Diseño y análisis de algoritmos

Tarea 4

Profesores Armando Castañeda y Sergio Rajsbaum
Ayudantes Luis Gómez y Diego Velázquez

Jueves 19 de septiembre de 2019
Fecha de entrega: Jueves 3 de octubre de 2019

INSTRUCCIONES:

- Se puede hacer en equipos de 2 o 3 personas, pero hay que entregarla individualmente. La resuelven entre todos, pero cada quien la escribe en sus palabras. Anotar en cada tarea el nombre de todos los miembros del equipo.
- Las respuestas deben estar escritas con claridad, todos los enunciados demostrados.
- No se aceptan tareas después de la fecha límite.
- No escribas la implementación de tus algoritmos.
- Si el ejercicio dice "prueba", "demuestra" o "muestra", no debes dejar ningún hecho sin justificar; esto significa que debes decir por qué lo que escribes es verdad. Si lo que se pide es una explicación, es suficiente que enuncies los hechos que explican lo que se pide sin decir por qué son verdaderos pero tienes que manifestarlos completamente.

Ejercicio 1. 1. Da un ejemplo de una familia de digráficas de n vértices con pesos no negativos tal que admita una ejecución del algoritmo de Dijkstra en la cual todos los nodos actualizan su padre y su estimación de distancia $d(v)$ una sola vez; es decir, la primera oferta de camino que reciben, es la del camino mas corto. Demuestra que tu ejemplo cumple lo que se pide.

2. Da un ejemplo de una familia de digráficas con pesos no negativos tal que cualquier ejecución del algoritmo de Dijkstra requiera que algún vértice actualice su padre y su estimación de distancia dos veces. Demuestra que tu ejemplo cumple lo que se pide.

Ejercicio 2. 1. De acuerdo con el algoritmo de Dijkstra que revisamos en clase, para cada $n \in \mathbb{N}$ con $n > 2$, presenta una familia de gráficas de n vértices con pesos tanto positivos como negativos y sin ciclos dirigidos de longitud negativa para la cual se cumple que si G es una gráfica de la familia, el algoritmo de Dijkstra falla en encontrar el camino más corto entre s y algún otro vértice de G. Demuestra que de hecho existe tal vértice.

2. Toma una gráfica de la familia que propones y muestra que Bellam-Ford sí encuentra el camino que Dijkstra no.

Ejercicio 3. De acuerdo con el algoritmo de Dijkstra que revisamos en clase, para cada $n \in \mathbb{N}$ con $n > 2$, presenta una familia de gráficas de n vértices con pesos positivos y negativos y sin ciclos dirigidos de longitud negativa para la cual se cumple que si G es una gráfica de la familia, el algoritmo de Dijkstra no falla en encontrar el camino más corto entre dos vértices de G. Prueba que en efecto el algoritmo encuentra los caminos más cortos.

Ejercicio 4. Considera la estructura de datos de cola de prioridad, implementada con heaps binarios, y el arreglo $A = (10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, 2)$. Explica cómo construyes una cola de prioridad que tenga los elementos de A (tienes que dar la construcción del árbol binario); no puedes ordenar el arreglo de entrada antes de crear la cola. Escribe explícitamente la cola resultante.

Ejercicio 5. Demuestra en detalle y con una figura, por qué el algoritmo de Dijkstra es correcto.

Ejercicio 6. Ofrece un algoritmo que, dada una cola de prioridad implementada con heaps binarios y con n elementos, elimine el elemento i-ésimo de la cola. Analiza su complejidad y demuestra que en efecto tu algoritmo es correcto y que tiene la complejidad que presumes.

Ejercicio 7. Sea $G = (V, E)$ una gráfica. Demuestra que cualesquiera dos de las siguientes propiedades implican la tercera.

