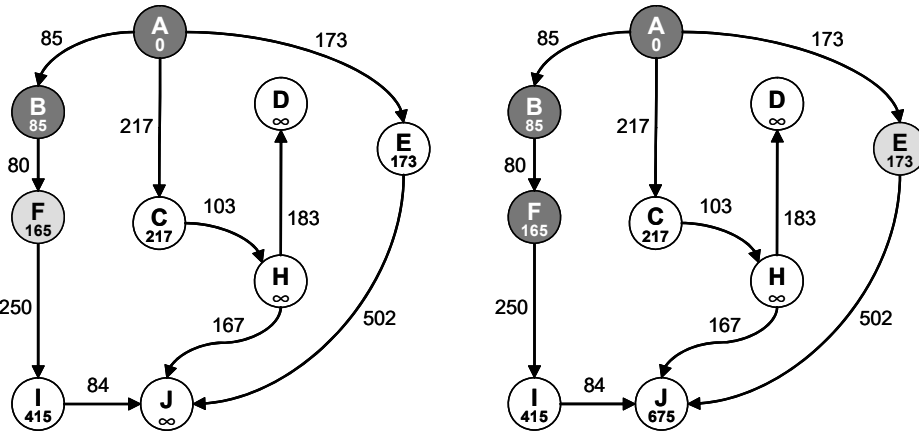


Figure 31.4 - Processing cities **F** and **E**

In the following iterations:

- **E** becomes internal, **C** is selected ($d_C = 217$ is the lowest) and examined, its neighbor **H** is updated with $d_H = d_C + d_{CH} = 320$. Since $320 < \infty$, the distance of **H** is updated (Figure 31.5/left).

- **C** becomes internal, **H** is selected ($d_H = 320$ is the lowest) and examined, its neighbors **D** and **J** are updated. For **J**, $d_J = d_H + d_{HJ} = 487$. Since $487 < 675$, the distance of **J** is replaced (Figure 31.5/right).

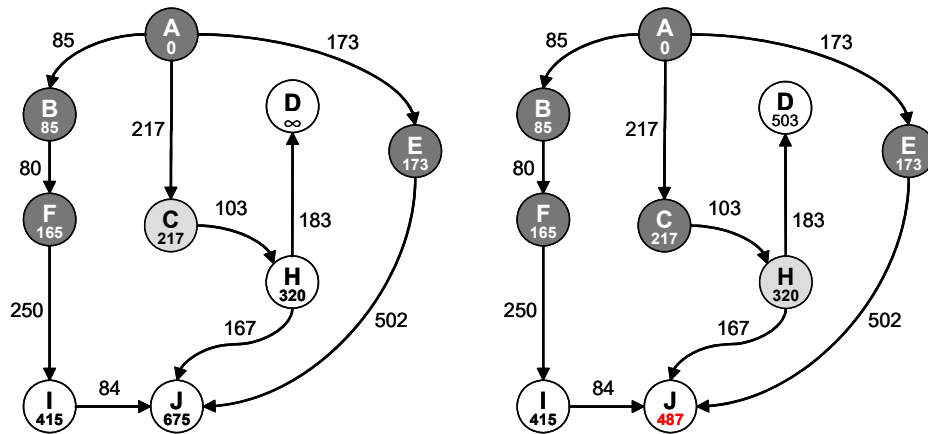


Figure 31.5 - Processing cities **C** and **H**. Improving the path to **J**.

- **H** becomes internal, **I** is selected ($d_I = 415$ is the lowest) and examined, its neighbor **J** is tentatively updated. $d_J = d_I + d_{IJ} = 499$. Since $499 > 487$, the distance of **J** is **not** replaced (Figure 31.6/left).
- **I** becomes internal, **J** is selected ($d_J = 487$ is the lowest) and examined, it has no neighbor so that its examination stops (Figure 31.6/right).

