**Project 1**

**Rearranging Railroad Cars**

**Problem Description**

A freight train has n railroad cars. Each is to be left at a different station. Assume that the n stations are numbered 1 through n and that the freight train visits these stations in the order n through 1. The railroad cars are labeled by their destination. To facilitate removal of the railroad cars from the train, we must reorder the cars so that they are in the order 1 through n from front to back. When the cars are in this order, the last car is detached at each station. We rearrange the cars at a shunting yard that has an *input track*, an *output track*, and k holding tracks between the input and output tracks.

[581742963] [987654321]

H3

H3

H2

H1

output track

H2

H1

Input track

1. Initial (b) Final

Figure 1 A three-track example

Figure 1 shows a shunting yard with k=3 holding tracks H1, H2 and H3. The n cars of the freight train begin in the input track and are to end up in the output track in the order 1 through n from right to left. In figure 1 (a), n=9; the cars are initially in the order 5, 8, 1, 7, 4, 2, 9, 6, 3 from back to front. Figure 1(b) shows the cars rearranged in the desired order.

**Solution Strategy**

To rearrange the cars, we examine the cars on the input track from front to back. If the car being examined is the next one in the output arrangement, we move it directly to the output track. If not, we move it to a holding track and leave it there until it is time to place it in the output track. The holding tracks operate in a LIFO manner as cars enter and leave these tracks from the top. When rearranging cars, only the following moves are permitted:

* A car may be moved from the front (i.e., right end) of the input track to the top of one of the holding tracks or to the left end of the output track.
* A car may be moved from the top of a holding track to the left end of the output track.

Consider the input arrangement of Figure 1. Car 3 is at the front and cannot be output yet, as it is to be preceded by cars 1 and 2. So car 3 is detached and moved to the holding track H1. The next car, car 6, is also to be moved to a holding track. If car 6 is moved to H1, the rearrangement cannot be completed because car 3 will be below car 6. However, car 3 is to be output before car 6 and so must leave H1 before car 6 does. So car 6 is put into H2. The next car, car 9, is put into H3 because putting it into either H1 or H2 will make it impossible to complete the rearrangement. *Notice that whenever the car labels in a holding track are not in increasing order from top to bottom, the rearrangement cannot be completed.* The current state of the holding tracks is shown in Figure 2(a).

7

4

2

9

6

3

9

6

3

**H1 H2 H3 H1 H2 H3**

1. **(b)**

**Figure 2 Track states**

Car 2 is considered next. It can be moved into any of the holding tracks while satisfying the requirements that car labels in any holding track be in increasing order, but moving it to H1 is preferred. If car 2 is moved to H3, then we have no place to move cars 7 and 8. If we move it to H2, then the next car, car 4, will have to be moved to H3 and we will have no place for cars 5, 7 and 8. The least restrictions on future car placement arise when the new car u is moved to the holding track that has at its top a car with smallest label v such that v > u. We will use this assignment rule to select the holding track.

When car 4 is considered, the cars at the top of the three holding tracks are 2, 6, and 9. Using our assignment rule, car 4 is moved to H2. Car 7 is then moved to H3. Figure 2 (b) shows the current state of the holding tracks. The next car, car 1, is moved to the output track. It is now time to move car 2 from H1 to the output track. Next car 3 is moved from H1, and then car 4 is moved from H2. No other cars can be moved to the output at this time.

The next input car, car 8, is moved to H1. Then car 5 is moved from the input track to the output track. Following this move, car 6 is moved from H2. Then car 7 is moved from H3, car 8 from H1, and car 9 from H3.

While three holding tracks are sufficient to rearrange the cars from the initial ordering of Figure 1(a), other initial arrangements may need more tracks. For example, the initial arrangement 1, n, n-1, …, 2 requires n-1 holding tracks.

**Project 2**

**Parenthesis Matching**

**Problem Description**

In this problem we are to match the left and the right parentheses in a character string. For example, the string (a\*(b + c) + d) has left parentheses at positions 0 and 3 and right parentheses at positions 7 and 10. The left parenthesis at position 0 matches the right at position 10, while the left parenthesis at position 3 matches the right parenthesis at position 7. In the string ( a + b)) (, the right parenthesis at position 5 has no matching left parenthesis, and the left parenthesis at position 6 has no matching right parenthesis. The objective is to write a JAVA program that inputs a string and outputs the pairs of matched parentheses as well as those parentheses for which there is not match. Notice that the parenthesis matching problem is equivalent to the problem of matching braces ({ and }) in a JAVA program.