1. G es conexa.
2. G no tiene ciclos.
3. $|E| = |V| - 1$.

Ejercicio 8. Escribe un resumen de una cuartilla de las páginas 51 a la 67 del libro *Out of their Minds: the lives and discoveries of 15 great computer scientists* de Shasha y Lazere, acerca de la vida y obra de Dijkstra. [1]

Ejercicio 9. Para resolver el problema del cálculo de distancias entre todos los vértices de una gráfica dirigida $G = (V, E)$ con pesos en las aristas positivos o negativos, se pueden utilizar los siguientes algoritmos con respectivas complejidades:

Algoritmo de Ford	$O(\|V\|^2\|E\|)$
Algoritmo de Floyd	$O(\|V\|^3)$
Algoritmo de Johnson	$O(\|V\|^2 \log \|V\| + \|V\| \cdot \|E\|)$

[1] Una cuartilla significa que de ser entregado en computadora (o máquina de escribir ¡ja!) deben apegarse a estos estándares: hoja carta o A4; letra Arial 12 puntos; interlineado 1.5; separación de caracteres normal; márgenes normales (1 pulgada de cada lado); no deben separar los párrafos con renglones, usen sangría para iniciar un nuevo párrafo. Esto es para que no escriban ni mucho ni poco, además facilita la lectura. Si van a entregar a mano, deben escribir 250 palabras como máximo y mínimo 200 (en computadora van a escribir una cantidad similar de palabras). En cualquier caso, ninguna oración debe ser de más de dos renglones de largo (eviten las oraciones subordinadas a toda costa).

Determina qué algoritmo conviene usar como función de $|E|$.

Ejercicio 10. Prueba que agregar una constante a todos los pesos y que multiplicar por una constante positiva no altera el conjunto de árboles generadores mínimos de una gráfica conexa con pesos dada.

Ejercicio 11. Demuestra que si todas las aristas de una gráfica conexa G tienen pesos diferentes dos a dos, entonces G tiene un sólo árbol generador mínimo.

Ejercicio 12. Dado un árbol generador mínimo de una gráfica con pesos $G = (V, E)$, supongamos que una arista de G es eliminada sin desconectar la gráfica. Describe cómo encontrar un nuevo árbol recubridor mínimo en tiempo proporcional a $|E|$.

Ejercicio 13. Diseña un algoritmo que resuelva el siguiente problema y demuestra que tiene la complejidad que se solicita: dada una gráfica conexa G con pesos positivos, encontrar, en a lo más tiempo $O(mlog(m))$, un árbol recubridor mínimo que minimice la arista más costosa.

Las siguientes páginas fueron tomadas de Shasha, D., & Lazere, C. (1998). *Out of their minds: the lives and discoveries of 15 great computer scientists.* Springer Science & Business Media.

having only local temporal order, since any notion of global time depended on keeping clocks perfectly synchronized or sending a message to a clock server. He then offered an algorithm for approximate clock synchronization. His model and his algorithms have laid the foundations for much of the infrastructure for cyberspace.

Robert E. Tarjan has always enjoyed drawing circles and lines between the circles. In the process, he has invented algorithms to lay out road networks on a surface, find the best flows through networks, and store historical snapshots. He has also brought elegance to the analysis of algorithms. Tarjan has given the world precise new criteria for judging the quality of algorithms, drawing upon such unlikely concepts as amortization in investing and competitiveness in economics.

A lost traveler in rural Maine might stop for directions to his final destination only to be told, "You can't get there from here." The traveler might then refer to his map to solve the problem, because he intuitively knows there is a way. Computer scientists find themselves in a similar situation but without the traveler's certainty. *Stephen Cook* and *Leonid Levin* defined a family of problems that include many interesting questions in circuit design, logic, resource allocation, and scheduling. They showed that either all of these problems are hard or all are easy. The trouble is, they don't know which. Neither does anyone else. Imagine Julia Child describing a meal that she does not know how to cook—or even whether it can be cooked at all.

Edsger W. Dijkstra

APPALLING PROSE AND THE SHORTEST PATH

I asked my mother [a mathematician] whether mathematics was a difficult topic. She said to be sure to learn all the formulas and be sure you know them. The second thing to remember is if you need more than five lines to prove something, then you're on the wrong track.

