Enclosing walks and image segmentation algorithms

A. Malistov, I. Ivanov-Pogodaev, A. Kanel-Belov

A. Algorithms

Consider a book with n pages. Every page contains some word. Moreover, there is a card with a single word. Using one operation we can look an arbitrary page and read a word. How many operations are required to find the word from the card in the book? It is clear that n tests are sufficient to look up all the pages in the book.

Suppose that all the words in the book are sorted in alphabetical order. Now $\lceil \log_2 n \rceil$ operations are sufficient to check for the existence the word in the book. First of all, we can check the midpage of the book against our word and eliminate half of the book from further consideration. The *binary search* algorithm repeats this procedure, halving the size of the remaining portion of the book each time.

So, we can make a search faster using specially prepared data. Modern computer data bases contain billions records. The linear search algorithm must look up all the records. We can find any record using 30 operations with the binary search algorithm.

Consider the computer operating the memory with some registers. A register is a location with both an address and a content — a single integer number x ($|x| < L$, L is a positive integer). L depends on input data of our problems. Address is an integer more or equal to zero. The following operations are permitted: addition, subtraction, multiplication, integer division, modulo operation, comparison and conditional branch, which may or may not cause branching depending on the comparison result. The *running time* of an algorithm is the number of operations or "steps" executed. The memory capacity used by an algorithm is the maximal address of memory unit used.

Consider an n-element linear array. Are there exist two equal elements in the array? We can solve this problem using $n(n-1)/2$ comparisons — for every pair of elements. Algorithm needs some additional operations except the comparisons. For example, this is an increment for the loop variables. The number of additional operations is less than $C_1 n^2$ for some constant C_1. Thus, the number of all operations is less than $C_2 n^2$ for some constant C_2. They often describe the running time of algorithm ignoring the constant factors. They say that the running time of the algorithm is $O(n^2)$ (big O notation).

Suppose that we want to compute result using some input parameters. Usually, more complex input increases algorithm's running time. Let n be a some integer which depends on the input. For example, n is array size or number of nodes etc. This parameter is called "input length function". There are different inputs for fixed n. We say that an algorithm has a running time $O\big(f(n)\big)$ if there exist some constants C, n_0 such that for any $n > n_0$ the number of operations executed less than $Cf(n)$ for any input of length n. Thus one can find two equal numbers in array of size n using $O(n^2)$ operations. There is an $O(n \log n)$ algorithm which solves this problem[1]. The running time of an algorithm can depend on the size L of the memory unit. An algorithm is *polynomial* if there is a polynom $p(n, \log_2 L)$ such that the runing time and memory used are less than $p(n, \log_2 L)$ for any input. An algorithm is *strong polynomial* if there is a polynom $p(n)$ such that the runing time and memory used are less than $p(n)$ for any input. For exapmle, $O(n^4)$ algoritms and $O(n \log n)$ algorithms are strong polynomial.

O-notations can be easily extended to multiple variables case. For example, for graph with V vertices and E edges, the running time $O(EV^2 \log V)$ means that there exist constants C, V_0 and E_0 such that for all $V > V_0$ and $E > E_0$ the number of operations

[1] We will use baseless log inside of big O notation. The thing is that two logarithms with different bases are just scalar multiples of each other.

executed less than $CEV^2 \log V$. Usually V is a set of vertices and E is a set of graph edges. $|V|$ and $|E|$ are sizes of these sets. We will write just V and E inside of big O notation. $O(V^2E)$ and $O(|V|^2|E|)$ are the same.

▶ **A1.** Describe an $O(n \log n)$ algorithm that determines whether or not there exist two equal elements in the array of size n.

An algorithm can use additional memory to store temporary data. The size of required memory can be estimated via O notation. There are many fast algorithms that require much memory. To the other side, there are many low memory algorithms that require much time to perform. Unless otherwise stated, any algorithm has an $O(n)$ memory capacity (or $O(E + V)$ for graphs). This means that, for the algorithm, there is a constant C such that this algorithm requires memory amount less than Cn. (or $CV + CE$ for graphs).

▶ **A2.** Suppose that there are no memory restrictions. Describe an $O(n)$-algorithm that determines whether or not there exist two equal elements in the array of size n.

