Part B: Greedy Algorithms and Dynamic Programming

Wayne Goddard, School of Computing, Clemson University, 2019

Chapter B1: Greedy Algorithms and Spanning Trees

In a greedy algorithm, the optimal solution is built up one piece at a time. At each stage the best feasible candidate is chosen as the next piece of the solution. There is no back-tracking.

These notes are based on the discussion in Brassard and Bratley.

B1.1 The Generic Greedy Algorithm

The elements of a greedy algorithm are:

1. A set C of candidates
2. A set S of selected items
3. A ***solution check***: does the set S provide a solution to the problem (ignoring questions of optimality)?
4. A ***feasibility check***: can the set S be extended to a solution to the problem?
5. A ***select*** function which evaluates the items in C
6. An ***objective function***

Example. How do you make change in South Africa with the minimum number of coins? (The coins are 1c, 2c, 5c, 10c, 20c, 50c.) Answer: Repeatedly add the largest coin that doesn't go over.

The set C of candidates is the infinite collection of coins $\{1, 2, 5, 10, 20, 50\}$. The feasibility check is that the next coin must not bring the total to more than that which is required. The select function is the value of the coin. The solution check is whether the selected coins reach the desired target.

However, a greedy algorithm does not work for every monetary system. Give an example!

In general, one can describe the greedy algorithm as follows:

```
Greedy(C:set)
    S := [ ]
    while not Solution(S) and C nonempty do {
        x := element of C that maximizes Select(x)
        C := C − [x]
        if Feasible(S +[x]) then S += [x]
        }
    if Solution(S) then return S
                   else return "no solution"
```

There are greedy algorithms for many problems. Unfortunately most of those do not work! It is not simply a matter of devising the algorithm—one must prove that the algorithm does in fact work.

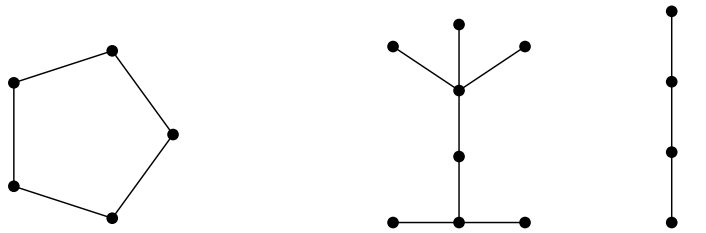
One useful concept for proving the correctness of greedy algorithms is the definition of a ***promising*** set. This is a set that can be extended to an optimal solution. It follows that:

LEMMA. *If S is promising at every step of the Greedy procedure and the procedure returns a solution, then the solution is optimal.*

B1.2 Graphs and Minimum Spanning Trees

A ***graph*** is a collection of nodes some pairs of which are joined by edges. In a ***weighted graph*** each edge is labeled with a weight. In an unweighted graph each edge is assumed to have unit weight. The number of nodes is n and the number of edges is a.

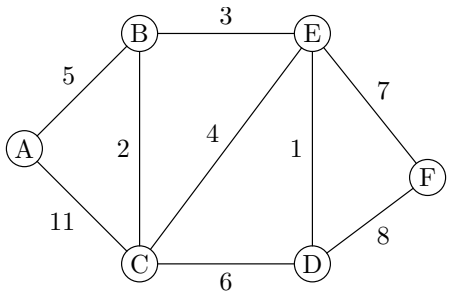
A ***walk*** in a graph is a sequence of edges where the end of one edge is the start of the next. A ***cycle*** in a graph is a walk of distinct edges that takes you back to where you started without any repeated intermediate node. A graph without a cycle is called a ***forest***. A graph is ***connected*** if there is a walk between every pair of nodes. A ***tree*** is a connected forest.



cycle *tree* *path*

A ***subgraph*** of a given graph is a graph which contains some of the edges and some

of the nodes of the given graph. A subgraph is a ***spanning*** subgraph if it contains all the nodes of the original graph. A ***spanning tree*** is a spanning subgraph that is a tree. The ***weight*** of a subgraph is the sum of the weights of the edges. The ***minimum spanning tree*** is the spanning tree with the minimum weight. For the following picture, a spanning tree would have 5 edges; for example, the edges BC, AB, BE, EF and DF. But this is not optimal.



