Introduction to Algorithms

Part 2: Greedy Algorithms Dynamic Programming Graph Algorithms

Chapter 1: 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.

1.1 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.

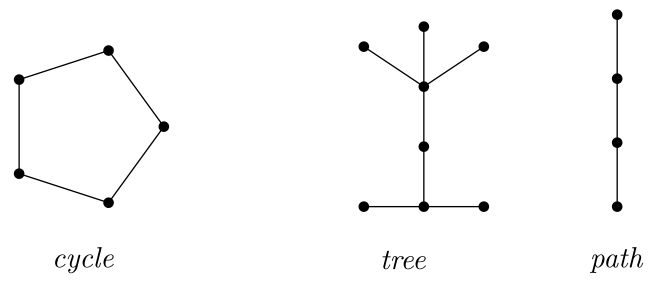
One useful concept for proving the correctness of greedy algorithms is the definition of a ***promising*** set. This is a set that can be extended to an optimal solution. It follows that:

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

1.2 Graphs and Minimum Spanning Trees

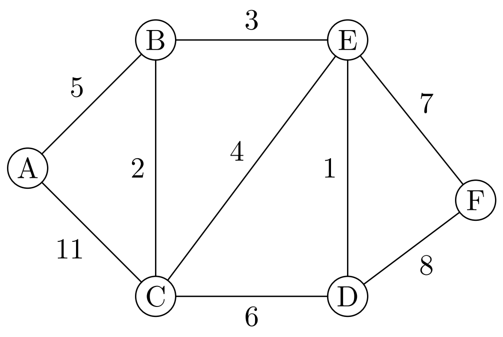
A ***graph*** is a collection of nodes some pairs of which are joined by edges. In a ***weighted graph*** each edge is labeled with a weight. In an unweighted graph each edge is assumed to have unit weight. The number of nodes is n and the number of edges is a.

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.



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.

1.3 Primm's Minimum Spanning Tree

This is a greedy algorithm. One starts at an arbitrary node and maintains a tree throughout. At each step, we add one node to the tree.

Primm
– 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 therefore ***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 Primm's 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 keeps track of

- *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 code is as follows, assuming one starts at node 1:

```
Primm(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 ← $\infty$
        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
```

1.4 Kruskal's Minimum Spanning Tree

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 4, 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 Primm. But actually, we can use a better data structure and bring it down.

1.5 Disjoint Set Structure

So, in implementing Kruskal's algorithm we need to keep track of the components of our growing forest. This requires a data structure which represents an arbitrary collection of disjoint sets. It should allow two operations:
- FIND tells one what set a value is in,
- MERGE combines two sets.

Above, we had an array implementation which took time $O(1)$ for FIND but $O(n)$ for MERGE.

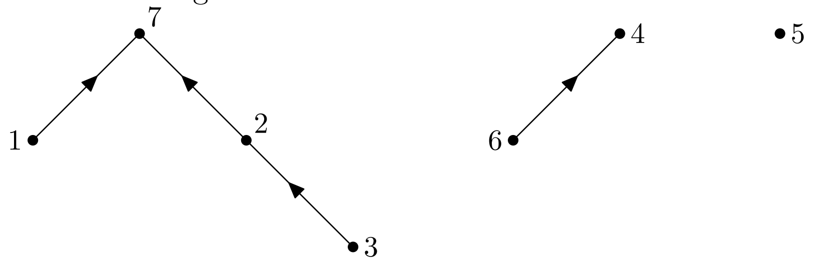
Try the following. Again store values in an array A. But this time each set is stored as a rooted sub-tree according to the following scheme:

If $A[i] = i$, then i is the label of the set and the root of some sub-tree.
If $A[i] \neq i$, then $A[i]$ is the parent of i in some sub-tree.

For example, if the components are $\{1, 2, 3, 7\}$, $\{4, 6\}$, $\{5\}$, then the array might be

	1	2	3	4	5	6	7
A	7	7	2	4	5	4	7

which represents the following sub-trees.