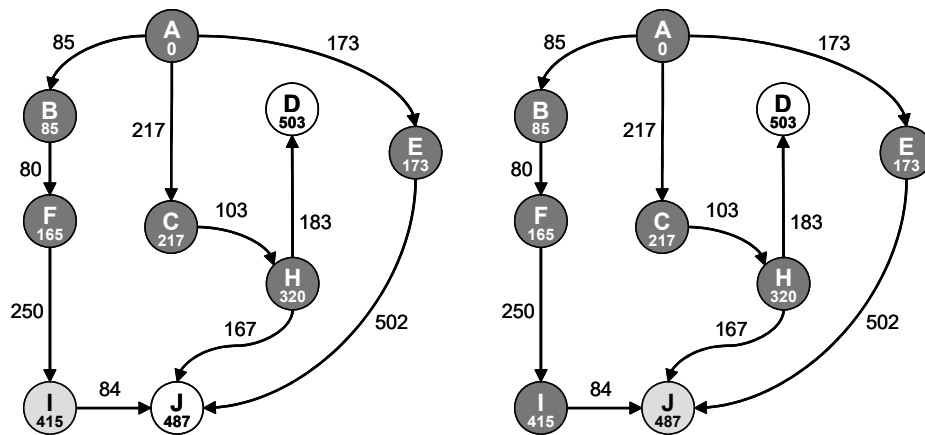


Figure 31.6 - Processing cities **I** and **J**

- **J** becomes internal, the last external node **D** is selected and becomes internal.

The final state is shown in Figure 31.7. The shortest path from **A** to **J** is **A.C.H.J**, with a distance of **487** km.

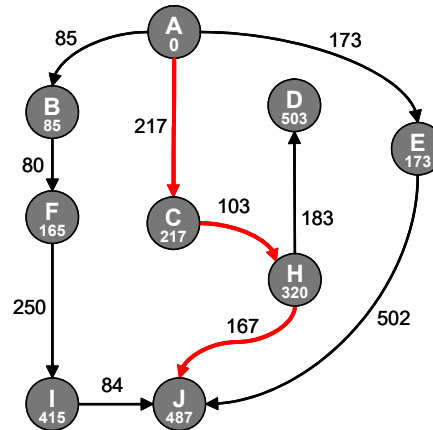


Figure 31.7 - The shortest path from **A** to **J** is **A.C.H.J**

Important notes

It is worth to note that this algorithm still is valid if the graph is not acyclic, that is, includes circuits. It is also valid if it is not directed, i.e., if paths can be followed in both directions.

The algorithm produces the shortest path from the starting city to each of the other cities. Actually, it provides much more! If d_{AK} is the length of the shortest path from **A** to **K**, and if **E** is in this path, then $d_{EK} = d_{AK} - d_{AE}$.

31.3 Representation of the graph

Table CITY stores the description of the nodes (the cities) of the graph: CityID (technical id), Name (city name), Mark (**0** for external nodes and **1** for internal nodes), Path (the current shortest path to this city from the starting city) and Dist (length of the current shortest path).

Tables NPATH and PATH both represents the edges (the paths) of the graph. NPATH expresses them through *city names* while PATH uses *city ID*'s. The first one is more convenient for users while the second one is more efficient for complex calculations (Script 31.1)

To make SQL queries more expressive, we also define views, namely EXTERNAL, that selects external cities, and ExtPATH that provides an extended presentation of PATH (Script 31.2).

Script 31.3 loads the data describing the graph of Figure 31.2.

If the graph is not directed, we create the inverse paths:

```
insert into NPATH(City1, City2, Len)
      select City2, City1, Len from NPATH;
```

The starting state of the graph is defined as follows (999999 denotes *infinity*):

```
update CITY set Mark = 0, Path = '', Dist = 999999;
```

```
create table CITY(CityID integer not null
                  primary key autoincrement,
                  Name   varchar(32) not null,
                  Mark    integer,
                  Path    varchar(256),
                  Dist    integer);

create table NPATH(City1 varchar(32) not null,
                  City2 varchar(32) not null,
                  Len    integer not null);

create table PATH(City1 integer not null,
                  City2 integer not null,
                  Len    integer not null);
```

Script 31.1 - Data structure for City/Path graphs

```
create view EXTERNAL as select * from CITY where Mark = 0;
create view ExtPATH(City1, Name1, Dist1, Len, City2, Name2, Dist2)
as select P.City1, C1.Name, C1.Dist, P.Len,
        P.City2, C2.Name, C2.Dist
   from   CITY as C1, PATH as P, CITY as C2
  where  C1.CityID = P.City1
 and     P.City2 = C2.CityID;
```

Script 31.2 - Complementary views

```
insert into CITY(Name) values ('A'), ('B'), ('C'), ('D'), ('E'),
                              ('F'), ('H'), ('I'), ('J');

insert into NPATH(City1, City2, Len) values
('A', 'B', 85), ('A', 'C', 217), ('A', 'E', 173), ('B', 'F', 80),
('C', 'H', 103), ('F', 'I', 250), ('H', 'D', 183),
('H', 'J', 167), ('E', 'J', 502), ('I', 'J', 84);
```

