Notes 7: Dynamic programming and Dijkstra's algorithm

7.1 Optimal substructure and memoization

We have seen in Sec. 2.1 how the solution of certain problems can be expressed as a combination the solutions of smaller copies of the same problem. This led us to the divide-and-conquer paradigm which allowed us to write down some very succinct algorithms for solving problems like sorting, searching, computing Fourier transforms etc. A problem that can be solved optimally by breaking it into subproblems and then recursively finding the optimal solutions to the subproblems is said to have optimal substructure.

Although divide-and-conquer algorithms are often very concise in terms of coding, for certain types of problems they can be extremely inefficient in terms of run time. This is particularly the case if the recursive subdivision into smaller and smaller problems frequently generates repeated instances of the same problem applied to the same data. When two or more subproblems generated by a recursive subdivision of a larger problem end up applying the same operations to the same input data, the sub-problems are said to be overlapping. Divide-and-conquer algorithms work best when all subproblems are non-overlapping. When a significant number of subproblems are overlapping, it can be more efficient to adopt a strategy in which we solve each subproblem only once, store the result and then look up the answer when the same subproblem recurs later in the calculation. The process of storing results of certain calculations which occur frequently in order to speed up a larger calculation is termed memoization. Memoization is an important tool in the design of algorithms which allows the programmer to trade memory for speed.

To give a concrete example, consider the following recursive Python function to compute the n^{th} Fibonacci number:

```
# Function to compute Fibonacci numbers by naive recursion
def F(n):
   if n == 1 or  n ==0 :
      return n
   else:
      return F(n-1) + F(n-2)
```

At this point, it should be obvious how and why it works. Let us assess the efficiency of this function. Using the same arguments as before, the computational complexity, $F(n)$ satisfies the recursion relation

$$F(n) = F(n-1) + F(n-2)$$

with starting conditions $F(0) = 1$ and $F(1) = 1$. Starting from the ansatz $F(n) = x^n$ and solving the resulting quadratic equation for x, the solution to this recursion relation which satisfies the starting conditions is

$$F(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right],$$

which clearly grows exponentially as n increases. At first sight this might seem surprising since the obvious iterative algorithm for computing Fibonacci numbers is clearly $O(n)$. The reason for the poor scaling is that the recursive function $F(n)$ generates an exponentially increasing number of overlapping subproblems as the recursion proceeds. To see this consider the sequence of function

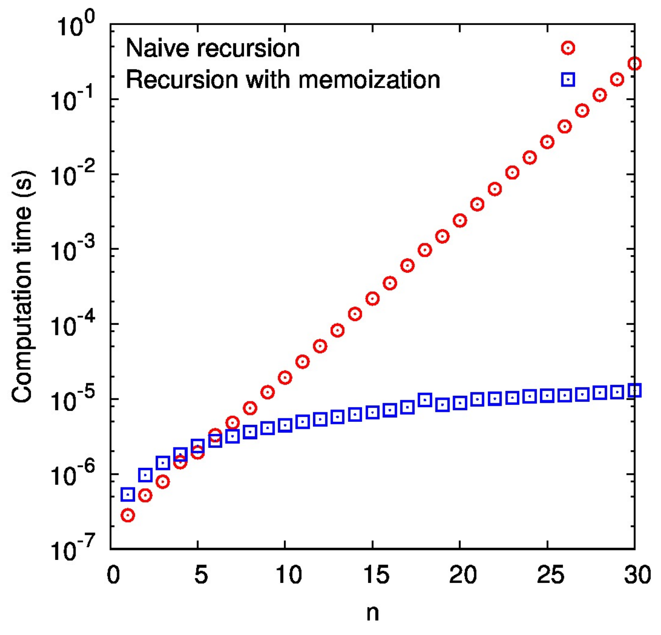


Figure 7.1: Lin-Log plot showing the time required to compute the n^{th} Fibonacci number using the two algorithms described in the text.

calls which are made when computing $F(5)$:

$$\begin{aligned}
F(5) &= F(4)+F(3)\\
&= (F(3)+F(2))+(F(2)+F(1))\\
&= ((F(2)+F(1))+(F(1)+F(0)))+((F(1)+F(0))+F(1))\\
&= (((F(1)+F(0))+F(1))+(F(1)+F(0)))+((F(1)+F(0))+F(1)).
\end{aligned}$$

Sifting through this execution stack we see, for example, that $F(2)$ was computed 3 times and $F(3)$ was computed twice. As n grows the duplication of sub-problems becomes sufficiently frequent to result in an exponential time algorithm.