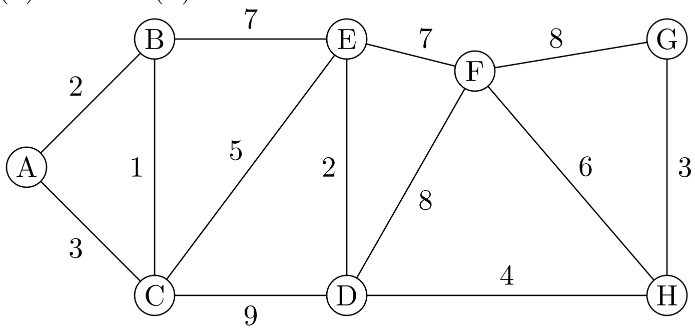
To determine the label of the set of a node, one follows the pointers up to the root. To combine two sets, one changes the root of one to point to the root of the other.

These operations take time proportional to the depth of the sub-tree. And so we haven't made any progress, yet. But there is a simple fix: When merging, make the root of the shorter one point to the root of the taller one. Recall that the ***depth*** of a rooted tree is the maximum number of edges from the root to another node. It can be shown that the depth after a series of k merges is at most $\log k$. (See exercise.) This means that both operations run in time $O(\log n)$. Note that one can keep track of the depth of a sub-tree. Further improvements are possible.

Applied to Kruskal, this gives an $O(a \log a)$ algorithm for finding a minimum spanning tree, since the sorting of the edges is now the most time-consuming part.

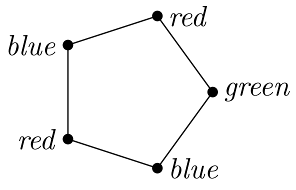
Exercises

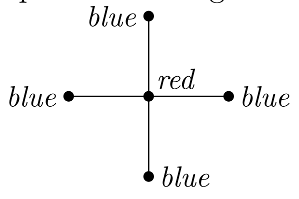
1. Prove that the greedy algorithm works for U.S. coinage. Concoct an example monetary system where it doesn't work.
2. In what order are the edges of a minimum spanning tree chosen in the following graph, using (a) Primm (b) Kruskal?



3. Write an essay 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.

5. Suppose in Kruskal's algorithm we use the rooted-tree disjoint-set data structure for keeping track of components. If the nodes are A,B,C,D,E,F,G,H and the edges that Kruskal adds are in order AB, AC, DE, EF, AG, DH, GH, what does the final data structure look like?

6. In the rooted-tree disjoint-set data structure, show that a tree of depth d has at least 2^d nodes.

Chapter 2: 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.

2.1 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.

2.2 Largest Common Subsequence

The largest increasing subsequence problem 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

$g(m, n)$ to be the longest common subsequence that ends with $A[m]$ 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:

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

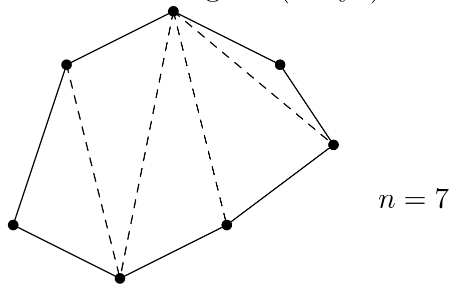
A simple implementation yields an $O(n^3)$ algorithm. Note that rather than doing recursion, one works one's way systematically through the table of $g(m,n)$ starting with $g(1,1)$, then $g(2,1)$, then $g(3,1)$, and so on.

2.3 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?)

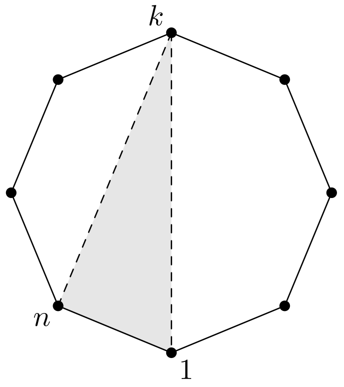


Now, suppose that associated with any possible triangle there is a ***weight*** function. For example, one might care about the perimeter of the triangle. Then the ***weight*** of a triangulation is the sum of the weights of the triangles. An ***optimal triangulation*** is one of minimum weight.

(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, $\ldots$, v_n where v_n is adjacent to v_1. Now, the side $v_1 v_n$ 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 nonoriginal segment as the base of the triangle, we get two smaller problems, and the boundaries of these polygons again only have one nonoriginal 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.

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 3: Shortest Paths

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.