Script 31.3 - Load the City/Path graph of Figure 31.2

Script 31.4 derives PATH rows from NPATH rows, asks the user the name of the starting city and updates the state of this city.

```
insert into PATH select C1.CityID,C2.CityID,Len
                  from   CITY C1, NPATH P, CITY C2
                  where  C1.Name = P.City1
                  and     P.City2 = C2.Name;

ask start = Starting city:[select Name, CityID
                             from   CITY order by Name];

update CITY set Mark = 0, Dist = 0, Path = Name
where CityID = '$start$';
```

Script 31.4 - Prepare data and select the starting city

31.4 Computing the shortest paths from a starting city

The structure of the algorithm is a loop each iteration of which selects and examines an external city. Since each external city will be selected once and only once, the loop will look as follows:

```
extract N = select count(1) from CITY;
for step = [1,$N$];
  <A. select S = external city with the lowest distance>
  <B. update the neighbors of S>
  <C. make S an internal city>
endfor;
```

A. Selection of S, an external city

We extract the ID of a city from the EXTERNAL view where the distance (column Dist) is equal to the minimum of the distances ($\min(\text{Dist})$) of all the external cities. If more than one city may satisfy this condition, the first one is selected. Hence the query:

```
extract S = select CityID from EXTERNAL
           where Dist = (select min(Dist) from EXTERNAL);
```

B. Updating the neighbors of S

This task is the core of the algorithm. It will be translated into a single SQL update query. To make it more expressive, we first define two shorthands that translate natural concepts:

– the *neighbors* of S:

```
set neighbors = select City2 from PATH where City1 = $$$;
```


- the tentative *new distance* of each neighbor of S:

```
set newDist = (select Dist1 + Len from ExtPATH
               where City1 = $$
               and    City2 = CITY.CityID);
```

This part of the update query computes the distance of the neighbor city being updated, denoted by `CITY.CityID`. `Dist1` is the distance of the selected external city and `Len` is the length of the path between these two cities.

The updating can then be written very naturally, as follows:

```
update CITY
set    Dist = min(Dist,$newDist$)
where  CityID in ($neighbors$);
```

Recording the paths

So far, we have computed the *shortest distance* between two cities but we have not recorded the intermediate cities that constitute this *shortest path*. When we update a neighbor, if the new distance is lesser than its current distance, we also replace the current path of the neighbor by that of the selected external city to which we append the name of the neighbor:

```
update CITY
      Path = case when Dist > $newDist$
                then  (select Path from CITY
                       where CityID = $$) || '.' || Name
                else  Path
            end
where  CityID in ($neighbors$);
```

By merging both update queries we get:

```
update CITY
set    Dist = min(Dist,$newDist$),
      Path = case when Dist > $newDist$
                then  (select Path from CITY
                       where CityID = $$) || '.' || Name
                else  Path
            end
where  CityID in ($neighbors$);
```

C. Making S an internal city

The SQL expression is straightforward:

```
update CITY set Mark = 1 where CityID = $$;
```

The complete algorithm is shown in Script 31.5.

```

extract N = select count(1) from CITY;
for step = [1,$N$];
    extract S = select CityID from EXTERNAL
                where Dist = (select min(Dist)
                             from EXTERNAL) #1;

    set neighbors = select City2 from PATH
                    where City1 = $$;

    set newDist = (select Dist1 + Len from ExtPATH
                  where City1 = $$
                    and City2 = CITY.CityID);

    update CITY
    set Dist = min(Dist,$newDist$),
        Path = case when Dist > $newDist$
                    then (select Path from CITY
                          where CityID = $$) || '.' || Name
                    else Path
                    end
    where CityID in ($neighbors$);

    update CITY set Mark = 1 where CityID = $$;
endfor;

```

Script 31.5 - Computing the shortest paths of cities from a starting city

The execution of this script for the problem depicted in Figure 31.1 provides this solution:

CityID	Name	Mark	Path	Dist
1	A	1	A	0
2	B	1	A.B	85
3	C	1	A.C	217
4	D	1	A.C.H.D	503
5	E	1	A.E	173
6	F	1	A.B.F	165
7	G	1	A.C.G	403
8	H	1	A.C.H	320
9	I	1	A.B.F.I	415
10	J	1	A.C.H.J	487

If we are only interested by the shortest path between two definite nodes, we can stop the iterations and exit the loop once the target city has been made internal.