Let us now consider how memoization can be used to greatly speed up this algorithm by storing the results of computations in memory and then looking up the answer when the same computation occurs again later. In order to store the intermediate results, we use a special data structure known as an associative array [1] (called memo in the Python code below). An associative array is a data structure consisting of a set of key-value pairs where each allowed key appears at most once and which supports the operations of addition, removal and modification of pairs in addition to a lookup operation on the keys. In our case the keys will be integers, n, and the corresponding values will be the Fibonacci numbers associated with these integers. Before calling the function, the associative array memo is initialised with the pairs $0:1$ and $1:1$ corresponding to the first two Fibonacci numbers (the base cases in the recursive version above). Here is the code:

```
# Function to compute Fibonacci numbers by recursion with memoization

memo = {0:0, 1:1}

def F2(n):
   if n not in memo :
      memo[n] = F2(n-1) + F2(n-2)
   return memo[n]
```

This is deceptively simple. If the key n is already stored it simply looks up the corresponding value and returns it. If not, it computes it (using the same recursion as before), stores the result for future use and returns the result. The run times of the two versions of the Fibonacci code are plotted as a function of n in Fig. 7.1. The memoization version of the code actually runs in $O(n)$ time. This is because each sub-problem is computed only once.

In reality, one would not use either of these methods to compute Fibonacci numbers. This exercise is purely pedagogical and intended to illustrate the fact that intelligently reusing previously computed results can lead to hug efficiency gains in some problems. This is the feature which generalises far beyond this simple example.

7.2 Dynamic programming

A common class of problems involves making sequences of decisions, each having a known cost or benefit, such that the total cost of the sequence is minimized (or the total benefit is maximised). Such problems often exhibit optimal substructure and are amenable to a solution technique called dynamic programming (DP) which tries to take advantage of overlapping subproblems and solve each subproblem only once.

Consider for example the following toy problem (the checkerboard problem). Consider a checkerboard with $n \times n$ squares and a cost-function $c(i, j)$ which returns a positive cost associated with square i, j (i being the row, j being the column). Here is an example of the 5×5 case. Row and column indices are in bold on the left and bottom rows and costs, $c(i, j)$, are in the interior of the table.

5	2	1	4	10	10
4	3	5	1	4	1
3	2	3	7	3	1
2	4	2	3	2	1
1	0	0	0	0	0
	1	**2**	**3**	**4**	**5**

The checker can start anywhere in row 1 and is only allowed to make the usual moves:

5					
4					
3					
2		X	X	X	
1			O		
	1	**2**	**3**	**4**	**5**

The objective is to find the sequence of moves which minimises the total cost of reaching row n. One strategy would be to choose the move which has the smallest cost at each step. Such a strategy of making locally optimal moves in the hope of finding a global optimum is called a greedy algorithm. Greedy algorithms are usually not optimal but are often used as cheap and easy ways of getting "not-too-bad" solutions. The above table of costs has been designed to set a trap for a greedy strategy to illustrate how locally optimal moves need not necessarily lead to optimal or even near-optimal solutions.

This problem exhibits optimal substructure. That is, the solution to the entire problem relies on solutions to subproblems. Let us define a function $q(i, j)$ as the minimum cost to reach square (i, j). If we can find the values of this function for all the squares at rank n, we pick the minimum and follow that path backwards to get the optimal path. Clearly that $q(i, j)$ is equal to the minimum cost to get to any of the three squares below it plus $c(i, j)$. Mathematically:

$$q(i,j) = \begin{cases} \infty & \text{if } j < 1 \text{ or } j > n \\ 0 & \text{if } j = 1 \\ \min\{q(i-1,j-1), q(i-1,j), q(i-1,j+1)\} + c(i,j) & \text{otherwise} \end{cases} \tag{7.1}$$

The first line is there to make it easier to write the function recursively (no special cases needed for edges):

```
def q(i, j):
    if j < 1 or j > n:
        return infinity
    else if i == 1:
        return c(i, j)
    else:
        return min( q(i-1, j-1), q(i-1, j), q(i-1, j+1) ) + c(i, j)
```

This function will work although, like the Fibonacci example above, it is very slow for large n since recompute the same shortest paths over and over. We can compute it much faster in a bottom-up fashion if we store path-costs in a two-dimensional array q[i, j] rather than using a function thus avoiding recomputation. Before computing the cost of a path, we check the array q[i, j] to see if the path cost is already there. In fact, if you had not learned about recursion, this is probably how you would have instinctively coded up this problem anyway!