Trees have some useful properties:

LEMMA. *(a) If a tree has n nodes then it has $n-1$ edges.*
(b) If an edge is added to a tree, a unique cycle is created.

The proof is left as an exercise.

B1.3 Prim's Minimum Spanning Tree Algorithm

Prim provided a greedy algorithm to find a minimum spanning tree (though this had been previously published by Jarník). One starts at an arbitrary node and maintains a tree throughout. At each step, we add one node to the tree.

Prim
– Candidates = edges
– Feasibility Test = no cycles
– Select Function = weight of edge if incident with current tree, else ∞.
(Note we minimize Select(x))

Example. For the graph in the previous picture, suppose we started at node A. Then we would add edges to the tree in the order: AB, BC, BE, DE, EF.

We need to discuss (1) validity (2) running time.

◊ ***Validity***

A collection of edges is ***promising*** if it can be completed to a minimum spanning tree. An edge is said to ***extend*** a set B of nodes if precisely one end of the edge is in B.

LEMMA. *Let G be a weighted graph and let B be a subset of the nodes. Let P be a promising set of edges such that no edge in P extends B. Let e be any edge of largest weight that extends B. Then $P \cup \{e\}$ is promising.*

PROOF. Let U be a minimum spanning tree that contains P. (U exists since P is promising.) If U contains e then we are done. So suppose that U does not contain e. Then adding e to U creates a cycle. There must be at least one other edge of this cycle that extends B. Call this edge f. Note that e has weight at most that of f (look at the definition of e), and that f is not in P (look at the definition of P).

Now let U' be the graph obtained from U by deleting f and adding e. The subgraph U' is a spanning tree. (Why?) And its weight is at most that of U. Since U was minimum, this means that U' is a minimum spanning tree. Since U' contains P and e, it follows that $P \cup \{e\}$ is promising. ◊

If we let B be the set of nodes inside our tree, then it is a direct consequence of the lemma that at each stage of the algorithm, the tree constructed so far is promising. When the tree reaches full size, it must therefore be optimal.

◊ ***Running time***

By a bit of thought, this algorithm can be implemented in time $O(n^2)$. One stores

- *for each node outside B, the smallest weight edge from B to it*

This is stored in an array called minDist. When a node is added to B, one updates this array.

The pseudocode given below uses that idea and assumes one starts at node 1. However, the running time can be improved using a priority queue such as a (Fibonacci) heap.

```
Prim(G:graph)
        Tree ← [ ]
        B ← [1]
        for i=2 to n do {
            nearest[i] ← 1
            minDist[i] ← weight[1,i]
            }
    while B not full do {
        min ← ∞
        for all j not in B do
            if minDist[j] < min then {
                min ← minDist[j]
                newBie ← j
                }
        Tree ← Tree + Edge(newBie,nearest[newBie])
        B += newBie
        for all j not in B do
            if weight[newBie,j] < minDist[j] then {
                minDist[j] ← weight[newBie,j]
                nearest[j] ← newBie
                }
    }
    return Tree
```

B1.4 Kruskal's Minimum Spanning Tree

Kruskal also found a greedy algorithm. This time a forest is maintained throughout.

Kruskal

– Candidates = edges
– Feasibility Test = no cycles
– Select Function = weight of edge
(Note we minimize Select(x))

The validity is left as an exercise. Use the above lemma.

Example. For the graph on page 3, the algorithm would proceed as follows. DE, BC, BE, CE not taken, AB, CD not taken, EF.

◊ *Running time*

One obvious idea is to pre-sort the edges.