**Solution Strategy**

We observe that if we scan the input expression from left to right, then each right parenthesis is matched to the most recently seen unmatched left parenthesis. This observation motivates us to save the position of left parentheses on a stack as they are encountered in a left-to-right scan. When a right parenthesis is encountered, it is matched to the left parenthesis (if any) at the top of the stack. The matched left parenthesis is deleted from the stack.

**Project 3**

**Switch Box Routing**

**Problem Description**

In the switch box routing problem, we are given a rectangular routing region with pins at the periphery. Pairs of pins are to be connected together by laying a metal path between the two pins. This path is confined to the routing region and is called a wire. If two wires intersect, an electrical short occurs. So wire intersections are forbidden. Each pair of pins that is to be connected is called a **net**. We are to determine whether the given nets can be routed with no intersections. Figure 3 (a) shows a sample switch box instance with eight pins and four nets. The nets are (1,4), (2,3), (5,6), and (7, 8). The wire routing of Figure 2(b) has a pair of intersecting wires (those for nets (1, 4) and (2, 3)), whereas the routing of Figure 3(c) has no intersections. Since the four nets can be routed with no intersections, the given switch box is a **routable switch box**. Our problem is to input a switch box routing instance and determine whether it is routable.

While the wires in both figures 3(b) and (c) are composed of straight line segments parallel to the x- and y- axes, segments that are not parallel to these axes as well as segments that are not straight lines are permissible.

1 2

1 2

1 2

8

7

6

3

4

3

4

8

7

6

routing

region

8

7

6

5

3

4

5

5

1. (b) (c)

Figure 3 Sample switch box

**Solution Strategy**

To solve the switch box routing problem, we note that when a net is connected, the wire partitions the routing region into two regions. The pins that fall on the boundary of a partition do not depend on the wire path, but only on the pins of the net that was routed. For instance, when net (1, 4) is routed, we get two regions. One contains the pins 2 and 3, and the other contains the pins 5 through 8. If there is now a net with one pin in one region and the other in a different region, this new net cannot be routed and the routing instance is unroutable. If there is not net with this property, then since the wires cannot cross between regions, we can attempt to determine whether each region is independently routable. To make this determination, we pick a net in one of the regions; this net partitions its region into two regions, and none of the remaining nets should have a pin in one partition and another in the other partition.

We can implement this strategy by moving around the periphery of the switch box in either clockwise or counterclockwise order, beginning at any pin. If we traverse the pins of Figure 3(a) in clockwise order, beginning at pin 1, the pins are examined in the order, 1, 2, …, 8. The pins that lie between pin 1 and its net partner, pin 4, define one region of the first partition, and those that lie between pins 4 and 1 define the other. We will place pin 1 on a stack and then continue processing pins until pin 4 is encountered. This procedure allows us to process one of the regions before going on to the other. The next pin, pin 2, and its net partner, pin 3, partition the current region into two regions. As before, pin 2 is placed on the stack, and we proceed to pin 3. Since pin 3’s partner, pin 2, is at the top of the stack, we have completed a region and pin 2 is deleted from the stack. Proceeding in this way, we are able to complete the processing of all created regions, and the stack is empty after pin 8 is examined.

What happens to a nonroutable instance? Suppose the nets for Figure 3(a) are (1, 5), (2, 3), (4, 7) and (6, 8). Pins 1 and 2 are put on the stack initially. When pin 3 is examined, pin 2 is deleted from the stack. Next pin 4 is added to the stack, as pin 4 and the pin at the stack top do not define a region boundary. When pin 5 is examined, it is also added to the stack. Even though pins 1 and 5 have both been seen, we are unable to complete the processing of the first region defined by this net, as pin 4’s routing has to cross the boundary. As a result, when we complete the examination of all pins, the stack will not be empty.

**Projects 4-5-6**

**Union-Find Problem**

**Equivalence Classes**

Suppose we have a set U = { 1, 2, …, n} of n elements and a set

R = {(i1, j1), (i2, j2), …, (ir, jr) of r relations. The relation R is an equivalence relation iff the following conditions are true:

* (a, a) R for all a U (the relation is reflexive).
* (a,b) R iff (b, a) R (the relation is symmetric).
* (a, b) R and (b, c) R imply that (a, c) R (the relation is transitive).

Often when we specify an equivalence relation R, we omit some of the pairs in R. The omitted pairs may be obtained by applying the reflexive, symmetric, and transitive properties of an equivalence relation.