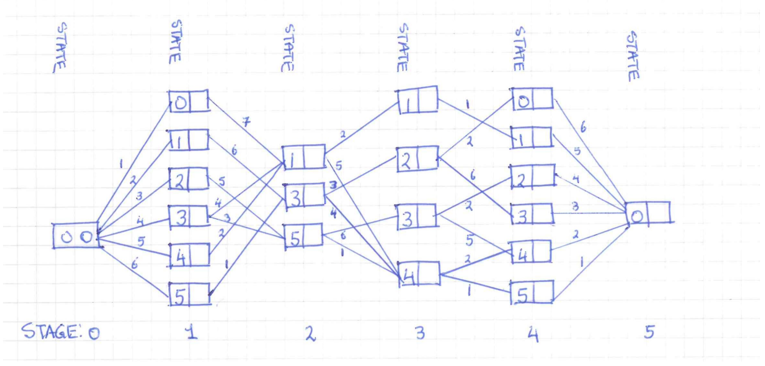
```
def computeMinimumCosts():
   for i from 1 to n
      q[1, i] = 0
   for i from 1 to n
         q[i, 0]     = infinity
         q[i, n + 1] = infinity
   for i from 2 to n
      for j from 1 to n
         m := min(q[i-1, j-1], q[i-1, j], q[i-1, j+1])
         q[i, j] := m + c(i, j)
```

Here is (I think) the contents of the array q[i,j] after running this function:

5	9	6	9	13	13
4	7	9	5	6	3
3	4	7	9	4	2
2	4	2	3	2	1
1	0	0	0	0	0
	1	**2**	**3**	**4**	**5**

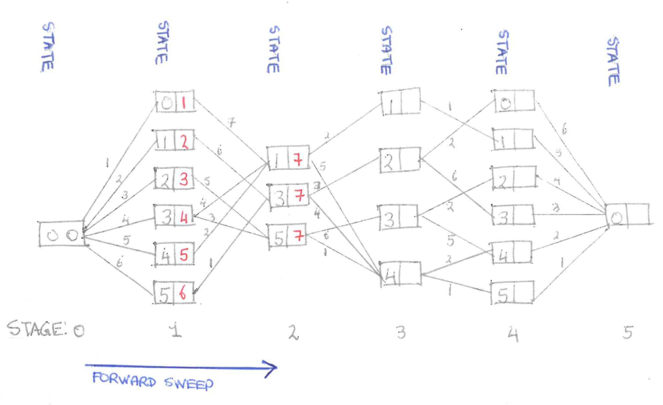
The optimal cost is therefore 6 and the optimal path can be reconstructed by reading backwards through the set of predecessors of the optimal solution. This little problem illustrates the two key features of DP: a forward sweep which stores at each node the cost of the best possible path to reach that node followed by a backtracking stage which follows the precursors of the final optimal state back through the different stages to reconstruct the optimal path. Note that the sequence constructed from DP is globally optimal.

More generally, a problem is amenable to dynamic programming if it can be broken up into a number of discrete stages and within each stage into a number of discrete states. Transitions occur in a directed fashion from states in earlier stages to states in later stages and each transition has an associated cost. The objective is to find the sequence of states going from a particular starting state, i_0, at stage 0 to a particular ending state, i_N, at stage N which minimises the total cost. Such problems are most naturally represented as a weighted directed graph with stages arranged from left to right and edges representing allowed transitions between states in consecutive stages. This is best represented as a picture:

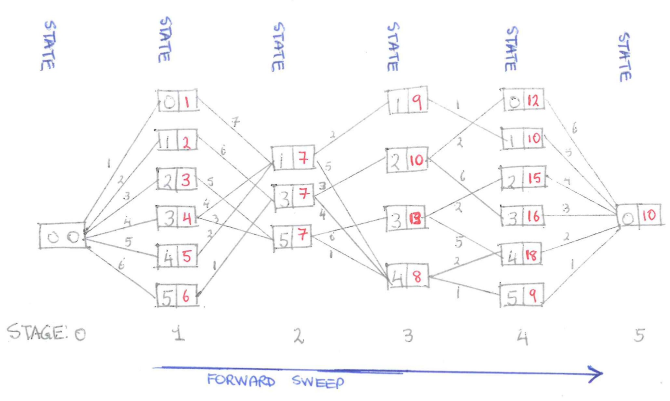


The weight associated with an edge linking state j in stage i to state k in stage $i+1$ is denoted by $w_{jk}(i)$ (just written as integers near the associated edges in the picture). To make this concrete, in the checkerboard example, the stages are the rows, the states are the columns, the edges are the allowed moves from row i to row $i+1$ and the weight of a given edge is the value of $c(i,j)$ on the square at which the move terminates.

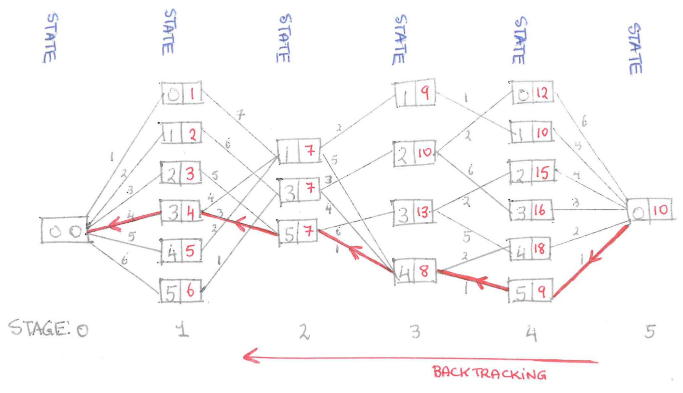
The problem is solved by first making a forward sweep through the graph from left to right. As the sweep proceeds, we label each vertex with a single integer which is the cost of the best way to have reached it:



When the end state is reached, the globally minimal cost becomes known:



A back-tracking pass can then be made, tracing back the set of edges which were followed in order to reach this optimal value. The result is a reconstruction of the optimal path:



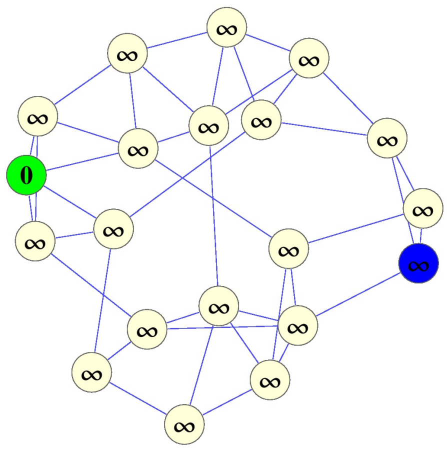
In this case, the optimal path turns out to be $0 \to 3 \to 5 \to 4 \to 5 \to 0$.

7.3 Dijkstra's shortest path algorithm

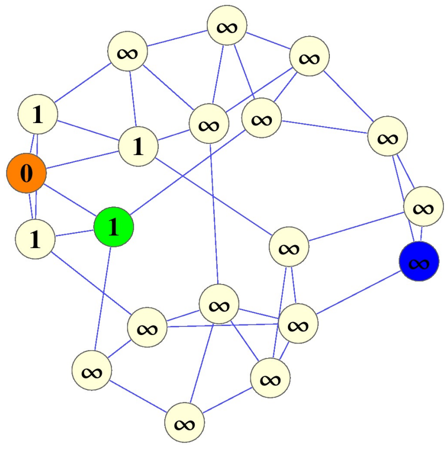
As discussed above it is clear that the problem of finding the shortest path between two nodes on a graph is of fundamental importance to solving DP problems. This is not a problem which is amenable to brute force enumeration since the number of possible paths between two vertices of a graph generally grows exponentially with the number of vertices. There are huge numbers of overlapping subproblems however. This is because if a sequence of vertices $p = \{v_1, v_2, \ldots, v_n\}$ is a shortest path between v_1 and v_n then every contiguous subsequence of vertices of p is itself a shortest path.