Let $G = (V, E, w)$ be an weighted directed graph. V is a set of its vertices, $E = \{(u, v)\}$ $(u, v \in V)$ is a set of graph edges, $w\colon E \to \mathbb{R}$ is a weight function. Graph description can hold $3|E| + 2$ memory units. First cell holds the number of graph vertices $|V|$, second cell holds the number of graph edges $|E|$. The next cells holds $|E|$ triplets for each edge. Each triplet is the index of outgoing vertex i, the index of incoming vertex j and weight $w(i, j)$.

A walk $s \rightsquigarrow t$ from s to t is a sequence $s = v_1, v_2, \ldots, v_n = t$ such that for any $i = \overline{1, n-1}$ there exists an edge $(v_i, v_{i+1}) \in E$ from v_i to v_{i+1}. *The weight* of a walk $s \rightsquigarrow t$ is sum of all edges weights. *A shortest walk* from vertex s to vertex t is any walk $s \rightsquigarrow t$ with minimal weight. The shortest distance from s to t is the weight of a shortest walk from s to t. Shortest walks are well defined if and only if the graph G contains no negative-weight cycles. Unless otherwise stated, only **positive-weight** graphs are considered.

▶ **A3.** Suppose that all weights of the graph G are equal to 1 (unweighted graph). Let s, t be two vertices of the graph G. Describe an $O(E)$ algorithm that searches for the shortest distance from vertex s to vertex t.

▶ **A4.** Suppose that all weights of the graph G are equal to 1 (unweighted graph). Describe an $O(E)$ algorithm that searches for the shortest walk from vertex s to vertex t. Output of this algorithm is a sequence s, u_1, u_2, …, t.

For any vertex s there is a *shortest-paths tree* $T_s = (V_s, E_s, w)$ such that 1) T_s is a subgraph of G, 2) s is the root of T_s, 3) V_s contains vertices reachable from s, 4) for all $v \in V_s$, the path from root s to v in T_s is the shortest path from s to v in G.

▶ **A5.** Prove that for any vertex s there is a *shortest-paths tree* T_s.

▶ **A6.** Let $G = (V, E, w)$ be an weighted, directed graph. Let s be a some vertex. Describe an $O(V^2)$ algorithm that searches for the shortest-paths tree T_s and shortest distances from vertex s to all other vertices.

Consider a tree. Each vertex except the root has a unique parent and probably some children. The root has children only. Let us divide all the vertices by the levels. Level 1 contains the root only. Level 2 contains the children of the root. Level 3 contains the children of the verices from level 2 etc. Level $(i + 1)$ contains the children of the verices from level i.

Binary tree is a tree in which each node has at most two children. We can enumerate all the vertices of the binary tree by the following way. The vertex of level 1 (the root)

has index 1. The vertices of level 2 have indices 2, 3. The vertices of level 3 have indices 4, 5, 6 and 7, and so on. The vertices of level i have indices from 2^{i-1} to $2^i - 1$. If some vertex is absent then its index is skipped.

A binary tree is called an *almost complete* binary tree if all the levels (maybe except the last one) is completely filled. Suppose that each tree node contains some integer (*key*) and an additional data of fixed size. A *binary heap* is a almost complete binary tree such that the key at every node is less than (or equal to) the key at its left child and the key at its right child. It is clear that the element with minimal key can be found using $O(1)$ operations.

▶ **A7.** Describe an $O(\log k)$ algorithm that pushes a single element into the binary heap and retains the heap property.

▶ **A8.** Describe an $O(\log k)$ algorithm that eliminates the minimal element from the binary heap and retains the heap property.

▶ **A9.** Describe an $O(\log k)$ algorithm that changes the key of the minimal element of the binary heap and retains the heap property.

▶ **A10.** Let $G = (V, E, w)$ be the weighted, directed graph. Let s be some vertex. Describe an $O(E \log V)$ algorithm that searches for the shortest-paths tree T_s and shortest distances from vertex s to all other vertices.

B. Planar graphs

Let $G = (V, E)$ be a planar graph. Every vertex v corresponds to some point on the plane with coordinates (x_v, y_v). We assume that all the edges in considered planar graphs has nonintersection segments representation on the plane. All edges intersect only at endpoints.