31.5 Building a general purpose *shortest path engine*

Joining two cities in an efficient way is just one specific example of a more general family of optimization problems that can be modelled by a graph. Cities are **nodes** of the graph while roads are directed or undirected **paths** between nodes. The length associated with a road can be generalized as the **cost** of the path. Finally, the distance of a city from the starting city (we will call it the **starting node**) also is its **cost**.

Now, we can write a set of procedures with which one can easily solve any problem pertaining to the *shortest path* family.

This set includes generic, problem-independent, procedures:

- **__Shortest-path_DB.sql**: creates tables NODE, PATH and NPATH and the views of the database. See Script 31.6.
- **__Shortest-path_Engine.sql**: applies Dijkstra algorithm to the problem stored in the database. See Script 31.7.
- **__Display_Shortest-path.sql**: displays the solution of the problem. See Script 31.8.
- **__Display_Nodes.sql**: displays the final state of table NODE (which includes the solution). See Script 31.9.

In addition, for any type of problems **X**, we have to write two scripts:

- **_X_Loader.sql**: loads in tables NODE and NPATH the data describing the specific problem **X**, creates, if needed, the inverse paths then derives the rows of table PATH.
- **X_Solver.sql**: main script used to solve any problem of type **X**; each of them has the minimal structure:

```
createDB InMemory;
execSQL __Shortest-path_DB.sql;
execSQL _X_Loader.sql;
execSQL __Shortest-path_Engine.sql
execSQL __Display_shortest-path.sql;
closeDB;
```

The script that computes the shortest path between two cities can then be rewritten as Scripts 31.10 and 31.11.

```

create table NODE (NodeID integer not null
                    primary key autoincrement,
                    Name   varchar(32) not null,
                    Mark   integer,
                    Path   varchar(256),
                    Cost   numeric);

create table NPATH(Node1 varchar(32) not null,
                  Node2  varchar(32) not null,
                  Cost   numeric not null);

create table PATH( Node1 integer not null,
                  Node2  integer not null,
                  Cost   numeric not null);

create view EXTERNAL as select * from NODE where Mark = 0;
create view ExtPATH(Node1,Name1,Cost1,Cost,Node2,Name2,Cost2)
as select P.Node1,C1.Name,C1.Cost,P.Cost,
        P.Node2,C2.Name,C2.Cost
from     NODE as C1, PATH as P, NODE as C2
where    C1.NodeID = P.Node1
and      P.Node2 = C2.NodeID;

```

Script 31.6 - Generic data structures for shortest path problems [script file _Shortest-path_DB.sql]

```

extract N = select count(1) from NODE;
for step = [1,$N$];
    extract S = select NodeID from EXTERNAL
                where Cost = (select min(Cost)
                             from EXTERNAL);

    set neighbors = select Node2 from PATH where Node1 = $$;
    set newCost = (select Cost1 + Cost from ExtPATH
                  where Node1 = $$ and Node2 = NODE.NodeID);

    update NODE
    set     Cost = min(Cost,$newCost$),
           Path = case when Cost > $newCost$
                     then (select Path from NODE
                           where NodeID = $$) || '.' || Name
                     else Path
           end
    where  NodeID in ($neighbors$);

    update NODE set Mark = 1 where NodeID = $$;
endfor;

```

Script 31.7 - The shortest path algorithm [script file _Shortest-path_Engine.sql]

```

if ('$end$' = '');
    write-ab -- No target node has been specified.;
    return;
endif;

extract cost,path = select Cost,Path from NODE
                    where NodeID = '$end$';
extract nam1 = select Name from NODE where NodeID = '$start$';
extract nam2 = select Name from NODE where NodeID = '$end$';
write-b -- Shortest path from "$nam1$" to "$nam2$": $path$;
write-a -- Cost of this path = $cost$.

```

Script 31.8 - Display the solution of the problem [script file `_Display_Shortest-path.sql`]

```

set maxSelect = $maxSelectWidth$;
extract maxSelectWidth = select max(length(Path))+34
                        from NODE;
select Name as Node,Path,Cost from NODE;
set maxSelectWidth = $maxSelect$;

```

Script 31.9 - Display the final state of table NODE [script file `_Display_Nodes.sql`]

```

insert into NODE(Name) values ('A'),('B'),('C'),('D'),('E'),
                              ('F'),('G'),('H'),('I'),('J');
insert into NPATH values
    ('A','B',85), ('A','C',217), ('A','E',173), ('B','F',80),
    ('C','G',186), ('C','H',103), ('F','I',250), ('H','D',183),
    ('H','J',167), ('E','J',502), ('I','J',84);
insert into PATH select C1.NodeID, C2.NodeID, P.Cost
                  from NODE C1, NPATH P, NODE C2
                  where C1.Name = P.Node1
                  and P.Node2 = C2.Name;
update NODE set Mark = 0, Path = '', Cost = 999999;