Dijkstra's algorithm is the fundamental method of solution for this problem. The idea is to do a sweep through the vertices of the graph, starting at the source node and maintaining a set of labels on each vertex indicating the shortest way to reach it from the source. The algorithm exploits the overlapping subproblems using the fact that if p_1 is a shortest path from v_1 to v_n and p_2 is a shortest path from v_n to v_m then the concatenation of p_2 with p_1 is a shortest path form v_1 to v_m. As a result each vertex is only visited once. We do not assume that the graph is directed or has the stage structure in the DP example above.

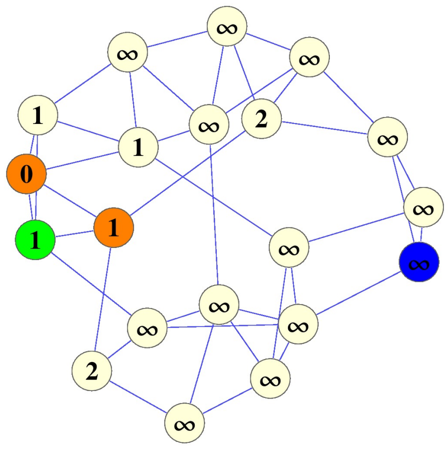
We shall illustrate the procedure in pictures before writing a more formal description of the algorithm. Consider the following graph with 20 nodes:



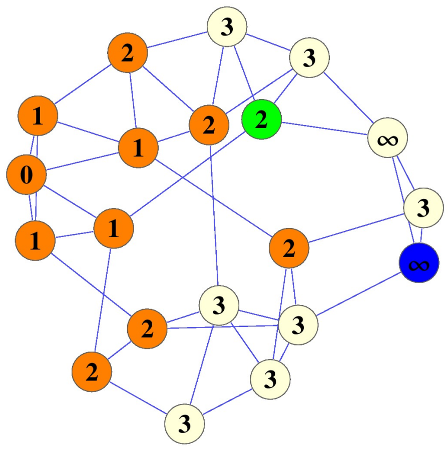
The target node is indicated in blue. All vertices except the source are initially labeled with a distance of ∞ from the source. In this context, ∞ indicates that the distance is not yet known. The source vertex is labeled with a distance of 0. All vertices are initially marked as unvisited (white). To start the search, the current active vertex (indicated in green) is set equal to the source vertex.



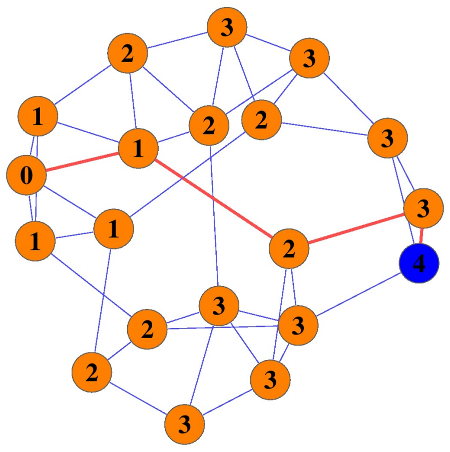
At the first step, the distances of the neighbours of the current active vertex are examined. All of them currently have unknown distance from the source. Since they are neighbours of the current active vertex, there exists a (necessarily shortest) path of length 1 from the current active vertex to each of them and since the current active vertex is a distance 0 from the source, we can set all of distance labels to 1. We then mark the source node as visited (orange) and select the new active vertex to be one of the neighbours of the source. The general rule is to choose the new active vertex to be one of the unvisited vertices whose distance to the source is a minimum.



After two steps, several paths of length 2 have been identified and the active vertex has moved on to the next unvisited vertex at distance 1. The distance labels of the unvisited neighbours of the current active vertex are updated whenever a new path is found which is shorter than the value stored in the current label. The current active vertex is then marked as visited and the new active vertex set equal to one of the unvisited vertices whose distance to the source is a minimum.



After many steps of the procedure, lots of vertices are orange (visited) and labeled with the length of the shortest distance back to the source. The efficiency of the algorithms stems from the fact that it is never necessary to revisit the orange vertices.



In this case, the algorithm is "unlucky": the target vertex is not found until the very last move: 19 intermediate steps were required with the target finally found on the 20th step. Shortest paths like the red one can now be read of by working backwards from the target towards the source. From any visited vertex it is always possible to find a neighbour which is one step nearer to the source. There may be several such neighbours however. In this example, there are two neighbours of the target at distance 3, each of which leads to a valid shortest path.