—Edsger W. Dijkstra

Science, like dress design, has its fashions. The few who ignore fashions in order to grapple with the fundamental questions of their discipline take a big gamble. Those who succeed earn the right to criticize. Throughout a career dating back to the 1950s, Edsger W. Dijkstra has both gambled successfully and criticized severely. His work on the shortest path algorithm and mutual exclusion is characterized by an Old World elegance and simplicity that he would like the rest of the world to share.

Born in Rotterdam in 1930, Dijkstra is the son of two scientists—his father was a chemist; his mother, a mathematician. Early on, Dijkstra demonstrated an aptitude and a liking for science.

> My older sister had a Meccano [like an American Erector set]—long metal strips with holes. I made a lot of machines. I remember

> I constructed two special cranes. One was constructed in such a fashion that no matter what its load was, its center of gravity was right above the fulcrum. The other one was such that when the distance between the center of the crane and the load was changed, the load remained the same height.

At age 12, in 1942, Dijkstra entered the Gymnasium Erasminium, an elite high school, where he received a traditional Dutch education—classical Greek and Latin, French, German, English, biology, mathematics, physics, and chemistry. The war brought hardship to most Dutch civilians including Dijkstra and his family. Toward the end of the occupation, when food was scarce, his family sent him out of the city.

> I traveled with a friend of a friend of my father's who still had a car. We drove to the country. There was no gasoline. The car had to run on methane. . . . The radiator broke. It was freezing. I was fourteen and very weak—my heart couldn't manage more than forty beats per minute.

The young Dijkstra rejoined his family in July 1945. Political idealism was in the air. Dijkstra thought he might study law and serve his country in the United Nations, but his father dissuaded him.

> I was talked out of law—the grades I had on my final examinations for mathematics, chemistry, and physics were so glorious.

Instead, he entered the University of Leiden where he had to choose between physics and mathematics.

> I decided that if I didn't study physics at the university, I would never do it. I felt that mathematics would look after itself.

Having elected to study theoretical physics, Dijkstra observed that many problems in the field required extensive calculation, so he decided to learn to program. Thanks to their wartime code-breaking work, the British led the development of European computing in the 1940s and 50s. In 1951, Dijkstra attended summer school in programming at Cambridge University. In March 1952 he got a part-time job at the Mathematical Centre in Amsterdam, where he became progressively more involved in computer programming.

> The Mathematical Centre was housed in an old school. The machine, called the ARMAC, occupied a classroom. It had a magnetic drum for memory [a rotating magnetic cylinder having recording heads capable of reading and writing from the outside surface]—advanced for the standards of the day.

In the early 1950s, before the advent of Fortran or Lisp, programmers wrote to suit the idiosyncratic design of each computer. A typical programmer would receive a list of the instructions that the machine could perform. If the hardware designers could simplify their design at the the expense of programming complexity, they would not hesitate to do so. The hardware designers working with Dijkstra, however, did just the opposite.

> They would never include something in the machine unless I thought it was okay. I was to write down the functional specification that was the machine's reference manual. They referred to it as "The Appalling Prose"—it was as rigorous as a legal document.

Dijkstra still had not committed himself to a career in programming—a field that was virtually unknown in the Netherlands.

> I finished my studies at Leiden as quickly as possible. Physics was a very respectable intellectual discipline. I explained to [Adriaan] van Wijngaarden [his advisor and an early Dutch computer pioneer] my hesitation about being a programmer. I told him I missed the underlying intellectual discipline of physics. He agreed that until that moment there was not much of a programming discipline, but then he went on to explain that automatic computers were here to stay, that we were just at the beginning and could not I be one of the persons called to make programming a respectable discipline in the years to come?

Dijkstra's doubts reflected the widespread ignorance about programming. When Dijkstra applied for a license to marry M. C. Debets, a colleague who was also a programmer, the bureaucrats did not recognize programmer as a profession, so he reluctantly labeled himself a "theoretical physicist."