```

Script 31.10 - Loading the data for the Inter-City problem depicted in Figure 31.1 [script file `_Inter-City_Loader#1.sql`]

```

set script =
  $scriptDirectory$/Case-Studies/Case_Shortest_Path;
createDB InMemory;
execSQL $script$/_Shortest-path_DB.sql;
execSQL $script$/_Inter-City_Loader#1.sql;
ask start,end = Starting node:[select Name,NodeID
                                from   NODE order by Name]
|Target node:[select Name,NodeID
                  from   NODE order by Name];
  if ('$DIALOGbutton$' = 'Cancel' or '$start$' = '') stop;
update NODE set Mark = 0, Cost = 0, Path = Name
where NodeID = '$start$';
execSQL $script$/_Shortest-path_Engine.sql;
execSQL $script$/_Display_Shortest-path.sql;
closeDB;

```

Script 31.11 - The solver for Inter-City problems [script file **Inter-City_Solver.sql**]

31.6 Ariadne's thread: solving mazes

Mazes are popular brain puzzles that ask players to walk across a convoluted path from an entry gate to an exit gate. In many of them, there is only one route from these gates, a pattern called *unicursal maze* or *labyrinth*. Multicursal mazes allow more than one route between gates.

The small multicursal maze of Figure 31.8 comprises two distinct start-to-exit routes, depicted by lines red and blue. The first one (red) requires **20** steps (a step is a unitary square room) while the last one (blue) is **18** step long, and therefore is the shortest one.

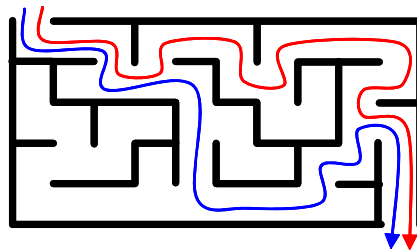


Figure 31.8 - A typical multicursal maze²

2. Derived from <https://en.wikipedia.org/wiki/Maze>

A maze can be described in two equivalent ways. On the one hand, one shows its **walls**, between which the route to find will be drawn. This is the usual way mazes are described (Figure 31.9/left). The second convention consists in showing the available **paths** the maze is made up of, i.e., the space between walls (Figure 31.9/right). We will reason on this latter representation.

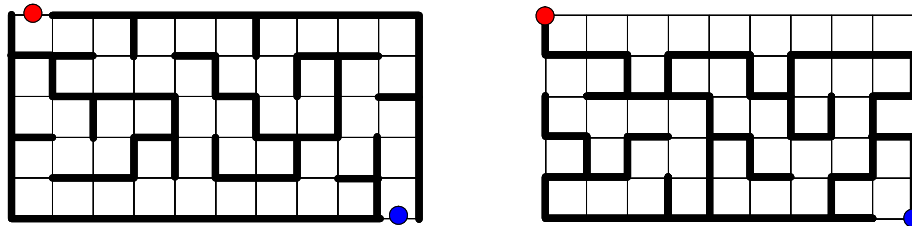


Figure 31.9 - Dual representations of a maze: walls (left) and paths (right)

Considering the grid of Figure 31.10, in which each node is denoted by its row and column identifiers (e.g., B8, E2, etc.), building a maze is simply performed by marking the edges that belong to the maze (Figure 31.9/right).

In this way, solving a maze appears to be a special case of finding the shortest path between cities. The nodes of the grid are some sort of *cities* while the edges are *roads* between cities, all with the same arbitrary *length*, say, 1.

	1	2	3	4	5	6	7	8	9	10
A										
B										
C										
D										
E										
F										

Figure 31.10 - The background grid of a maze

A real maze will comprise quite a lot of nodes and edges. To help players to build them, we suggest to describe them in a text file (named **Maze_map.txt**) where a node is represented by star symbol '*', a horizontal edges by '-' and a vertical edge by symbol '|' (Figure 31.11).

This file is then transformed into a script made up of `insert` queries that load the description of the maze into tables `NODE`, `NNPATH` and `PATH`. This transformation is performed by small Python program **Generate-Maze.py**,³ which generates file **Maze_Loader.sql**.