**Example 1**. Suppose n = 14 and R = {(1, 11), (7, 11), (2, 12), (12, 8),

(11, 12), (3, 13), (4, 13), (13, 14), (14, 9), (5, 14), (6, 10)}. We have omitted all pairs of the form (a, a) because these pairs are implied by the reflexive property. Similarly, we have omitted all symmetric pairs. Since (1, 11) R, the symmetric property requires (11, 1) . Other omitted pairs are obtained by applying the transitive property. For example, (7, 11) and (11, 12) imply (7, 12) .

Two elements a and b are equivalent if (a, b) An equivalence class is defined to be a maximal set of equivalent elements. Maximal means that no elements outside the class is equivalent to an element in the class. Since it is not possible for an element to be in more than one equivalence class, an equivalence relation partitions the universe U into disjoint classes.

**Example 2** Consider the equivalence relation in Example 1. Since elements 1 and 11, and 11 and 12 are equivalent, elements 1, 11, and 12 are equivalent. They are therefore in the same class. These three elements do not, however, form an equivalence class, as they are equivalent to other elements (e.g., 7). So {1, 11, 12} is not a maximal set of equivalent elements. The set {1, 2, 7, 8, 11, 12} is an equivalence class. The relation R defines two other equivalence classes: {3, 4, 5, 9, 13, 14} and {6, 10}. Notice that the three equivalence classes are disjoint.

In the **offline equivalence class** problem, we are given n and R and we need to determine the equivalence classes. From the definition of an equivalence class, it follows that each element is in exactly one equivalence class. In the **online equivalence class** problem, we begin with n elements, each in a separate equivalence class. We are to process a sequence of the operations: (1) combine (a, b), …combines the equivalence classes that contain elements a and b into a single class and (2) find(theElement) … determines the class that currently contains element theElement. The purpose of the find operation is to determine whether two elements are in the same class. Hence the find operation is to be implemented to return the same answer for elements in the same class and different answers for elements in different classes.

We can write the combine operation in terms of two **finds** and **a union** that actually takes two different classes and makes one. So combine (a, b) is equivalent to

classA = find(a);

classB = find (b);

if (classA != classB)

union(classA, classB);

Notice that with the find and union operations, we can add new relations to R. For instance, to add the relation (a, b) , we determine whether a and b are already in the same class. If they are, then the new relation is redundant. If they are not, then we perform a **union** on the two classes that contain a and b.

The online equivalence problem is more commonly known as the **union-find** problem.

**Applications**

A machine-scheduling problem and a circuit-wiring problem may be modeled as online equivalence class problems.

**Example 3**  A certain factory has a single machine that is to perform n tasks. Task i has an integer release time ri and an integer deadline di. The completion of each task requires one unit of time on this machine. A **feasible schedule** is an assignment of tasks to time slots on the machine such that task i is assigned to a time slot between its release time and deadline and no slot has more than one task assigned to it.

Consider the following four tasks:

Task A B C D

Release time 0 0 1 2

Deadline 4 4 2 3

Tasks A and B are released at time 0, task C is released at time 1, and task D is released at time 2. The following task-to-slot assignment is a feasible schedule: do task A from 0 to 1; task C from 1 to 2; task D from 2 to 3; and task B from 3 to 4

0 1 2 3 4

C

B

D

A

Figure 4 A schedule for four tasks

**Solution Strategy**

An intuitively appealing method to construct a schedule is

1. Sort the tasks into nonincreasing order of release time.
2. Consider the tasks in this nonincreasing order. For each task determine the free slot nearest to, but not after, its deadline. If this free slot is before the task’s release time, fail. Otherwise, assign the task to this slot.

This strategy fails to find a feasible schedule only when such a schedule does not exist.

The online equivalence class problem can be used to implement step (2). For this step, let d denote the latest deadline of any task. The usable time slots are of the form “from *i-1* to *i*” where 1≤ i ≤ d. We will refer to these usable slots as slots 1 through d. For any slot a, define *near(a)* as the largest i such that i≤ a and slot i is free. If no such I exists, define *near(a) = near(0) = 0*. Two slots a and b are in the same equivalence class iff *near(a) = near(b).*

Prior to the scheduling of any task, *near(a) = a* for all slots, and each slot is in a separate equivalence class. When slot *a* is assigned a task in step (2), *near* changes for all slots *b* with *near(b) = a*. For these slots the new value of near is *near(a-1)*. Hence when slot a is assigned a task, we need to perform aunion on the equivalence classes that currently contain slots *a* and *a-1*. If with each equivalence class e we retain, in *nearest[e],* the value of *near* of its members, then near(a) is given by nearest[find(a)]. (Assume that the equivalence class name is taken to be whatever the find operation returns.)