Shortest Paths

Still at the Mathematical Centre, Dijkstra was asked to demonstrate the powers of the ARMAC for the forthcoming International Mathematical Conference of 1956. He started to think about the problem of determining the shortest route between two points on a railroad map. One sunny Saturday morning Dijkstra and his wife were sitting on the terrace of a cafe sipping coffee. Suddenly he fell silent.

> I was engaged in thought. My wife knew such periods. . . . The problem was so simple that you could find the solution without pencil and paper.

Figure 1 represents diagramatically—using highways instead of rail lines—the problem Dijkstra posed to himself. The task is to find the fastest route from City S to City T—the shortest path. Try the problem before looking at the legend text or the next paragraph.

Dijkstra's basic approach is to form an ever-growing "core set" of cities between your origin, City S, and your destination City T. At any given step of the algorithm, you know the minimum time to drive to every city in the core set. Initially, the core set consists of City S and it takes no time to drive there. At each subsequent step, you find a city outside the core set, call it X, having the property that the time to drive from City S to X is shorter than the time to drive to any other city outside the core set. Since it takes at least 0 minutes time to drive any route, X must be directly linked to some city in the core set, call it Y. The time to drive to X is then just the minimum time to drive to Y from City S (which you already know since Y is in the core set) plus the time to drive from Y to X. Now, you add X to the core set and record the time you have computed. If X is City T, then you are done. Figure 1 shows how the core set grows with each step of the algorithm.

> This was the first graph problem I ever posed myself and solved. The amazing thing was that I didn't publish it. It was not amazing at the time. At the time, algorithms were hardly considered a scientific topic.

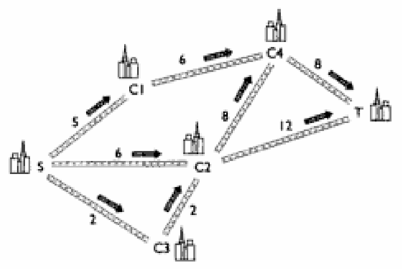

Figure 1
Dijkstra's shortest-path algorithm.
The original core set will consist of S alone.
Next, C3 will be added with a total of 2.
Next, C2 will be added with a total cost of 4 via the route S → C3 → C2.
Next, C1 will be added with a total cost of 5.
So the core set at this point consists of S, C1, C2, and C3.
Next, C4 will be added at a cost of 11 through the route S → C1 → C4.
At this point T will be added at a cost of 16 through the route S → C3 → C2 → T.

> The mathematical culture of the day was very much identified with the continuum and infinity. Could a finite discrete problem be of any interest? Obviously the number of paths from here to there on a finite graph is finite. Each path is of finite length. You must search for the minimum of a finite set. Any finite set has a minimum—next problem, please. It was not considered mathematically respectable.
>
> For many years, I had felt guilty about my lack of mathematical education. But eventually, I think I was glad I was spared the mathematical prejudices of the day.

The shortest-path algorithm, now known simply as Dijkstra's algorithm, has since been used in road building, routing through communications networks, and airline flight planning—any application in which one must find the best way to travel to a destination. Dijkstra soon twisted the algorithm slightly to solve a related practical problem.

Machine designers Loopstra and Scholten, who had been the engineers of the ARMAC, were building their next machine and sought a way to convey electricity to all essential circuits, while using as little expensive copper wire as possible. Dijkstra solved the problem with a method that he called, for technical reasons, the shortest subspanning tree algorithm.

> Then I had two nice graph algorithms. Still there was no popular journal to publish it. I published it eventually in *Numerische Mathematik* in the first issue. Of course it had nothing to do with numerical mathematics.

It was an unusual paper for the time. For one thing, it proposed an efficient way to solve a finite problem. For another, it carefully proved its results.

