**Instituto Tecnológico de Costa Rica**

**Análisis de Algoritmos**

**Grupo 50**

**Profesora**

Ana Lorena Valerio Solís

**Proyecto Programado # 2**

**Assembly Line Balancing Problem (ALBP)**

**Integrantes**

Nelson Andrey Vega Soto

Yuliana Rojas Serrano

**Introducción:**

Con la elaboración del primer proyecto programado de análisis de algoritmos del segundo semestre del 2019, se pretende poner a prueba nuestras habilidades aprendidas en clases, para esto se nos ha brindado un enunciado en el cual se detalla la elaboración del proyecto.

Este consiste en realizar una serie de algoritmos (2 en total) con funciones específicas cada uno, esto con el fin de poder obtener el balanceo en una línea de ensamblaje, para esto se implementarán dos tipos de algoritmos, uno centrado en programación dinámica y el otro un algoritmo genético.

Este proyecto tiene el propósito de medir las habilidades a la hora de generar algoritmos eficaces y además de implementar una medición correcta a cada uno de ellos, esto para permitirnos observar su comportamiento y crecimiento con el fin de analizarlos.

El balanceo de una línea de ensamblaje es de suma importancia para cualquier fabrica, empresa o negocio que utilicen una línea de ensamblaje para crear sus productos, entre mejor este balanceada, menos tiempos gastara y por ende menos recursos, es por esto por lo que se pretende generar dos algoritmos muy eficaces que permitan realizar el balanceo de esta.

**Análisis del Problema:**

Ante la situación de mejorar el balanceo de una línea de ensamblaje de cualquier empresa, se pretende generar dos algoritmos eficaces, capaces de resolver este problema con el mayor rendimiento posible para la empresa.

Para esto se nos ha brindado la tarea de crear un algoritmo que implique la programación dinámica y otro que se base en la programación con algoritmos genéticos.

En una empresa, el balanceo de línea de ensamblaje es de suma importancia para el ahorro de los costos en la producción de sus artículos. Entre menos tiempo ocioso tengan las estaciones de tareas, mayor será el ahorro y mayor será la producción de los artículos.

Además, se pretenden realizar mediciones a los algoritmos con el propósito de observar su comportamiento, ver la cantidad de memoria invertida en el proceso, la cantidad de asignaciones y la cantidad de comparaciones que debe realizar para alcanza su objetivo.

**Solución del Problema**

Para la solución de este problema, se utilizarán ArrayList de objetos creados, con el propósito de almacenar la información de la Línea de Ensamblaje.

Para poder crear la Línea de Ensamblaje Aleatoria, se utilizan las clases almacenadas en el Package **Ensamblaje**, este almacenara dos clases, una de nombre ensamblaje y otra de nombre estación.

En ensamblaje se encuentra el método que permite crear cierto número de tareas, esto con el propósito de crear la línea de ensamblaje de 120 y 1200.

Además, en el Main del proyecto, se encuentran creados las cuatro primeras líneas de ensambles (10, 20, 30 y 60 tareas), en la misma clase se corren los algoritmos y se establece un menú con el propósito de mejorar la ejecución de las consultas.

**Para crear el algoritmo Dinámico se aplicó la siguiente estrategia.**

Se creo una función **buscarPosibles()** que permite buscar todas las posibles tareas que se puedan insertar en la estación, para esto se analizaran cuales tareas tienen todos los precedentes ya utilizados, esto se hace con la función **precedentesUsados()**, si esta función retorna true indica que todos los precedentes de la tarea evaluada están utilizados, sino retorna false, la función **buscarPosibles()** agrega en una lista la tarea si esta no existe ya en esa lista.

Anteriormente, la función **lineaEnsamblaje()** se encarga de realizar un ciclo basado en la cantidad de tareas totales, esto con el fin de utilizar todas las estaciones establecidas, luego, de la lista de las tareas posibles, agarra la tarea con mayor tiempo y la intenta introducir en la estación, si cabe, entonces hace la reducción de tiempo y agrega la tarea, de lo contrario sigue con la siguiente estación. Cada vez que una tarea es agregada, se llama a la función **buscarPosibles()** para que actualice la lista de posibles tareas.