Write a Java program for the scheduling problem in Example 3. Model the problem as the online equivalence class problem and use the chain method. Test the correctness of your program.

**Example 4. [From Wires to Nets]** An electronic circuit consists of components, pins, and wires. Figure 5 shows a circuit with the three components A, B, and C.

C

B

A

Figure 5. A three-chip circuit on a printed circuit board.

wires

pins

Each wire connects a pair of pins. Two pins a and b are **electrically equivalent**  if they are either connected by a wire or there is a sequence i1, i2, …, ik of pins such that a, i1; i1, i2; i2, i3; … ; ik-1, ; and ik, b are all connected by wires. A net is a maximal set of electrically equivalent pins. Maximal means that no pin outside the net is electrically equivalent to a pin in the net.

Consider the circuit shown in Figure 6.

5

3

4

2

1●●

14

11

13

12

10

9

8

7

Figure 6 Circuit with pins and wires shown

In this figure only the pins and the wires have been shown. The 14 pins are numbered 1 through 14. Each wire may be described by the two pins that it connects. For instance, the wire connecting pins 1 and 11 is described by the pair (1, 11), which is equivalent to the pair (11, 1). The set of wires is {(1, 11), (7, 11), (2, 12), (12, 8), (11, 12), (3, 13), (4, 13), (13, 14), (14, 9), (5, 14), (6, 10)}. The nets are {1, 2, 7, 8, 11, 12}, {3, 4, 5, 9, 13, 14} and {6, 10}.

In the **offline net finding problem**, we are given the pins and wires and are to determine the nets. This problem is modeled by the offline equivalence problem with each pin being a member of U and each wire a member of R.

In the **online** version we begin with a collection of pins and no wires and are to perform a sequence of operations of the form (1) add a wire to connect pins a and b and (2) find the net that contains pin a. The purpose of the find operation is to determine whether two pins are in the same net or in different nets. This version of the net problem may be modeled by the online equivalence class problem. Initially, there are no wires, and we have R = . The net find operation corresponds to the equivalence class find operation and adding a new wire (a, b) corresponds to combine(a,b), which is equivalent to

union(find(a), find(b)).

Write a Java program for the online net finding problem described in Example 4. Model the problem as the online equivalence class problem and use the chain method. Test the correctness of your program.

**First Union-Find Solution**

A simple solution to the online equivalence class problem is to use an array equiv-Class and let equivClass[i] be the class that currently contains element i. Create methods to initialize, union, and find. n is the number of elements. N and equivClass are both assumed to be static data members. To unite two different classes, we arbitrarily pick one of these classes and change the equivClass values of all elements in this class to correspond to the equivClass values of the elements of the other class.

**Second Union-Find Solution**

The time complexity of the union operation can be reduced by keeping a chain for each equivalence class because we can find all elements in a given equivalence class by going down the chain for that class, rather than by examining all equivClass values. If each equivalence class knows its size, we can choose to change the equivClass values of the smaller equivalence class and perform the union operation faster. By using simulated pointers, we get quick access to the node that represents element e. We adopt the following conventions:

* EquivNode is a class with data members equivClass, size, and next.

Class EquivNode

{

int equivClass; //element class identifier

int size; // size of the class

int next; // pointer to next element in class

/\*\* constructor \*/

EquivNode(int theClass, int theSize)

{

equivClass = theClass;

size = theSize;

// next has the default value 0

}

}

* An array node[1:n] of type EquivNode is used to represent the n elements together with the equivalence class chains.
* node[e].equivClass is both the value to be returned by find(e) and a pointer to the first node in the chain for the equivalence class node[e].equivClass.
* node[e].size is defined only if e is the first node on a chain. In this case node[e].size is the number of nodes on the chain that begins at node[e].
* node[e].next gives the next node on the chain that contains node e. Since the nodes in use are numbered 1 through n, a null pointer can be simulated by 0 rather than by -1.

**Offline Equivalence Class Problem**

**Problem Description**

The inputs to the offline equivalence problem are the number of elements n, the number of relation pairs r, and the r relation pairs. We are to partition the n elements into equivalence classes.