> Almost all mathematicians at that time were in the teaching business. There was hardly any industrial mathematics. Then it was okay to leave something for the intelligent reader. At least that was what mathematicians felt; the standard way proofs were published was to publish a sketch of a proof.
>
> It's quite clear that during those years of working on the early machines, I developed a set of quality standards, a set of values that were very different form the standard mathematical culture.
>
> An emphasis on simplicity, completeness, correctness. It started with the writing of the Appalling Prose; a reference manual should contain everything; it should be complete and unambiguous. Machines are very unforgiving. They execute the program as is. The freedom of meaning one thing and saying something different is not permitted.

The Critical Section Problem

Dijkstra's innovative work on mutual exclusion and cooperating sequential processes began in the early 1960s with the designs of the ARMAC's successors, the X1 and later the X8. From a hardware point of view, they were typical machines of the day.

> The X8 had a big core storage cycle of 10 microseconds [about 100 times slower than today's RAMs]. It had a red button and you pushed it and the machine stopped. And if you pushed the green button, it would start again.

The software, by contrast, started a trend. Dijkstra arranged for each device attached to the computer to perform its tasks one step at a time, while exchanging messages with the computer. In computer jargon, these are called communicating sequential processes. Dijkstra started to think about ways to coordinate or synchronize these processes "so that I could reason about them."

Programmers face this challenge very often. Suppose two processes access the same data at the same time. One process might modify the data and therefore cause the other process to behave incorrectly. Avoiding bad behavior requires that while one of the processes accesses the shared data, the other one does not. This is what Dijkstra meant by synchronization.

Dijkstra thought again about trains—this time a train signaling system known as a semaphore. Suppose that there are two separate tracks between cities X and Y, one from X to Y and the other from Y to X. If the two tracks narrow to one during a portion of the route, then trains going in both directions must use the same piece of track. To avoid collisions, engineers use semaphores to ensure that only one train is on the shared track at any one time. The semaphores ensure that there is a green light in only one direction at a time and that the lights don't change color while there is a train on that critical piece of track. Thus use of semaphores ensures that only one train will be on the critical track at a given time. This is called ***mutual exclusion.***

Dijkstra applied the notion of mutual exclusion to the communication between the computer and its attached keyboard. These two devices exchange information through a communication area in memory known as a buffer. The basic rule is that only one of these two should be reading or writing the buffer at a time.

> I realized that the coupling between the typewriter [the keyboard with its circuitry] and the machine was totally symmetric. Just as the machine would be forced to wait while the buffer was still full, so the typewriter would be forced to wait while the buffer was still empty. I knew we had a logical symmetry. I remember it was very liberating, very refreshing.

The cooperation of a number of units each with its own speed and clock—that was the given of the technology. What I wanted to do was arrange the cooperation in such a way that it would be independent of the relative speed ratios. I wanted to do this for safety's sake.

In 1961, Dijkstra thought of a way to represent the necessary protocols using two operations suggested by the railway semaphore: P and V. P stands for *passeren,* which in Dutch means "to pass," while V stands for *vrijgeven,* meaning "to give free." It is a testimony to the power of this idea that computer designers still use these letters, despite the dominance of English in computer science. Dijkstra's elegant solution came from his desire for clean reasoning.

The invention was not so much the P and V operations. The greater jump consisted of the decision to ignore relative speeds and to make reasoning about a system independent of that. That is not something that is taken for granted. What I remember is resistance to that idea. People found it difficult to accept that knowledge about relative speeds should be ignored.

That surely is no longer true. Virtually all modern processors and most memory boards support the functionality of P and V in hardware by means of a *test and set* instruction or something similar. These instructions either lock a computer resource, such as a buffer, and return success or determine that the resource is already locked and therefore return failure. IBM's 360 architecture was one of the first to implement test and set in 1964, thus giving the entire idea legitimacy.

Dining Philosophers

Dijkstra had been able to see things differently from his peers as a result of what he calls a "happy fact of my isolation." His ability to pose fundamental problems that others overlooked became apparent once again in 1965.