Each time we consider an edge, we have to check whether the edge is feasible or not. That is, would adding it create a cycle. It seems reasonable to keep track of the different components of the forest. Then, each time we consider an edge, we check whether the two ends are in the same component or not. If they are we discard the edge. If the two ends are in separate components, then we merge the two components.

We need a data structure. Simplest idea: number each node and use an array Comp where, for each node, the entry in Comp gives the smallest number of a node in the same component. Testing whether the two ends of an edge are in different components involves comparing two array entries: $O(1)$ time. Total time spent querying: $O(a)$. This is insignificant compared to the $O(a \log a)$ needed for sorting the edges.

However, merging two components takes $O(n)$ time (in the worst case). Fortunately, we only have to do a merge $n - 1$ times. So total time is $O(n^2)$.

The running time of Kruskal's algorithm, as presented, is

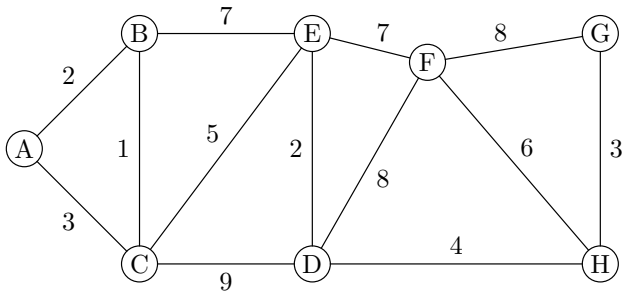
$$\max\{O(n^2), O(a \log a)\}$$

which is no improvement over the simple implementation of Prim. But actually, we can use a better data structure and bring it down. See the section on "disjoint set" data structure later.

Exercises

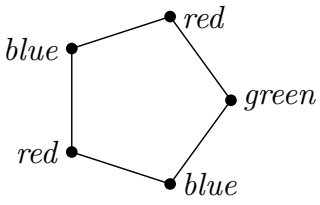
1. Prove that the greedy algorithm works for U.S. coinage. Concoct an example monetary system where it doesn't work.

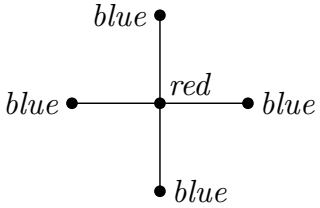
2. In what order are the edges of a minimum spanning tree chosen in the following graph, using (a) Prim (b) Kruskal?



3. Write a couple of pages on spanning tree algorithms, explaining why they work and how to implement them. Include at least one algorithm not described here.

4. In a ***coloring*** of a graph, one assigns colors to the nodes such that any two nodes connected by an edge receive different colors. An optimal coloring is one which uses the fewest colors. For example, here is an optimal coloring of these two graphs.





Dr I.B. Greedy proposes the following algorithm for finding an optimal coloring in a graph where the nodes are numbered 1 up to n: Use as colors the positive integers and color the nodes in increasing order, each time choosing the smallest unused color of the neighbors colored so far.

(a) How long does this algorithm take? Justify.

(b) Does the algorithm work? Justify.

Chapter B2: Dynamic Programming

Often there is no way to divide a problem into a ***small*** number of subproblems whose solution can be combined to solve the original problem. In such cases we may attempt to divide the problem into many subproblems, then divide each subproblem into smaller subproblems and so on. If this is all we do, we will most likely end up with an exponential-time algorithm.

Often, however, there is actually a limited number of possible subproblems, and we end up solving a particular subproblem several times. If instead we keep track of the solution to each subproblem that is solved, and simply look up the answer when needed, we would obtain a much faster algorithm.

In practice it is often simplest to create a ***table*** of the solutions to all possible subproblems that may arise. We fill the table not paying too much attention to whether or not a particular subproblem is actually needed in the overall solution. Rather, we fill the table in a particular order.

B2.1 Longest Increasing Subsequence

Consider the problem of: Given a sequence of n values, find the ***longest increasing subsequence***. By subsequence we mean that the values must occur in the order of the sequence, but they ***need not be consecutive***.