Este ciclo lo repite hasta terminar todas las estaciones.

**Para crear el algoritmo Genético se aplicó la siguiente estrategia.**

**Creación de los padres:** Para crear a los padres (Línea de ensamblaje realizada aleatoriamente), se implemento la misma estrategia que para el algoritmo dinámico, sin embargo, la elección de las tareas fue realizadas aleatoriamente. Cabe rescatar que la cantidad de estaciones no son establecidas por la formula, sino que son un 70% de la cantidad de tareas, entonces si existen 10 tareas, existirán 7 estaciones, esto ya que, entre más estaciones, mejor serán los cruces entre los padres. Cuando la cantidad de estaciones sobrantes sea igual a la cantidad de tareas sobrantes, entonces se distribuye una tarea por estación para evitar que las estaciones queden vacías.

**Cruces:** Una vez tenida cierta cantidad de padres (en este caso 5), se establecía crear la descendencia de estos, antes de esto, se procedió a llamar la función **fitnessPadre()** la cual tiene el propósito de retornar los mejor 3 padres de una lista, teniendo estos tres padre, se inicia con los cruces de datos, para esto se llamara la función **crearHijos()**, la cual esta encargada de brindarle la información de cuales padres se van a cruzar a la función  **crearHijo(),** por lo generar para el hijo 1 se cruzan los padres 1 y 2, para el hijo 2 los padres 2 y 3 y para el hijo 3 los padres 1 y 3.

Llamada la función **crearHijo()**, esta se encargar de cruzar a los padres por estaciones, así que realiza un ciclo por la cantidad de estaciones que existen, una vez acá, se verifica cual de las dos estaciones es mejor, si la del padre uno o la del padre dos, esto se deriva de cual presenta menos tiempo ocioso. Sea cual sea el padre con la mejor estación, entonces en las estaciones del hijo se procederá a insertar esa misma estación, eliminando las tareas que ya existen en esa estación (esto para no insertar tareas repetidas).

Esto se hará con todas las estaciones de ambos padres hasta completar todo el material genético del hijo.

**Mutación:** En el algoritmo existen dos procesos de mutación, el primero se representa cuando a la hora se insertar las estaciones de un padre en un hijo, existen en esa estación tareas que ya han sido insertadas, en ese momento se procederá a eliminar esa tarea y por ende se alterará el material genético.

La segunda y la más importante ocurre cuando se procede a mutar el mejor hijo, para saber cual es este hijo se procede a llamar la función **fitnessHijo()** encargada de devolver el mejor hijo de los tres basándose en cual de estos tiene el menor tiempo ocioso.

Para entrar en contexto de esta mutación, es necesario recordar que la cantidad de estaciones de un hijo y de padre equivale al 70% del total de las tareas, lo cual si quedara así seria un poco ineficiente a la hora de medir el tiempo ocioso de la línea de ensamblaje esto ya que existen muchas estaciones con una sola tarea y por ende el tiempo sobrante es muy grande. Para solucionar este problema, se plantea la mutación de estaciones, para esto no se va a eliminar material genético, sino que más bien se reacomodara, para esto se realizara una función llamada **mutación()** que recibe un hijo y devuelve uno mejor (llamado prodigio), para esto el algoritmo realizara un ciclo en todas las estaciones y buscara cuales de las tareas de las estaciones superiores puede reacomodar en las estaciones inferiores, para esto se llamaran dos funciones **buscarPosiblesMutantes()** y **precedentesUsadosM()** estas funciones se encargaran de verificar si una tarea puede insertarse o no en las estaciones anteriores, para esto verifica que cada precedente de esa tarea ya este en las estaciones inferiores, si es así se puede correr, de lo contrario no.

Este proceso se realiza hasta completar todas las tareas sin dejar ninguna por fuera, las estaciones sobrantes simplemente se eliminar. Una vez hecho esto se retorna el prodigio y el algoritmo genético habrá concluido su función.