3. Its translation into an SQLfast script is left as an exercise.

```

* * * * *
|
*--*--* *--*--* *--*--*
|
* *--*--*--* *--* *--*
|
*--* *--* *--* *--* *--*
|
*--*--* * *--* *--* *
|
*--*--*--*--*--*--*--*

```

Figure 31.11 - Code of a maze [file **Maze_map.txt**]

```

set script =
  $scriptDirectory$/Case-Studies/Case_Shortest_Path;
createDB InMemory;
execSQL $script$/_Shortest-path_DB.sql;
execSQL $script$/Maze_Loader.sql;
ask start,end = Starting node:[select Name,NodeID
                                from   NODE order by Name]
|Target node:[select Name,NodeID
                  from   NODE order by Name];
if ('$DIALOGbutton$' = 'Cancel' or '$start$' = '') stop;
update NODE set Mark = 0, Cost = 0, Path = Name
where NodeID = '$start$';
execSQL $script$/_Shortest-path_Engine.sql;
execSQL $script$/_Display_Shortest-path.sql;
closeDB;

```

Script 31.12 - The maze solver [script file **Maze_Solver.sql**]

The larger maze of Figure 31.12 is defined in text file **Maze_map_Large.txt**.

Some references on maze algorithm

Some of the main maze solving techniques are described in https://en.wikipedia.org/wiki/Maze_solving_algorithm. The *shortest path* algorithm is briefly mentioned, but does not refer to Dijkstra's algorithm explicitly.

Generating maze is also quite interesting. Reference https://en.wikipedia.org/wiki/Maze_generation_algorithm can be used as a starting point. In particular, it provides Python programs to generate mazes. Cellular automata (see Chapter 32, *Conway's*

Game of Life) with special generation rules can be used to create large mazes. They are described in <http://www.conwaylife.com/w/index.php?title=Maze>.

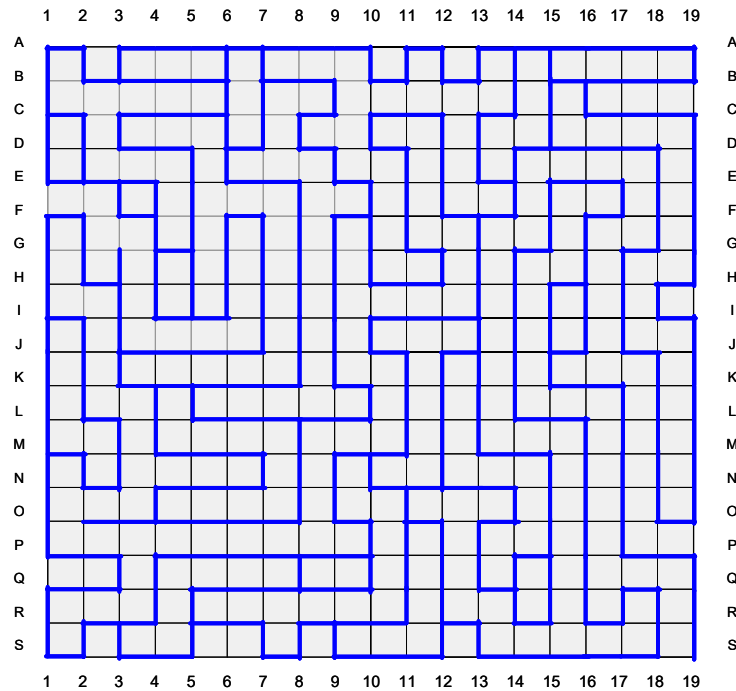


Figure 31.12 - A real size maze (text file **Maze_map_Large.txt**)

31.7 Exploring a planet

Let us imagine that we are to control an exploratory vehicle, sent on a remote planet, to make it move from a starting position to a target position in a hostile 2D space. This vehicle, generally called a *rover*, obeys elementary commands, which are to move one step in one of the four cardinal directions: N, S, E and W. The floor itself is modeled as a regular grid like that of Figure 31.13/left.

In this example, the rover starts at position A5 (the red point) and has to reach target position J8 (the blue point). Each step consumes a unitary electrical power, say, 1 Wh (one watt during one hour). Since the power of the rover is limited, it would be nice to guide the rover so that it consumes the lowest possible energy to accomplish its task.

Clearly, this problem is fairly close to that of the shortest path between cities: the nodes of the grid are kind of cities (named A1, A2, ..., J9, J10) and steps are roads between these cities, with unitary cost.