For example, consider 3,4,1,8,6,5. In this case, 4,8,5 is a subsequence, but 1,3,4 is not. The subsequence 4,8 is increasing, the subsequence 3,4,1 is not. Out of all the subsequences, we want to find the longest one that is increasing. In this case this is 3,4,5 or 3,4,8 or 3,4,6.

Now we want an efficient algorithm for this. One idea is to try all subsequences. But there are too many. Divide-and-conquer does not appear to work either.

Instead we need an idea. And the idea is this: if we know the longest increasing subsequence that ends at 3, and the longest one that ends at 4, and ... the longest one that ends at 6, then we can determine the longest that ends at 5.

How? Well, any increasing subsequence that ends at 5 has a penultimate element that is smaller than 5. And to get the longest increasing subsequence that ends at 5 with, for example, penultimate 4, one takes the longest increasing subsequence that ends with 4 and appends 5.

Books on dynamic programming talk about a ***principle of optimal substructure***: paraphrased, this is that

- *"a portion of the optimal solution is itself an optimal solution to a portion of the problem".*

For example, in our case we are interested in the longest increasing sequence; call it $\mathcal{L}$. If the last element of $\mathcal{L}$ is $A[m]$, then the rest of $\mathcal{L}$ is the longest increasing subsequence of the sequence consisting only of those elements lying before $A[m]$ and smaller than it.

Suppose the input is array A. Let $f(m)$ denote the longest increasing subsequence that ends at $A[m]$. In our example:

A	3	4	1	8	6	5
f	1	2	1	3	3	3

In general, to compute $f(m)$: go through all the $i < m$, look at each i such that $A[i] < A[m]$, determine the maximum $f(i)$, and then add 1. In other words:

$$f(m) = 1 + \max_{i<m}\{\, f(i) : A[i] < A[m] \,\}$$

Efficiency. The calculation of a particular $f(m)$ takes $O(m)$ steps. The algorithm calculates $f(1)$, then $f(2)$, then $f(3)$ etc. Thus, the total work is quadratic—it's on the order of $1 + 2 + \cdots + n$.

Dynamic programming has a similar flavor to divide-and-conquer. But dynamic programming is a bottom-up approach: smaller problems are solved and then combined to solve the original problem. The efficiency comes from storing the intermediate results so that they do not have to be recomputed each time.

B2.2 Largest Common Subsequence

The largest increasing subsequence problem discussed above is a special case of the ***largest common subsequence problem***. In this problem, one is given two strings or arrays and must find the longest subsequence that appears in both of them.

(Explain why the longest increasing subsequence problem is a special case of the longest common subsequence problem.)

The approach is similar to above. One does organized iteration. Suppose the input is two arrays A and B. Then define

$$g(m, n) \textit{ to be the longest common subsequence that ends with } A[m] \textit{ and } B[n].$$

Obviously this is 0 unless $A[m] = B[n]$.

To compute $g(m, n)$ from previous information, we again look at the penultimate value in the optimal subsequence. Say the penultimate is in position i in A and in position j

in B (with of course $A[i] = B[j]$). Then the portion up to the penultimate is the longest common subsequence that ends with $A[i]$ and $B[j]$.

So we obtain the recursive formula:

$$g(m,n) = \begin{cases} 1 + \max\limits_{i<m,\, j<n} g(i,j) & \text{if } A[m] = B[n], \\ 0 & \text{otherwise} \end{cases}$$

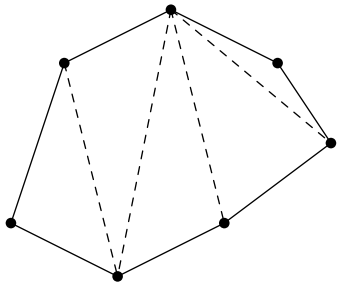
A simple implementation yields an $O(n^3)$ algorithm. Note that rather than doing recursion, one works one's way systematically through the table of $g(m,n)$ starting with $g(1,1)$, then $g(2,1)$, then $g(3,1)$, and so on.