Here is some pseudocode for a simple version of Dijkstra's algorithm which finds the length of the shortest path between a source and target vertex in a graph G. It is assumed that G is fully connected so that at least one shortest path exists between any two vertices.

```
input  : Graph G
input  : vertex source
input  : vertex target
// Initialisations
dist [source ] = 0 ;                         // Source is distance 0 from itself
for vertex $v \in G$ do
    if $v! =$ source then
        dist[v] = $\infty$;                              // $\infty$ here means unknown
    end
    add v to unvisitedQ ;                  // maintain list of unvisited vertices
end
v = source ;                          // Set current active vertex to the source
dv = 0 ;                                            // Initial distance is zero
// Main loop starts here
while $v! =$ target do
    foreach neighbour u of v ;           // loop over neighbours of active vertex
     do
        if u $\in$ unvisitedQ ;                    // only check those not yet visited
         then
            du = dist [u] ;                  // current estimate of distance of u
            if dv + 1 < du ;          // check if path to u through v is shorter
             then
                dist [u] = dv + 1 ;    // if yes, update estimate of distance of u
            end
        end
    end
    remove v from unvisitedQ ;                    // v will never be visited again
    v = fetch min distance vertex from unvisitedQ ;   // set new active vertex to be
     an
    ;                                           // unvisited vertex with minimum
    ;                                                  // distance from the source
    dv = dist [v]
end
```

A Python implementation of the algorithm is available in the class www site. Some comments on the above pseudocode:

- In this implementation of the algorithm, all edges have length 1. It is easy to generalise to the case where the edges have other weights by replacing line 18 with
  dist [u] = dv + **length** (u,v)
  where length(v_1,v_2) is the function which gives the weight of the edge linking vertices v_1 and v_2.
- Line 23 assumes that a function has been provided which searches the list of unvisiting vertices and associated list of distances and returns an unvisited vertex whose distance to the source is minimal. In the sample code, this is done using simple linear search. For this implementation the typical runtime of the algorithm (at least for sparse graphs) is $O(|V|^2)$ where $|V|$ is the number of vertices. This is because roughly we do one loop over the $|V|$ vertices, each of which requires a linear search of an array of length $|V|$. If the number of vertices is large, linear search maybe start to slow the algorithm down significantly. This can be remedied by using some of the more sophisticated searching and sorting procedures which we discussed earlier. A specialised data structure called a priority queue [2] supports insertion, deletion and retrieval of minimal key-value pairs (in this case the key would be the distance and the value would be the vertex) in $O(\log n)$ time where n is the number of entries in the queue.
- As written, the algorithm only calculates the length of the shortest path from source to target. In dynamic programming terms, it does the "sweep" stage of the DP solution. If we wish to obtain an actual path then we need to do the backtracking stage too. This means starting at the

target, stepping to a neighbouring vertex whose distance to the source is minimal and iterating this procedure until the source is reached. Here is some pseudocode which achieves this:

```
v = target ;                          // Set current active vertex to the target
dv = dist [v] ;                       // Initial distance is shortest path length
spath = [v]
while $v! =$ source do
    foreach neighbour $u$ of $v$ ;          // loop over neighbours of active vertex
     do
        if $u \notin$ unvisitedQ ;      // only check those visited during sweep stage
         then
            du = dist [u] ;                          // distance of u from source
            if $du == dv -1$ ;          // check if u is a step closer to the source
             then
                v=u ;                               // if yes, update active vertex
            end
        end
    end
    dv = dist [v];                         // Update distance of new active vertex
    add v to spath ;                          // Add new active vertex to the path
end
return spath
```

As we have remarked before, there may be more than one shortest path. This procedure will only find one of them but can be adapted to find all of them.

- If instead of terminating when the target vertex is found, line 10 were changed to terminate when the list of unvisited vertices is empty, then the sweep stage will have found the length of the shortest path from the source to any other vertex on the graph. Therefore in terms of computational complexity, finding all shortest paths is as easy (or as hard) as finding one shortest path since both are $O(|V|^2)$ operations!

Bibliography

[1] Associative array, 2014. http://en.wikipedia.org/wiki/Associative_array.

[2] Priority queue, 2014. http://en.wikipedia.org/wiki/Priority_queue.