**Solution Strategy**

The solution is in two phases. In the first phase we input the data and set up n lists to represent the relation pairs. For each relation pair (i, j), I is put on list[j] and j is put on list[i].

**Example 5**. Suppose that n = 9, r = 11, and the 11 relation pairs are

(1, 5), (1, 6), (3, 7), (4, 8), (5, 2), (6, 5), (4, 9), (9, 7), (7, 8), (3, 4), and

(6, 2). The nine lists are

list[1] = [5, 6]

list[2] = [5, 6]

list[3] = [7, 4]

list[4] = [8, 9, 3]

list[5] = [1, 2, 6]

list[6] = [1, 2, 5]

list[7] =[3, 9, 8]

list[8] = [4, 7]

list[9] = [4, 7]

Element order within the list is not important.

In the second phase, the equivalence classes are identified by first locating an element that has not been output as part of an equivalence class. This element becomes the **seed** for the next equivalence class. From the seed we identify all other members of the class as follows. The seed is put onto a list, *unprocessedList*, of elements that are in the same equivalence class as the seed and whose lists have yet to be processed. We remove an element i from *unprocessedList* and process *list[i]*. All elements on *list[i]* are in the same equivalence class as the seed; elements on *list[i]* that have not already been identified as class members are output and added to *unprocessedList*. This process of removing an element i from *unprocessedList* and then outputting and adding elements in *list[i]*  that have not already been output to *unprocessedList*  continues until the *unprocessedList*  becomes empty. At this time we have completed a class, and we proceed to find a seed for the next class.

Consider the data of Example 5. Let 1 be the first seed; 1 is output as a part of a new class and is also added to *unprocessedList* . Next 1 is removed from *unprocessedList,* and *list[1]*  is processed. The elements 5 and 6 that are in *list[1]*  are output as part of the same class as element 1; 5 and 6 are also added to *unprocessedList.* Either 5 or 6 is removed from *unprocessedList*, and its list is processed. Suppose that 5 is removed. The elements 1, 2, and 6 that are in *list[5]* are examined. Since 1 and 6 have already been output, we ignore them. Element 2 is output and added to *unprocessedList.* When the remaining elements (6 and 2) that are in *unprocessedList* are removed and processed, no additional element is output or added to *unprocessedList*; this list becomes empty, and we have identified an equivalence class.

To find another equivalence class, we search for a seed – an element not yet output. Element 3 has not been output and is used as the seed for the next class. Elements 3, 4, 7, 8, and 9 are output as part of this next class. Since no seeds remain, we have found all the classes.

Implement in Java the offline equivalence class problem.

**Project 7**

**Image-Component Labeling**

**Problem Description**

A digitized image is an m X m matrix of pixels. In a binary image each pixel is either 0 or 1. A 0 pixel represents image background, while a 1 represents a point on the image component. We will refer to pixels whose value is 1 as component pixels. Two pixels are adjacent if one is to the left, above, right or below the other. Two component pixels that are adjacent are pixels of the same image component. The objective of component labeling is to label the component pixels so that two pixels get the same label if and only if they are pixels of the same image component.

Consider Figure 7(a) that shows a 7 X 7 image. The blank squares represent background pixels, and the 1s represent component pixels. Pixels (1,3) and (2,3) are pixels of the same component because they are adjacent. Since component pixels (2,3) and (2,4) are adjacent, they are also from the same component. Hence the three pixels (1,3), (2,3), and (2,4) are from the same component. Since no other image pixels are adjacent to these three pixels, these three define an image component. The image of figure 7(a) has four components. The first component is defined by the pixel set (1,3), (2,3), (2,4); the second is (3,5), (4,4), (4,5), (5,5); the third is (5,2), (6,1), (6,2), (6,3), (7,1), (7,2), (7,3); and the fourth is (5,7), (6,7), (7,6), (7,7). In Figure 7(b) the component pixels have been given labels so that two pixels have the same label if and only if they are part of the same component. We use the numbers 2,3,4,… as component identifiers; there is no component numbered 1 because 1 designates an unlabeled component pixel.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 |  |  |  |  |
|  |  | 1 | 1 |  |  |  |
|  |  |  |  | 1 |  |  |
|  |  |  | 1 | 1 |  |  |
|  | 1 |  |  | 1 |  | 1 |
| 1 | 1 | 1 |  |  |  | 1 |
| 1 | 1 | 1 |  |  | 1 | 1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 2 |  |  |  |  |
|  |  | 2 | 2 |  |  |  |
|  |  |  |  | 3 |  |  |
|  |  |  | 3 | 3 |  |  |
|  | 4 |  |  | 3 |  | 5 |
| 4 | 4 | 4 |  |  |  | 5 |
| 4 | 4 | 4 |  |  | 5 | 5 |