**Cuadros de los primeros cuatro ensamblajes.**

|  |  |  |
| --- | --- | --- |
| Ensamblaje 1 – 10 tareas | | |
| Tarea | Tiempo | Precedente |
| 1 | 30 | - |
| 2 | 24 | 1 |
| 3 | 15 | 2 |
| 4 | 40 | 1 |
| 5 | 25 | 4,3 |
| 6 | 30 | 3 |
| 7 | 15 | 5 |
| 8 | 10 | 4 |
| 9 | 49 | 8 |
| 10 | 38 | 6,7,9 |

**10 Tareas: Trabajo diario 14400 segundos, 150 productos**

**20 Tareas: Trabajo diario 18000 segundos, 175 productos**

|  |  |  |
| --- | --- | --- |
| Ensamblaje 2 – 20 tareas | | |
| Tarea | Tiempo | Precedente |
| 1 | 10 | - |
| 2 | 20 | 1 |
| 3 | 25 | 1 |
| 4 | 15 | 1 |
| 5 | 30 | 2 |
| 6 | 31 | 2 |
| 7 | 15 | 3 |
| 8 | 10 | 4 |
| 9 | 19 | 5 |
| 10 | 32 | 6 |
| 11 | 40 | 9,10 |
| 12 | 23 | 6,7 |
| 13 | 27 | 8 |
| 14 | 40 | 13 |
| 15 | 35 | 8 |
| 16 | 20 | 15,14 |
| 17 | 10 | 7 |
| 18 | 17 | 11,12 |
| 19 | 24 | 18,17 |
| 20 | 57 | 16,19 |

**30 Tareas: Trabajo diario 21600 segundos, 200 productos**

|  |  |  |
| --- | --- | --- |
| Ensamblaje 3 | | |
| Tarea | Tiempo | Precedente |
| 1 | 10 | - |
| 2 | 20 | 1 |
| 3 | 25 | 2 |
| 4 | 15 | 3 |
| 5 | 30 | 2 |
| 6 | 30 | 2 |
| 7 | 15 | 1 |
| 8 | 10 | 7 |
| 9 | 19 | 4 |
| 10 | 32 | 6,8 |
| 11 | 12 | 5,9 |
| 12 | 23 | 9 |
| 13 | 27 | 4 |
| 14 | 32 | 13,12 |
| 15 | 35 | 14 |
| 16 | 20 | 12 |
| 17 | 10 | 14 |
| 18 | 17 | 17,16 |
| 19 | 24 | 11 |
| 20 | 39 | 10,19 |
| 21 | 40 | 20 |
| 22 | 23 | 21 |
| 23 | 27 | 18,22 |
| 24 | 35 | 8 |
| 25 | 35 | 24 |
| 26 | 20 | 25 |
| 27 | 10 | 26 |
| 28 | 17 | 27 |
| 29 | 24 | 28 |
| 30 | 62 | 29,15,23 |