▶ **B1.** Prove that for any planar graph $|E| < 3|V|$.

Further, we need Jordan's Theorem. You can use this theorem without proof.

Jordan's Theorem for polygons. Given the polygon, prove that this polygon divides the plane into an "interior" region bounded by the polygon and an "exterior" region containing all of the nearby and far away exterior points.

Consider a point H that does not belong to any edge or the vertices of the planar graph G. *The winding angle* $\omega(AB)$ for given edge AB around the point H is a signed angle AHB ($-\pi < \omega(AB) < \pi$). For any walk $w = e_1 e_2 \dots e_m$ ($e_i \in E$), *the winding angle* is a sum $\omega(e_1) + \omega(e_2) + \dots + \omega(e_m)$.

▶ **B2.** Prove that for any closed walk $v_1 v_2 v_3 \dots v_m v_1$ ($v_i \in V$), the winding angle around the point H is equal to $2\pi n$ (n is some integer). n is *the winding index* of the walk around H.

Fix some vertex s in the planar graph G. Further we will only consider walks that start in s. Let us call them s-walks. s-loop is a closed s-walk.

▶ **B3.** Consider a s-loop of winding index $n > 1$ around point H. Prove that there is a s-loop of winding index $n - 1$ around H.

C. Shortest closed walks around single point

In the sequel, we shall focus on the problem of the shortest closed walk of non-zero winding number with some source vertex s under different following conditions:

1. The number of points we want to enclose by loop: single point H or two points H_1, H_2
2. Directed or undirected graphs

3. Polynomial algorithms or fast $O(V \log V)$ algorithms.
4. Finding the shortest walks with non-zero winding index or finding the shortest walks with fixed winding index.

Given a weighted digraph $G = (V, E)$ embedded in the plane P, let H be a point of the plane P, let $s \in V$ be a source vertex. We want to find a shortest closed walk $s \rightsquigarrow s$ with a given winding number $n \neq 0$, $n \in \mathbb{Z}$, around H.

▶ **C1.** Consider an undirected graph. Find a shortest closed s-walk with nonzero index using $O(V \log V)$ operations.

▶ **C2.** Consider a directed graph. Find a shortest closed s-walk with nonzero index using $O(V \log V)$ operations.

▶ **C3.** Consider an undirected graph. Given index $k < 100$, find a shortest closed s-walk using $O(V \log V)$ operations.

▶ **C4.** Consider a directed graph. Describe a polynomial algorithm that finds a shortest closed s-walk with given index $k < 100$.

▶ **C5.** Consider a directed graph. Describe an $O(V \log V)$ algorithm that finds a shortest closed s-walk with given index $k < 100$.

D. Shortest walks around two points

Consider two points H_1 and H_2. The walks around these points have two winding angles and two indices. Recall we have a source vertex s. In the following problems we want to find closed walks with specified properties.

▶ **D1.** Consider an undirected graph. Given indices $k < 100$, $n < 100$, find a shortest closed s-walk using $O(V \log V)$ operations.

▶ **D2.** Consider a directed graph. Given indices $k < 100$, $n < 100$, describe a polynomial algorithm that finds a shortest closed s-walk.

▶ **D3.** Consider a directed graph. Given indices $k < 100$, $n < 100$, describe an $O(V \log V)$ algorithm that finds a shortest closed s-walk.

E. Shortest walks without source vertex

Now we have no source vertex. So we want to find a shortest closed walk.

A *lattice graph*, *mesh graph*, or *grid graph* is the graph whose vertices correspond to the points in the plane with integer coordinates ($x = 1, \ldots, n$; $y = 1, \ldots, m$) and two vertices are connected by an edge whenever the corresponding points are at distance 1. In directed graphs case, these two vertices are connected by two edges in both directions. The weights are probably asymmetric.

▶ **E1.** Is there exist an $O(V^{\frac{3}{2}} \log V)$-algorithm that searches for a shortest closed walk around H in directed lattice graph?

▶ **E2.** Try to find an $O(V \log V)$-algorithm that searches for a shortest closed walk around H in directed lattice graph.

▶ **E3.** Try to find an $O(V^{\frac{3}{2}} \log V)$-algorithm that searches for a shortest closed walk around H in directed planar graph.