1. A 7 x 7 image (b) Labeled components

Figure 7 Image-component labeling

**Problem 8**

**Wire Routing**

**Problem Description**

The problem of finding a shortest path in a grid has many applications. For example, a common approach to the wire-routing problem for electrical circuits is to impose a grid over the wire-routing region. The grid divides the routing region into an nxm array of squares much like a maze. A wire runs from the midpoint of one square a to the midpoint of another b. In doing so, the wire may make right-angle turns. Grid squares that already have a wire through them are blocked. To minimize signal delay, we wish to route the wire using a shortest path between a and b.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

1. A 7x7 grid

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | a |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  | b |  |  |  |
|  |  |  |  |  |  |  |

1. A wire between a and b

Figure 8 Wire-routing example

**Solution Strategy**

We have already solved the maze traversal problem. The shortest path between grid positions a and b is found in two passes – a distance-labeling pass and a path-identification pass. In the distance-labeling pass, we begin at position a and label its reachable neighbors 1 (i.e., they are at distance 1 from a). Next the reachable neighbors of squares labeled 1 are labeled 2. This labeling process is continued until we either reach b or have no more reachable neighbors. Figure 9 shows the result of the distance-labeling pass for the case a = (3, 2) and b = (4, 6). The shaded squares are blocked squares.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 2 |  |  |  |  |  |
| 2 | 1 |  |  |  |  |  |
| 1 | a | 1 | 2 |  |  |  |
| 2 | 1 | 2 |  |  | b |  |
|  | 2 | 3 | 4 |  | 8 |  |
|  |  |  | 5 | 6 | 7 | 8 |
|  |  |  | 6 | 7 | 8 |  |

1. Distance labeling

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | a |  |  |  |  |  |
|  |  |  |  |  | b |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

1. Wire path

Figure 9 Wire routing

Once we have reached b, we can label it with its distance (9 in the case of Figure 9). The distance-labeling pass is followed by the path-identification pass in which we begin at b and move to any one of its neighbors labeled 1 less than b’s label. Such a neighbor must exist as each grid’s label is 1 more than that of at least one of its neighbors. In the case of Figure 9 (a), we move from b to (5, 6). From here we move to one of its neighbors whose label is 1 less and so on until we reach a. In the example of figure 9, from (5, 6) we move to (6, 6) and then to (6, 5), (6, 4), (5, 4), and so on. Figure 9 (b) shows the constructed path, which is a shortest path between (3, 2) and (4, 6). Notice that the shortest path between

(3, 2) and (4, 6) is not unique; (3, 2), (3, 3), (4, 3), (5, 3), (5, 4), (6, 4), (6, 5), (6, 6), (5, 6), (4, 6) is another shortest path.

Implement in JAVA the Wire Routing problem.

Problem 9 **Machine Shop Simulation**

**Problem Description**

A machine shop (or factory or plant) comprises m machines or workstations. The machine shop works on jobs, and each job comprises several tasks. Each machine can process one task of one job at any time, and different machines perform different tasks. Once a machine begins to process a task, it continues processing that task until the task completes.

Example 6. A sheet metal plant might have one machine (or station) for each of the following tasks: design; cut the sheet metal to size; drill holes; cut holes; trim edges; shape the metal; and seal streams. Each of these machines/stations can work on one task at a time.

Each job includes several tasks. For example, to fabricate the heating and air-conditioning ducts for a new house, we would need to spend some time in the design phase, and then some time cutting the sheet metal stock to the right size pieces. We need to drill or cut the holes (depending on their size), shape the cut pieces into ducts, seal the seams, and trim any rough edges.

For each task of a job, there is a task time (how long does it take), and a machine on which it is to be performed. The tasks of a job are to be performed in a specific order. So a job goes first to the machine for its first task. When this first task is complete, the job goes to the machine for its second task, and so on until its last task completes. When a job arrives at a machine, the job may have to wait because the machine might be busy. In fact, several jobs may already be waiting for that machine.

Each machine in our machine shop can be in one of three states: active idle, and change over. In the active state the machine is working on a task of some job; in the idle state it is doing nothing; and in the change-over state the machine has completed a task and is preparing for a new task. In the change-over state, the machine operator might, for example, clean the machine, put away tools used for the last task, and take a break. The time each machine must spend in its change-over state depends on the machine.