In the fall of that year, Dijkstra sat down one evening and prepared a now-famous examination problem for his students at the Einhoven Technical University. Dijkstra called it the dining quintuple

problem, but it soon became known by the name Oxford Professor C. A. R. Hoare gave it: the dining philosophers problem.

Imagine that five Hunan philosophers are sitting around a table. Each one has a bowl full of rice, and a chopstick on either side of the bowl. The right chopstick of each philosopher is the left chopstick of his neighbor. (See Figure 2.) Now, the rules for dining are as follows:

1. Each philosopher thinks for a while, eats for a while, and then waits for a while.
2. To eat, a philosopher must hold both his right and left chopsticks.
3. The philosophers communicate only by the lifting and lowering of chopsticks. (They can neither speak nor write.)

Suppose that each philosopher uses the following algorithm in order to eat:

(i) Pick up right chopstick when available (waiting if right neighbor has it).

(ii) Pick up left chopstick when available (waiting if left neighbor has it).

(iii) Eat.

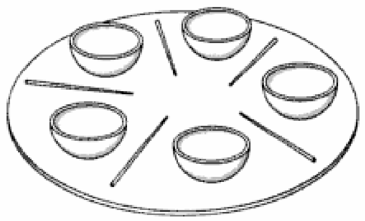

Figure 2
Dining philosophers problem. Each philosopher has one plate and one chopstick to his left and one chopstick to his right. To eat, he must hold both his left and right chopsticks, thereby preventing both of his neighbors from eating. The problem is to figure out a method by which each philosopher can eventually eat.

Several things can happen. If all philosophers decide to start eating at the same time, then they will all succeed at step (i), but will wait forever at step (ii). This situation is called "deadlock."

Seeing all his fellow philosophers holding only one chopstick, a philosopher waiting at step (ii) might put down his right chopstick and sit quietly for a while, watching his right neighbor eat. This gives rise to the possibility that an altruistic philosopher might never eat. This situation is called "starvation."

Even if all the philosophers do eat, some may eat more often than others. This situation is called "lack of fairness." Or life.

Variants of the dining philosophers problem crop up frequently in computer networks. For example, computers on a local area network often share a wire or broadcast channel over which only one message can be sent at a time. If all sites try to send at the same time, they all fail. If they then try again right away, they fail again. This is similar to deadlock. If one site always gets preference, then another site may starve or the protocol will be unfair.

For both philosophers and networks, one solution is randomization, as Michael Rabin will show us in the next chapter. If a philosopher or site cannot obtain a resource it needs, it waits an amount of time determined by some random process (e.g., a scintillation counter), then tries again. This scheme may still lead to starvation, since the process may be forever randomly unlucky, but the probability of such an event is small.

Several years after Dijkstra posed the dining philosophers problem, he was surprised to find that the designers of one of the most sophisticated computer systems then extant, M.I.T.'s MULTIX, had not thought about deadlock at all, and the system would, on occasion, abruptly stop—like so many philosophers each holding a single chopstick. With gentle irony, Dijkstra muses:

> You can hardly blame M.I.T. for not taking notice of an obscure computer scientist in a small town in the Netherlands.

Dijkstra in the United States

When Dijkstra accepted a job as Research Fellow for Burroughs Corporation, he used the position to push for verifiability in programming. But the discipline he advocates has proved to be distinctly unpopular, sometimes for cultural and sometimes for economic reasons.

> I think it was in 1970 that I gave my first talk in a foreign country on the design of programs that you could actually control and prove were correct. I gave the talk in Paris and it was a great success. On the way home, I gave the talk to a company in Brussels. The talk fell completely on its face. It turned out that management didn't like the idea at all. The company profits from maintenance contracts. The programmers didn't like the idea at all because it deprived them of the intellectual excitement of not quite understanding what they were doing. They liked the challenge of chasing the bugs.

Dijkstra's prodding, combined with the demand for high-quality software, has made the software industry significantly more disciplined. The one-line Dijkstra sound bite that every programmer knows is "GO TO considered harmful." GO TO causes a program to go from working on one job to working on something completely different without any plan for returning for the first job. Programs with many GO TO commands tend to be about as easy to follow as a legal contract in a Marx brothers' film.