3.1 Dijkstra's Algorithm

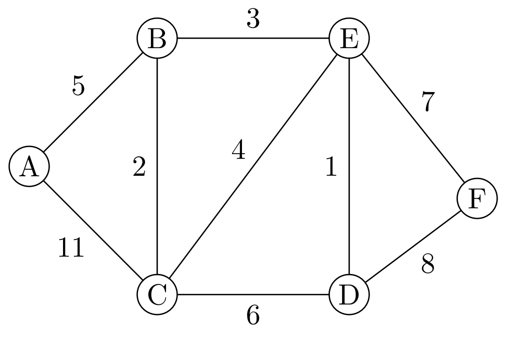
If we are just interested in finding the shortest path from one node to another node in a graph, then the famous algorithm is due to Dijkstra. It essentially finds a breadth-first search tree.

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.

3.2 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:

$$d_m(u, v) = \min \left\{ \begin{array}{l} d_{m-1}(u, v) \\ d_{m-1}(u, m) + d_{m-1}(m, v) \end{array} \right.$$

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 1.

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.

Chapter 4: Maximum Matchings and Trees

We consider the maximum matching problem. We start with the case of a tree. There are some nice algorithms on trees: both greedy algorithms and recursion can be successful.

4.1 Traversal of Rooted Trees

A ***rooted tree*** is a tree with one node designated the root. For a rooted tree, we can talk of ***parents***, ***children*** and ***descendants***. The ***subtree rooted*** at node p consists of the node p and all its descendants.

For a binary tree, we can visit the nodes of the tree in a specific order.

- In a ***preorder*** traversal, a node is visited before either of its children
- in a ***inorder*** traversal, a node is visited after visiting its left child and before its right child
- in an ***postorder*** traversal, a node is visited after both its children.

Assume the tree is stored with pointers from parent to children. Then a preorder, inorder or postorder traversal can be performed in linear time with the obvious recursive algorithm. For example:

```
Inorder(root)
    if root=nil exit
    Inorder( LeftChild(root) )
    visit(root)
    Inorder( RightChild(root) )
```

How does one justify the claim that takes linear time? (For you.) There is a natural generalization to nonbinary rooted trees of preorder and postorder traversals, and the recursive algorithm again takes linear time.

4.2 Matchings

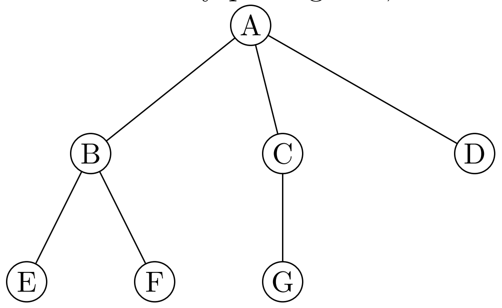
A ***matching*** in a graph is a collection of edges such that no pair of edges share a common node. A matching with the most edges is known as a ***maximum matching***.

This question comes up in scheduling problems. Suppose we have a school and we list all the classes occurring at a particular time as one set of nodes, and all the venues available at that time as another set of nodes, and join the two nodes if the venue is suitable for the class. The result is a graph. If there is a matching which uses all the classes, then the schedule for that time is possible.

An algorithm for maximum matching in trees is the following. A ***leaf-edge*** is an edge whose one end has no other neighbor. The (greedy) algorithm is to repeatedly take any leaf-edge.

```
TreeMatch(F:forest)
    M ← [ ]
    while F nonempty do {
        select any leaf-edge e
        M ← M + [e]
        F ← F − both ends of e
        }
```

A maximum matching can be obtained by pairing BE, then CG, then AD.

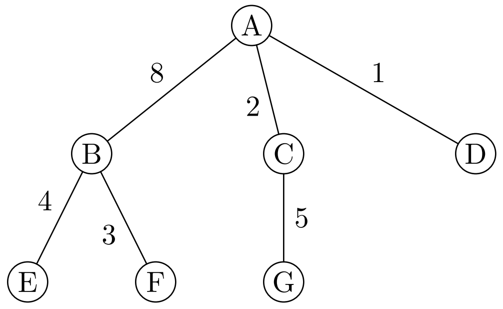


Why works? Well, assume e is a leaf edge and consider any maximum matching N. Suppose N does not contain e. Then if we add e to N, only one node now has two edges incident with it. So we can delete one of the edges of N and attain a maximum matching containing e. And so on. Hence at each stage M is promising.

One can implement this algorithm using a postorder traversal: every time one visits a node, if there is a child available then one pairs the node with one of its children.