B2.3 The Triangulation Problem

The following is based on the presentation of Cormen, Leiserson, and Rivest.

A ***polygon*** is a closed figure drawn in the plane which consists of a series of line segments, where two consecutive line segments join at a vertex. It is called a ***convex polygon*** if no line segment joining a pair of nonconsecutive vertices intersects the polygon.

To form a ***triangulation*** of the polygon, one adds line segments inside the polygon such that each interior region is a triangle. If the polygon has n vertices, $n-3$ line segments will be added and there will be $n-2$ triangles. (Why?)



$n = 7$

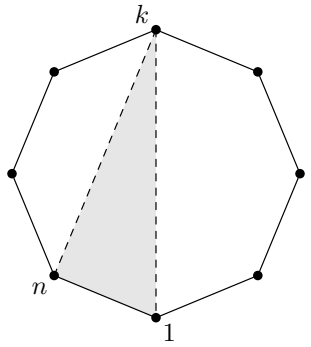
Now, suppose that associated with any possible triangle there is a ***weight*** function. For example, one might care about the perimeter of the triangle. Then the ***weight*** of a triangulation is the sum of the weights of the triangles. An ***optimal triangulation*** is one of minimum weight.

(This problem comes up in graphics. In particular it is useful to find the optimal triangulation where the weight function is the perimeter of the triangle.)

There is a recursive nature to the optimal triangulation. But one still has to be careful to ensure that the number of subproblems does not become exponential.

Assume that the vertices of the polygon are labeled v_1, v_2, ..., v_n where v_n is adjacent to v_1. Now, the side v_1v_n must be the side of some triangle: say v_k is the

other vertex of the triangle in the optimal triangulation. This triangle splits the polygon into two smaller polygons: one with vertices $\{v_1, v_2, \ldots, v_k\}$ and one with vertices $\{v_k, v_{k+1}, \ldots, v_n\}$. Furthermore, the optimal triangulation of the original problem includes optimal triangulations of these two smaller polygons.



Now, if we apply the recursion to the smaller polygons using the non-original segment as the base of the triangle, we get two smaller problems, and the boundaries of these polygons again only have one non-original segment.

Let us define $t[i, j]$ for $1 \leq i < j \leq n$ as the weight of an optimal triangulation of the polygon with vertices $\{v_i, v_{i+1}, \ldots, v_j\}$. If $i = j - 1$ then the polygon is degenerate (has only two vertices) and the optimal weight is define to be 0.

When $i < j - 1$, we have a polygon with at least three vertices. We need to minimize over all vertices v_k, for $k \in \{i+1, i+2, \ldots, j-1\}$, the weight of the triangle $v_i v_k v_j$ added to the weights of the optimal triangulations of the two smaller polygons with vertices $\{v_i, v_{i+1}, \ldots, v_k\}$ and $\{v_k, v_{k+1}, \ldots, v_j\}$.

Thus we obtain the formula:

$$t[i,j] = \begin{cases} 0 & \text{if } i = j-1 \\ \min\limits_{i<k<j} t[i,k] + t[k,j] + w(\triangle v_i v_k v_j) & \text{if } i < j-1 \end{cases}$$

The calculation of the time needed is left to the reader.

B2.4 Matrix Chain Multiplication

Recall that the product AB of two matrices A and B is valid exactly when the number of columns of A equals the number of rows of B. The result has the number of rows of A and the number of columns of B.

To multiple a sequence of matrices, such as $ABCD$, the number of columns of one matrix must equal the number of rows of the next matrix. If that condition is valid, then one can parenthesize the sequence into two-matrix products. For example, to

compute $ABCD$, we could proceed $((AB)C)D$ or $(AB)(CD)$ for example. It is known that matrix multiplication is associative: the choice of parentheses does not affect the answer. (It is not commutative though, so reordering is not allowed.)