To make things more realistic, we consider that the rover may have to move across swamp areas that will require more power. Let us say that a step in such an area will cost 3 units. In addition, the floor is scattered with rocks that force the rover to go around them to progress to the target. The grid of Figure 31.13/right shows where the swamp areas (light gray) and the rocks (solid dark gray) are located.

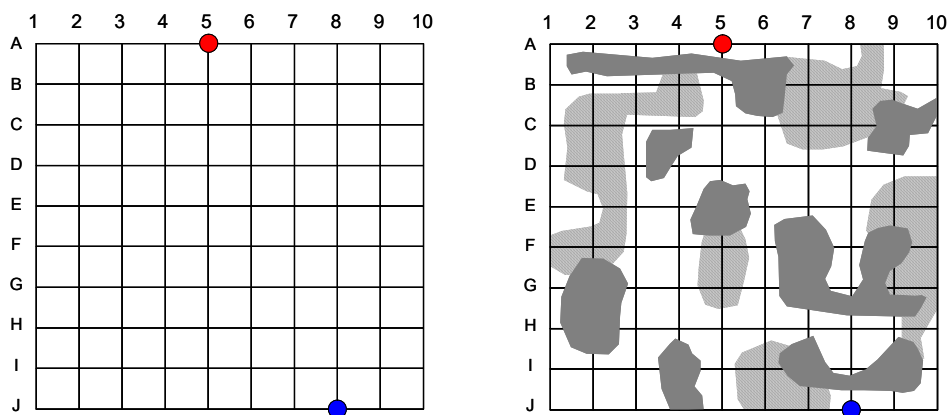


Figure 31.13 - The remote planet the rover has to explore is not as easy as planet earth!

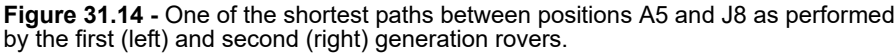
How to represent rocks, which are forbidden areas? The most obvious technique would consist in deleting all the paths partially or completely covered by a rock, as we did in solving the maze problem. We can also keep these paths, to which we assign an infinite cost, that is, practically, 999,999 units, so that they will never be chosen.

The solution for starting position A5 to target position J8 is shown in Figure 31.14/left. Its cost is 26 Wh.

Now, let us consider the second generation rover, which is able to climb rocks securely, but at the higher cost of 6 Wh per rock step. Our algorithm finds another, more sober, path, shown in Figure 31.14/right. Its cost is 22 Wh.

Script 31.13 shows an example of solver for rover problems. The unit cost of plain floor, swamp and rock can be chosen interactively. The script `Rover-on-Planet_Loader.sql` loads the data for the planet model of Figure 31.13/right.

As we can expect, most references on controlling rovers are classified by the NASA, and therefore are not available. Nevertheless, the reader can find some interesting hints in this reference: <https://en.wikipedia.org/wiki/Pathfinding>. In particular, it discusses pathfinding algorithms in video games.



```
set scripts = SQLfast-Tutorials/Chapter-52;
createDB InMemory;
execSQL $scripts$/_Shortest-path_DB.sql;
set c1 = 1;
set c2 = 3;
set c3 = 999999;
ask-u c1,c2,c3 = Cost of normal step:
                |Cost of swamp step:
                |Cost of rock step;;

execSQL $scripts$/Rover-on-Planet_Loader.sql;
ask start,end = Starting node:[select Name,NodeID
                                from   NODE order by Name]
                |Target node:[select Name,NodeID
                                from   NODE order by Name];
if ('$DIALOGbutton$' = 'Cancel' or '$start$' = '') stop;
update NODE set Mark = 0, Cost = 0, Path = Name
where NodeID = '$start$';
execSQL $scripts$/_Shortest-path_Engine.sql;
execSQL $scripts$/_Display_Shortest-path.sql;
closeDB;
```

Script 31.13 - The solver for rover problems [script file Rover-on-Planet_Solver.sql]

31.8 Performance and optimization

Finding real country maps comprising a large number of cities **AND** in which the lengths of the roads between them are specified is no easy task. To evaluate the performance of the algorithm developed above we will rely on artificial maps generated as square grids like those used to model mazes. Each node represents a city and each city is linked to its four closest neighbors (except border cities of course). So, an $N \times N$ grid represents a map with N^2 cities and $4 \times N \times (N - 1)$ roads.

The goal is to compute the shortest paths from the top left city to all the other cities. Figure 31.15 shows the time needed to get the result for increasing values of N (column **Time1**).