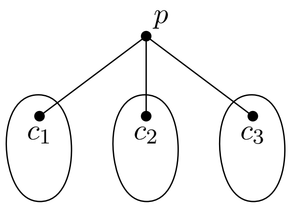
4.3 Weighted Matching in Trees

If the edges have weights, then the weight of the matching is the sum of the weights. The greedy algorithm does not work here. Consider for example the following tree: if we use the greedy algorithm we would start with the edge BE, but this is wrong.



There is a recursive algorithm, though it is a bit mysterious. We will work from the leaves up to the root. For a node p we define the subtree rooted at p, denoted T_p, as the tree consisting of p and all its descendants. At each level we calculate for the subtree T_p rooted at p two quantities:

- the maximum weight matching of T_p, denoted $m(T_p)$;
- the maximum weight matching over all matchings such that the root is not in the matching, denoted $mnr(T_p)$.



The maximum weight matching in the subtree T_p is easy to calculate if the root is specifically excluded. For, one takes the best matching in each of the subtrees rooted at each of the children and simply combines them:

$$mnr(T_p) = \sum_{\textit{children } c} m(T_c)$$

If the root is allowed to be in the matching, then apart from the above situation we also have the possibility that there is an edge in the optimal matching connecting p to one of its children. Say that child is c. Then in the subtree rooted at c we must take an optimal matching which does not include c, while in the other subtrees we can take any matching. This gives the following formula:

$$m(T_p) = \max \begin{cases} mnr(T_p) \\ \max\limits_{\textit{children } c} w(p-c) + mnr(T_c) + \sum\limits_{d \neq c} m(T_d) \end{cases}$$

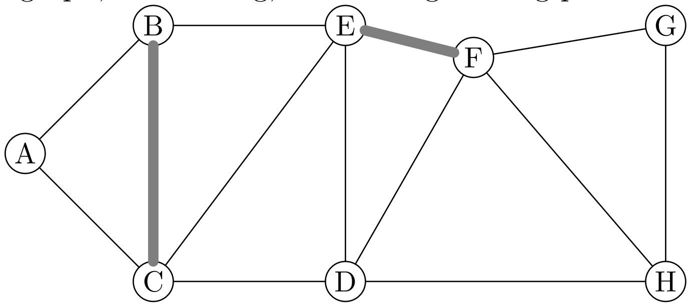
So our algorithm is: visit the nodes with a postorder traversal. At each node, the values for children are already calculated, and so we can calculate the values for the parent by the above recursive formulas. The result is a linear-time algorithm. (That this is true for non-binary trees requires some work to show.)

4.4 Maximum Matching in General Graphs

Let us return to the maximum matching problem without weights but now in a general graph.

Let M be a matching. A path is called ***alternating*** (with respect to M) if the first edge of the path is not in M and thereafter the edges alternate being in M and out of M. A path is called ***augmenting*** if it is alternating and neither end-node is incident with an edge of M.

EXAMPLE. Here's a graph, a matching, and an augmenting path is $ACBEFG$.



We also need the concept of the ***symmetric difference*** of two sets:

$$E \triangle F = (E - F) \cup (F - E).$$

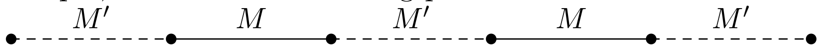
That is, the symmetric difference consists of those edges in precisely one of the two sets.

Before reading further, explain to yourself what the symmetric difference of two matchings can look like. The following result is due to Berge.

LEMMA. A matching is maximum if and only if there exists no augmenting path.

PROOF. We show: (1) if there is an augmenting path, then we can increase the matching; and (2) if the matching is not largest, then there is an augmenting path.

(1) Suppose M is a matching and there is an augmenting path P. Then let $M' = M \triangle P$. In other words, delete from M those edges lying in P and add back in those edges in P which didn't lie in M. The result is a matching. The matching M' has one more edge than M. For example, consider the following picture:



(2) Suppose M is a matching and there is a larger matching N. Consider the graph $H = M \triangle N$. Each node of H is incident with at most two edges, and if it is incident with two edges then one edge is from M and one is from M. This means that H consists of cycles and paths. Furthermore, a cycle in H must have an even number of edges: they alternate between M and N.

Now, since there are more edges in N than in M, in H there must be more edges of N than of M. In an even cycle the edges of M and N alternate so they are equinumerous.