However, some of these choices are more efficient than others. We consider here the problem of:

what choice of parentheses gives the fewest scalar multiplications?

We assume one uses the naive method for matrix multiplication (not, for example, Strassen). Recall that naive multiplication of an $m \times n$ matrix by an $n \times \ell$ matrix takes exactly $mn\ell$ scalar multiplications.

So the input is just the dimensions: say $p_0, p_1, \ldots, p_n$ where matrix A_i has dimensions $p_{i-1} \times p_i$. The goal is the optimal parenthesization to compute the product $A_1 A_2 \ldots A_n$.

It turns out that there are (very crudely) about 4^n possible parenthesizations. But this problem can be solved efficiently by dynamic programming. The key is that the sub-parenthesizations must themselves be optimal. (Huh?)

Let me explain. At the very last multiplication, we will multiply two matrices. This will be $(A_1 \ldots A_k)(A_{k+1} \ldots A_n)$ for some k. We don't know yet what k is. But one can observe that $A_1 \ldots A_k$ should be calculated as cheaply as possible, as should $A_{k+1} \ldots A_n$, and that how they are calculated does not affect how many scalar multiplications are needed when they are finally multiplied. Now think recursion. We don't know k: so try all possible k. To determine the optimal parenthesization for $A_1 \ldots A_k$, again the final step is a product of two matrices, say $(A_1 \ldots A_j)(A_{j+1} \ldots A_k)$. Looking some more, one can see that every problem in the recursion has the same structure...

So here is the algorithm: define $m[i, j]$ to be the minimum number of scalar multiplications to form the product $A_i \ldots A_j$. The product $A_i \ldots A_j$ will be computed by computing $A_i \ldots A_k$ and $A_{k+1} \ldots A_j$ and then multiplying the two results together. This means that the recurrence is (wait for it)

$$m[i, j] = \min \{ m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \}$$

where the minimum is over all k between i and j. We need to be a bit more precise on the range of k. If we define $m[i, i]$ to be 0, then we are okay if we say that k runs from i up to $j - 1$ (inclusive).

The code ends up filling a table of $m[i, j]$ with $i \leq j$. In particular, we fill the table diagonal by diagonal. First we do $m[i, j]$ where $j - i = 1$, then where $j - i = 2$, and so on. The final value is in $m[1, n]$.

Here is an example: suppose the p_i are $6, 5, 3, 1, 2, 4, 2$. Then the resulting table is as

follows:

$$\begin{pmatrix} 0 & 90 & 45 & 57 & 77 & 73 \\ & 0 & 15 & 25 & 43 & 41 \\ & & 0 & 6 & 20 & 22 \\ & & & 0 & 8 & 16 \\ & & & & 0 & 16 \\ & & & & & 0 \end{pmatrix}$$

For example, $m[2,5] = 43$ is calculated as the minimum of
$m[2,2] + m[3,5] + 5 \times 3 \times 4 = 0 + 20 + 60 = 80$
$m[2,3] + m[4,5] + 5 \times 1 \times 4 = 15 + 8 + 20 = 43$
$m[2,4] + m[5,5] + 5 \times 2 \times 4 = 25 + 0 + 40 = 65.$

The optimal parenthesization takes $m[1,6] = 73$ scalar multiplications. It can be determined to be $(A_1(A_2A_3))((A_4A_5)A_6)$ (and is unique).

Exercises

1. Illustrate the behavior of:

   a) The longest increasing subsequence algorithm on the list:
   2, 11, 3, 10, 8, 6, 7, 9, 1, 4, 5

   b) The longest common subsequence algorithm on the two lists:
   1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 2, 11, 3, 10, 8, 6, 7, 9, 1, 4, 5

2. Code up the longest common subsequence algorithm. Your algorithm should return one of the longest common subsequences, not just the length.

3. John Doe wants an algorithm to find a triangulation where the sum of the lengths of the additional line segments is as small as possible. Can you help?