N	Cities	Roads	Time1	Time2	Time3
10	100	360	0.124	0.109	0.001
20	400	1520	0.624	0.468	0.156
30	900	3480	1.872	1.123	0.812
40	1600	6240	4.446	2.215	2.558
50	2500	9800	9.236	3.776	6.240
60	3600	14160	17.175	6.048	13.010

Figure 31.15 - Computing time (in seconds) for various map sizes

Column **Time2** shows much better results. They are obtained by indexing table **PATH** on column (Node1). This index speeds up the evaluation of expressions **neighbors** and **newCost**.

Third column **Time3** shows the execution time of the shortest-path engine coded in Python (program **Shortest_Path.py**). Its times are the best for small problems but become, by far, the worst for large values of N .

Figure 31.16 represents graphically the execution time of these three techniques, respectively named Plain (**Time1**), Index (**Time2**) and Python (**Time3**). The fourth curve translates the theoretical law $N \log N$. It allows to observe that the result of the indexed technique is quite close to the theoretical performance of Dijkstra's algorithm, the complexity of which is $O(N \log N)$.⁴

To test other grid sizes from $N = 4$ to 64, use predefined script **Large-grid-maps_Solver.sql**.

4. The *complexity* of an algorithm is a measure of the limit of the behaviour of its execution time (or any other resource it needs, such as memory space) when the size of the problem (here N) increases. Writing that the execution time of Dijkstra's algorithms varies as $N \log N$ means that if the execution time is T for a certain value of N , then, for a grid of $2N$, the execution time will be a bit higher than $2T$ (linear) but much better than $4T$ (quadratic).

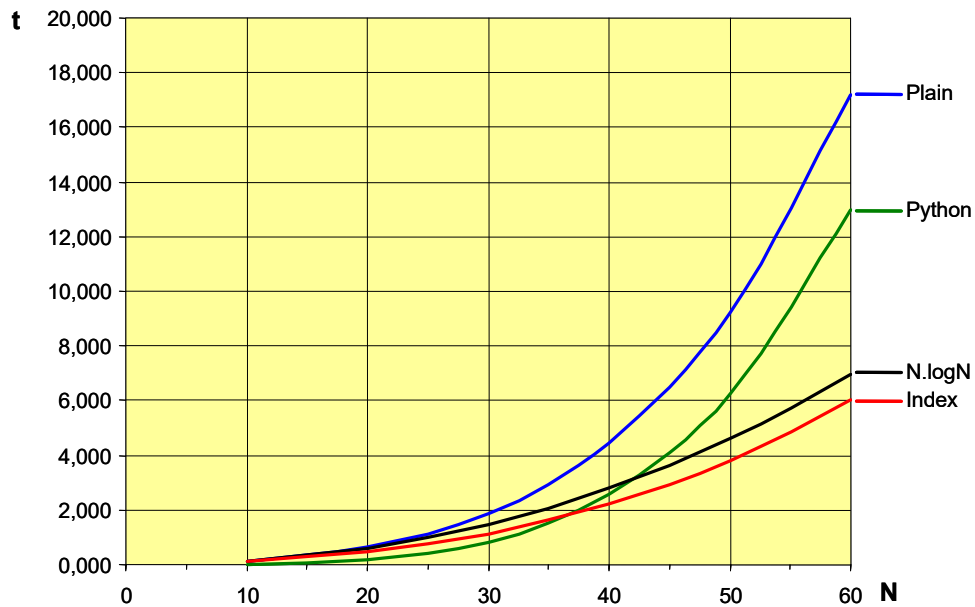


Figure 31.16 - Comparison of the performance of various algorithms

31.9 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Case_Shortest_Path**. They can be run from main script **Shortest-MAIN.sql**, that displays the selection box of Figure 31.17.

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

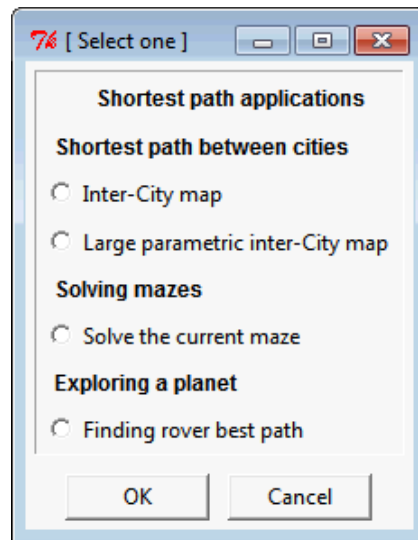


Figure 31.17 - Selecting and solving a best path problem