When a machine becomes available for a new job, it will need to pick one of the waiting jobs to work on. We assume that each machine serves its waiting jobs in a FIFO manner, and so the waiting jobs at each machine form a (FIFO) queue. Other assumptions for the selection of the next job are possible. For example, the next job may be selected by priority. Each job has a priority, and when a machine becomes free, the waiting job with the highest priority is selected.

The time at which a job’s last task completes is called its **finish time**. The length of a job is the sum of its task times. If a job of length *l* arrives at the machine shop at time 0 and completes at time *f*, then it must have spent exactly *f-l* amount of time waiting in machine queues. To keep customers happy, it is desirable to minimize the time a job spends waiting in machine queues. Machine shop performance can be improved if we know how much time jobs spend waiting and which machines are contributing most to this wait.

**How the Simulation Works**

When simulating a machine shop, we follow the jobs from machine to machine without physically performing the tasks. We simulate time by using a simulated clock that is advanced each time a task completes or a new job enters the machine shop. As tasks complete, new tasks are scheduled. Each time a task completes or a new job enters the shop, we say that an **event** has occurred. In addition, a start event initiates the simulation. When two or more events occur at the same time, we arbitrarily order these events. Figure 10 describes how a simulation works.

//initialize

Input the data;

Create the job queues at each machine;

Schedule first job in each machine queue;

// do the simulation

while (an unfinished job remains)

{

determine the next event;

if (the next event is the completion of a machine change over)

schedule the next job (if any) from this machine’s queue;

else

{ // a job task has completed

put the machine that finished the job task into its change-over

state;

move the job whose task has finished to the machine for its next

task (if any);

}

}

Figure 10 The mechanics of simulation

Example 7 Consider a machine shop that has m = 3 machines and n = 4 jobs. We assume that all four jobs are available at time 0 and that no new jobs become available during the simulation. The simulation will continue until all jobs have completed.

The three machines, M1, M2 and M3, have a change-over time of 2, 3, and 1, respectively. So when a task completes, machine 1 must wait two time units before starting another, machine 2 must wait three time units, and machine 3 must wait one time unit. Figure 11 (a) gives the characteristics of the four jobs. Job 1, for example, has three tasks. Each task is specified as a pair of the form (machine, time). The first task of job 1 is to be done on M1 and takes two time units, the second is to be done on M2 and takes four time units, and the third is to be done on M1 and takes one time unit. The job lengths (the sum of their task times) are 7, 6, 8, and 4, respectively.

|  |  |  |  |
| --- | --- | --- | --- |
| Job # | # Tasks | Tasks | Length |
| 1 | 3 | (1,2 ) (2,4) (1,1) | 7 |
| 2 | 2 | (3, 4) (1,2 ) | 6 |
| 3 | 2 | (1, 4) (2, 4) | 8 |
| 4 | 2 | (3, 1) (2, 3) | 4 |

1. Job characteristics

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time | Machine Queues | | | Active Jobs | | | Finish Times | | |
|  | M1 | M2 | M3 | M1 | M2 | M3 | M1 | M2 | M3 |
| Init | 1,3 | - | 2, 4 | I | I | I | L | L | L |
| 0 | 3 | - | 4 | 1 | I | 2 | 2 | L | 4 |
| 2 | 3 | - | 4 | C | 1 | 2 | 4 | 6 | 4 |
| 4 | 2 | - | 4 | 3 | 1 | C | 8 | 6 | 5 |
| 5 | 2 | - | - | 3 | 1 | 4 | 8 | 6 | 6 |
| 6 | 2,1 | 4 | - | 3 | C | C | 8 | 9 | 7 |
| 7 | 2,1 | 4 | - | 3 | C | I | 8 | 9 | L |
| 8 | 2,1 | 4,3 | - | C | C | I | 10 | 9 | L |
| 9 | 2,1 | 3 | - | C | 4 | I | 10 | 12 | L |
| 10 | 1 | 3 | - | 2 | 4 | I | 12 | 12 | L |
| 12 | 1 | 3 | - | C | C | I | 14 | 15 | L |
| 14 | - | 3 | - | 1 | C | I | 15 | 15 | L |
| 15 | - | - | - | C | 3 | I | 17 | 19 | L |
| 16 | - | - | - | C | 3 | I | 17 | 19 | L |
| 17 | - | - | - | I | 3 | I | L | 19 | L |
|  |  |  |  |  |  |  |  |  |  |