   Jane Doe wants an algorithm to find an optimal triangulation where the weight function is the area of the triangle. Can you help?

4. *(From Cormen et al.)* Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j and we leave exactly one space between words, the number of extra characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$. The ***penalty*** for that line is the cube of the number of extra spaces. We wish to minimize the sum, over all lines except the last, of the penalties. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and storage requirements of your algorithm.

Chapter B3: Paths, Graphs, and Search

B3.1 Breadth-first Search

A ***search*** is a systematic way of searching through the nodes for a specific node. The two standard searches are breadth-first search and depth-first search, which both run in time proportional to the number of edges of the graph.

The idea behind ***breadth-first search*** is to:

> *Visit the source; then all its neighbors; then all their neighbors; and so on.*

If the graph is a tree and one starts at the root, then one visits the root, then the root's children, then the nodes at depth 2, and so on. That is, one level at a time. This is sometimes called ***level ordering***. BFS uses a ***queue***: each time a node is visited, one adds its (not yet visited) out-neighbors to the queue of nodes to be visited. The next node to be visited is extracted from the front of the queue.

```
BFS (start):
 enqueue start
 while queue not empty do {
      v = dequeue
      for all out-neighbors w of v do
           if ( w not visited ) {
                visit w
                enqueue w
           }
}
```

B3.2 Depth-First Search

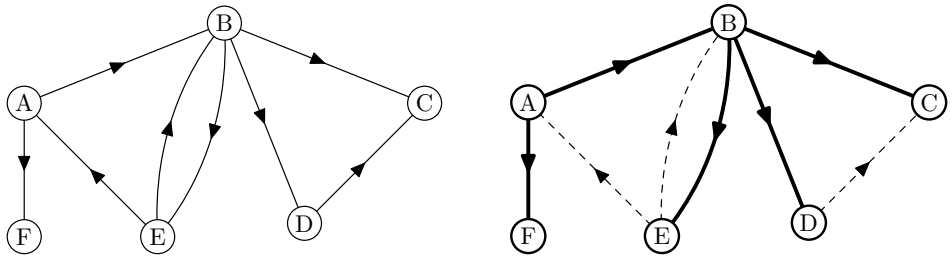
The idea for ***depth-first search*** (DFS) is "labyrinth wandering":

> *keep exploring new nodes from current node; when get stuck, backtrack to most recent node with unexplored neighbors*

In DFS, the search continues going deeper into the graph whenever possible. When the search reaches a dead end, it backtracks to the last (visited) node that has un-visited

neighbors, and continues searching from there. A DFS uses a ***stack***: each time a node is visited, its unvisited neighbors are pushed onto the stack for later use, while one of its children is explored next. When one reaches a dead end, one pops off the stack. The edges/arcs used to discover new nodes form a tree.

EXAMPLE. Here is graph and a DFS-tree from node A:




```
DFS(v):
 for all edges e outgoing from v do {
      w = other end of e
      if w unvisited then {
           label e as tree-edge
           recursively call DFS(w)
      }
 }
```

B3.3 Test for Strong Connectivity

Note that both BFS and DFS visit all nodes that are reachable. Thus they provide a linear-time test for connectivity in undirected graphs.

A directed graph is ***strongly connected*** if one can get from every node to every other node. Here is an algorithm to test whether a directed graph is strongly connected or not:

Strong Connectivity

1. Do a DFS from arbitrary node v and check that all nodes are reached
2. Reverse all arcs and repeat

Why does this work? Think of node v as the hub...

B3.4 Dijkstra's Algorithm

The following question comes up often. What is the quickest way to get from A to B? This is known as the shortest-path problem. The underlying structure is a graph. The graph need not be explicitly precalculated. It could be the state graph of a finite automaton, the search graph of an AI problem, or the position graph of a game.