So there must be a path P in H in which N is in the majority. But think about the path P: it must be augmenting with respect to M! ◇

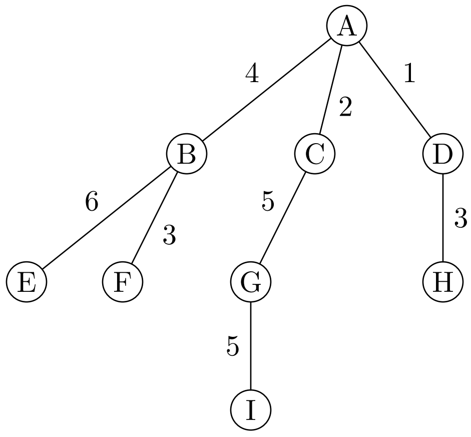
This lemma gives an iterative algorithm for finding a maximum matching in a graph.

```
MaximumMatching (G:graph)
    M ← [ ]
    repeat
        find augmenting path P
        if found then M ← M △ P
    until path not found
```

There's only the "small" problem of finding an augmenting path! We leave that to a later course.

Exercises

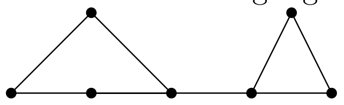
1. Illustrate the workings of the maximum matching algorithm on the following weighted tree.



2. A graph is ***bipartite*** if the nodes can be colored with two colors such that adjacent nodes receive different colors. Show that if a graph is bipartite then it cannot contain a cycle with an odd number of node. (The converse is also true.)

3. A tree is always bipartite. Construct a linear-time algorithm for finding a 2-coloring of a tree.

4. Illustrate the steps of the maximum matching algorithm on the following graph.



5. A set of nodes is said to be ***dominating*** if every node in the graph is either in the set or adjacent to a member of the set. The ***domination number*** of a graph is the minimum size of a dominating set. What is the domination number of a path on n nodes?

6. Describe a linear-time algorithm for finding the domination number of a binary tree.

Chapter 5: Revision Problems

1. A thief robbing a jeweler finds only some containers of metallic powders. Each powder has a value per gram. The thief wants to take away as valuable load as possible. Unfortunately her knapsack can hold only 20 kg. Devise a algorithm to determine the best load, quickly.

2. The Coinage system of Octavia has only coins which are a power of 8, that is, 1, 8, 64, 512 etc. Devise a simple algorithm for determining the minimum number of coins to make up a given value.

3. Given a graph, a ***happy*** coloring is one which colors the nodes with two colors red and blue such that at least half of the edges have different colored ends. Find an efficient algorithm for finding a happy coloring.

4. You are given a sorted array of n integers (which can be positive or negative) $x_1, x_2, \ldots, x_n$. Give an algorithm which finds an index i such that $x_i = i$ provided such an index exists. Your algorithm should take time $O(\log n)$ in the worst case.

5. You are given a list of n integers and must find the m smallest elements, where m is much smaller than n. Would you:
   (a) sort the list
   (b) use RankSelect m times
   (c) use some other method?

   Justify your answer.

6. When one wants to compute the product of n matrices, since matrix multiplication is associative we can put brackets wherever we like. For example, to compute $ABCD$, there are five possibilities: $(AB)(CD)$, $((AB)C)D$, $(A(BC))D$, $A((BC)D)$, $A(B(CD))$. Give an algorithm to determine how many possibilities there are for n matrices.

7. Given a collection of n line segments on the straight line $[x_i, y_i]$, devise a fast algorithm to count how many pairs of line segments intersect.

8. The obvious way to compute the maximum and minimum elements in an n-element set takes $2n - 3$ comparisons. Give an algorithm which takes fewer comparisons.

9. Design a fast algorithm that takes a sequence of numbers and an integer k and counts the number of increasing subsequences of length k.

Chapter 6: Books

I have made much use of:

• *Data Structures and Algorithms*,
A. Aho, J. Hopcroft and J. Ullman

• *Fundamentals of Algorithmics*,
G. Brassard and P. Bratley

• *Introduction to Algorithms*,
T. Cormen, C. Leiserson and R. Rivest

• *Introduction to Parellel Algorithms and Architectures: Arrays, Trees, Hypercubes*,
F.T. Leighton

• *Data Structures and Algorithm Analysis,*
M.A. Weiss