Nevertheless, for many programmers, free-thinking hacking in a white heat of inspiration remains the ideal. Dijkstra views that kind of approach as a pathology.[1]

> People get attached to their sources of misery—that's what stabilizes many marriages.

Deep in the Heart of Mathematics

In the early 1980s, Dijkstra and his family moved to Austin, Texas, where he was awarded the Schlumberger Centennial Chair in Computer Sciences at the University of Texas. Now that their children are grown, the couple enjoys traveling in a Volkswagen camper they've

[1]Such habits impose an economic cost in a world in which roughly 40 percent of all software projects are canceled. About 70 percent of the rest are late.

nicknamed the Touring Machine. In his intellectual travels, Dijkstra has returned to mathematics in his quest for rigor.

> I am working on streamlining the mathematical argument: making the argument simpler, cleaner. It's really trying to transfer experience from programming into the wider area of mathematics.
>
> We all know that if you want to make something big, you have to compose it out of components—modules of some sort. You must be able to isolate parts. . . . It's well known from programming that this is not just a matter of division of labor because if you choose the wrong interface or an inappropriate one, the work explodes by a factor of ten—it's not just a sum.
>
> As an example, I have four composers living in different towns and they decide to compose a string quartet. You do the first movement, you do the adagio, you do the finale. Another way of dividing it is you do the first violin, you do the cello, you do the viola. In that latter distribution, an enormous amount of communication would be necessary between composers. That's a nice example of a practical and an impractical division of labor. Programmers have to think about this. A well-engineered mathematical theory has all the characteristics of the practical division of labor. The standard reaction of the inexperienced theorist who has demonstrated a complicated argument is to fall in love with that argument.

Dijkstra is impatient with definitions that cause problems for the reader. He once stopped reading Winston Churchill's *A History of the English Speaking Peoples* because "It was unnecessarily complicated. He would refer to the same people under different names." To Dijkstra, good definitions and a well-crafted argument are as essential as the idea itself.

> Whenever you are developing something new, you have tasks. You have to create a new subject matter. You have to create a language which is appropriate to discuss the subject matter. Many people are insufficiently aware of that second obligation.

Dijkstra's latest crusade for formalism in computer science and mathematics has spawned a book, *Formal Development of Programs and Proofs*. His central thesis equates writing programs with

writing a clear proof. Research on this thesis and related topics takes place in summer meetings, one near Munich and one in Glasgow. Americans are, for the most part, conspicuously absent.

> Americans have a pathological fear of formal manipulation. It seems that the United States has a century of demathematicization which of course is very tragic because in that same century the mathematical computer is invented which is a major mathematical challenge. Somehow or other the mathematical nature of the challenge seems to have been ignored here as politically unpalatable.

Often Dijkstra appears as a Nabokovian character, a cultured European in a land of cowboys. He is for that reason a controversial figure—younger computer scientists, while awed by his accomplishments, wonder if he is concerned with things that still matter. By the same token, Dijkstra has doubts about the topics chosen by his colleagues. For example, when asked where artificial intelligence fits within computer science, he replies wryly, "Not." To some extent, he sees artificial intelligence as a manifestation of American naiveté.

> The Europeans tend to maintain a greater distinction between man and machine and have lower expectations of both.

There are two sides of Edsger W. Dijkstra. On one side, we see the creative scientist whose love of good problems and enduring solutions have made enormous contribution to the science and practice of computing. On the other side, there is the impatient observer of human folly whose acerbic pen (often a Montblanc) has alienated many colleagues. Dijkstra insists that he is an easy person to understand.

> I'm very constant in my opinions and judgments—frighteningly so.

To young scientists seeking insight into his research methods, he offers three golden rules.

1. Never compete with colleagues.
2. Try the most difficult thing you can do.
3. Choose what is scientifically healthy and relevant. Don't compromise on scientific integrity.