1. Simulation

|  |  |  |
| --- | --- | --- |
| Job # | Finish Time | Wait Time |
| 1 | 15 | 8 |
| 2 | 12 | 6 |
| 3 | 19 | 11 |
| 4 | 12 | 8 |
| Total | 58 | 33 |

1. Finish and wait times

Figure 11 Machine shop simulation example

Figure 11 (b) shows the machine shop simulation. Initially, the four jobs are placed into queues corresponding to their first tasks. The first task for jobs 1 and 3 are to be done on M1, so these jobs are placed on the queue for M1. The first tasks for jobs 2 and 4 are to be done on M3. Consequently, these jobs begin on the queue for M3. The queue for M2 is empty. At the start all three machines are idle. We use the symbol I to indicate that the machines have no active job at this time. Since no machine is active, the time at which they will finish their current active task is undefined and denoted by the symbol L (large time).

The simulation begins at time 0. That is, the first event, the start event, occurs at time 0. At this time the first job in each machine queue is scheduled on the corresponding machine. Job 1’s first task is scheduled on M1, and job 2’s first task on M3. The queue for M1 now contains job 3 only, while that for M3 contains job 4 only. The queue for M2 remains empty. Job 1 becomes the active job on M1, and job 2 the active job on M3. M2 remains idle. The finish time for M1 becomes 2 (current time of 0 plus task time of 2), and the finish time for M3 becomes 4.

The next event occurs at time 2. This time is determined by finding the minimum of the machine finish times. At time 2 machine M1 completes its active task. This task is a job 1 task. Job 1 is moved to machine M2 for the next task. Since M2 is idle, the processing of job 1’s second task begins immediately. This task will complete at time 6 (current time of 2 plus task time of 4). M1 goes into its change-over state and will remain in this state for two time units. Its active job is set to C (change over), and its finish time is set to 4.

At time 4 both M1 and M3 complete their active tasks. As machine M1 completes a change-over task, that machine begins a new job; selecting the first job, job 3, from its queue. Since the task length for job 3’s next task is 4, the task will complete at time 8 and the finish time for M1 becomes 8. The next task for job 2, which just completed its first task on machine M3, needs to be done on M1. Since M1 is busy, job 2 is added to M1’s job queue. M3 moves into its change-over state and completes this change-over task at time 5. You should now be able to follow the remaining sequence of events.

Figure 11 (c) gives the finish and wait times. Since the length of job 2 is 6 and its finish time is 12, job 2 must have spent a total of 12-6 = 6 time units waiting in machine queues. Similarly, job 4 must have spent 12-4 = 8 time units waiting in queues.

We may determine the distribution of the 33 units of total wait time across the three machines. For example, job 4 joined the queue for M3 at time 0 and did not become active until time 5. So this job waited at M3 for five time units. No other job experienced a wait at M3. The total wait time at M3 was, therefore, five time units. Going through Figure 11 (b), we can compute the wait times for M1 and M2. The numbers are 18 and 10, respectively. As expected the sum of the job wait times (33) equals the sum of the machine wait times.

High-Level Simulator Design

In designing the simulator, we will assume that all jobs are available initially ( no jobs enter the shop during the simulation). Further, we assume that the simulation is to be run until all jobs complete.

We can define a class MachineShopSimulator. The simulator is a complex program, so we have to break it into modules. The tasks to be performed by the simulator are input the data and put the jobs into the queues for their first tasks; perform the start event (do the initial loading of jobs onto the machines); run through all the events (perform the actual simulation); and output the machine wait times. You have to create a JAVA method for each task. Here is the main method:

Public static void main(String [] args)

{

largeTime = Integer.MAX\_VALUE;

inputData(); // get machine and job data

startShop(); // initial machine loading

simulate(); // run all jobs through shop

outputStatistics(); // output machine wait times

}

Main method for machine shop simulation

The variable largeTime is a class data member of MachineShopSimulator. This variable denotes a time that is larger than any permissible simulated time; that is all tasks of all jobs must complete before the time largeTime.

Classes:

* Class Task
* Class Job
* Class Machine
* Class EventList

Data Members of MachineShopSimulator:

private static int timeNow; //current time

private static int numMachines; // number of machines

private static int numJobs; // number of jobs

private static EventList eList; // pointer to event list

private static Machine [] machine; // array of machines

private static int largeTime; // all machines finish before this