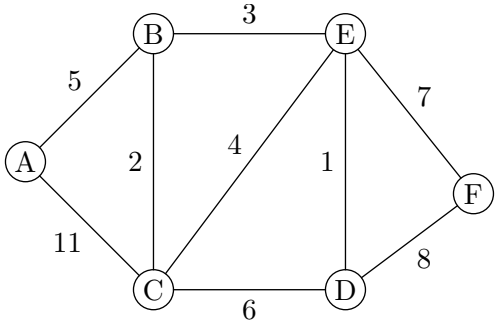
If we are just interested in finding the shortest path from one node to another node in a graph, then the famous algorithm is due to Dijkstra. It essentially finds a breadth-first search tree.

We grow the tree one node at a time. We define the auxiliary function $currDis(v)$ for a node v as the length of the shortest path to v subject to the restriction that the penultimate node is in the current tree. At each stage we add to the current tree that node which has the smallest value of $currDis(v)$. We then update the value of $currDis(v)$ for the remaining nodes.

The following produces the distance from node a to all other nodes.

```
ShortestPath (G:graph, a:node)
 for all nodes v do currDis(v) ← infinity
 currDis(a) ← 0
 remainder ← [ all nodes ]
 while remainder nonempty do {
      let w be node with minimum value of currDis
      remainder −= [w]
      for all nodes v in remainder do
           currDis (v) ← min ( currDis(v), currDis(w)+length(w,v) )
 }
```

Example. For the following graph, the table gives the value of currDis at each stage.



A	B	C	D	E	F
⓪	∞	∞	∞	∞	∞
	⑤	11	∞	∞	∞
		⑦	∞	8	∞
			13	⑧	∞
			⑨		15
					⑮

Complexity: quadratic. We go through the while loop about n times and the update loop takes $O(n)$ work. To potentially speed up, use a priority queue that supports decreaseKey. See section on Fibonacci heap later.

B3.5 The All Pairs Shortest Path Problem

Suppose we wanted instead to calculate the shortest path between every pair of nodes. One idea would be to run Dijkstra with every node as source node.

Another algorithm is the following dynamic programming algorithm known as Floyd–Warshall. Suppose the nodes are ordered 1 up to n. Then we define

> $d_m(u, v)$ *as the length of the shortest path between* u *and* v *that uses only the nodes numbered* 1 *up to* m *as intermediates.*

The answer we want is $d_n(u, v)$ for all u and v. (Why?)

There is a formula for d_m in terms of d_{m-1}. Consider the shortest u to v path that uses only nodes labeled up to m—call it P. There are two possibilities. Either the path P uses node m or it doesn't. If it doesn't, then P is the shortest u to v path that uses only nodes up to $m-1$. If it does use m, then the segment of P from u to m is the shortest path from u to m using only nodes up to $m-1$, and the segment of P from m to v is the shortest path from m to v using only nodes up to $m-1$.

Hence we obtain:

$$d_m(u, v) = \min \begin{cases} d_{m-1}(u, v) \\ d_{m-1}(u, m) + d_{m-1}(m, v) \end{cases}$$

The resultant program iterates m from $m = 0$ to $m = n - 1$. Each time there are $O(n^2)$ values of $d_m(u, v)$ to be calculated, and each calculation takes $O(1)$ time. Hence, we have an $O(n^3)$ algorithm. (Same as Dijkstra but runs faster.) Note the storage requirements.

Exercises

1. Implement Floyd–Warshall using your favorite programming language.

   The program should take the input graph in the form of a text file provided by the user. The first line is the number of nodes. Each remaining line is an edge: three integers in order provide the number of each node and then the weight. The end of the input is signified by the triple `0 0 0`.

2. Suggest ways in which the efficiency of Floyd–Warshall might be improved.

3. Illustrate the steps of Dijkstra, using node A as source, on the graph in Exercise 2 of Chapter B1.

4. Sometimes graphs have edges with negative weights. Does the concept of distance still make sense? Do the above algorithms still work? Where might one find such graphs? Discuss carefully.