**60 Tareas: Trabajo diario 28800 segundos, 240 productos**

|  |  |  |
| --- | --- | --- |
| Ensamblaje 4 | | |
| Tarea | Tiempo | Precedente |
| 1 | 10 | - |
| 2 | 20 | 1 |
| 3 | 25 | 2 |
| 4 | 15 | 3 |
| 5 | 30 | 2 |
| 6 | 40 | 2 |
| 7 | 15 | 1 |
| 8 | 10 | 7 |
| 9 | 19 | 4 |
| 10 | 32 | 6,8 |
| 11 | 40 | 5,9 |
| 12 | 23 | 9 |
| 13 | 27 | 4 |
| 14 | 40 | 13,12 |
| 15 | 35 | 14 |
| 16 | 20 | 12 |
| 17 | 10 | 14 |
| 18 | 17 | 17,16 |
| 19 | 24 | 11 |
| 20 | 50 | 10,19 |
| 21 | 40 | 20 |
| 22 | 23 | 21 |
| 23 | 27 | 18,22 |
| 24 | 40 | 8 |
| 25 | 35 | 24 |
| 26 | 20 | 25 |
| 27 | 10 | 26 |
| 28 | 17 | 27 |
| 29 | 24 | 28 |
| 30 | 50 | 29,15,23 |
| 31 | 30 | 30 |
| 32 | 20 | 31 |
| 33 | 15 | 32 |
| 34 | 60 | 31 |
| 35 | 70 | 33,34 |
| 36 | 30 | 33 |
| 37 | 20 | 35 |
| 38 | 10 | 34 |
| 39 | 5 | 38 |
| 40 | 43 | 36,37,39 |
| 41 | 10 | 25 |
| 42 | 20 | 41 |
| 43 | 25 | 41 |
| 44 | 15 | 41 |
| 45 | 30 | 42 |
| 46 | 40 | 42 |
| 47 | 15 | 43 |
| 48 | 10 | 44 |
| 49 | 19 | 45 |
| 50 | 32 | 46 |
| 51 | 40 | 49,50 |
| 52 | 23 | 46,47 |
| 53 | 27 | 48 |
| 54 | 40 | 53 |
| 55 | 35 | 48 |
| 56 | 20 | 55,54 |
| 57 | 10 | 47 |
| 58 | 17 | 51,52 |
| 59 | 24 | 57,58 |
| 60 | 75 | 56,59,40 |

**Análisis de Resultados:**

|  |  |  |
| --- | --- | --- |
| **Algoritmo** | **Estado** | **Comentario** |
| Dinámico | Finalizado y completo | El algoritmo funciona correctamente, crea las estaciones a la perfección y presenta una eficiencia casi de un 100% |
| Genético | Finalizado y completo | El algoritmo genético funciona correctamente, sin embargo, este no siempre presenta una eficiencia excelente, esto ya que, para alcanzar este punto, se debió crear la mayor cantidad de padres posibles, para que de esta manera a la hora de ejecutar la función Fitness los padres restantes fueran los mas fuertes y de esta manera sus hijos también. Sin embargo, por motivos de tiempo de ejecución se crean únicamente 5 padres por cada línea de ensamblaje. |

**Algoritmo #1: Dinámico**

**Medición Empírica**

| Operaciones | Cantidad de datos de entrada por defecto | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 60 | 120 | 1200 |
| Asignaciones | 454 | 1540 | 3234 | 12104 | 96863 | 31541789 |
| Comparaciones | 707 | 2661 | 5816 | 22696 | 139627 | 35850716 |

**Factor de Crecimiento**

|  |  |  |  |
| --- | --- | --- | --- |
| Talla | Factor talla | Factor Asig | Factor Comp |
| 20 a10 | 2 | 3,392 | 3,763 |
| 30 a 20 | 1,5 | 2,1 | 2,185 |
| 60 a 30 | 2 | 3,742 | 3,902 |
| 120 a 60 | 2 | 8,002 | 6,152 |
| 1200 a 120 | 10 | 325,6329 | 256,7906 |

|  |  |  |
| --- | --- | --- |
| Clasificación del comportamiento de las **asignaciones** | O(n3) |  |
| Clasificación del comportamiento de las **comparaciones** | O(n2) |

|  |  |  |  |
| --- | --- | --- | --- |
| Clasificación según su entrada de los datos use la notación  Theta, O Grande, y Omega según corresponda | | | |
| **Entrada de los datos** | **10-20 y 30-60** | **30 - 20** | **120 ->** |
| Clasificación | Ω(2N) | ϴ (n2) | O(n3) |

**Medición Analítica**

|  |  |
| --- | --- |
| Código fuente | Líneas ejecutadas |
| public void lineaEnsamblaje() {  buscarPosibles();  for (int e = 0; e < ensamblaje.getEstaciones().size(); e++) {  Estacion estacionT = ensamblaje.getEstaciones().get(e);  int tiempo = ensamblaje.getTiempoCiclo();  int i = 0;  while (i < listaPosiblesTareas.size()) {  if (tiempo >= listaPosiblesTareas.get(i).getTiempo()) {  estacionT.setTarea(listaPosiblesTareas.get(i).getNombre());  tiempo -= listaPosiblesTareas.get(i).getTiempo();  listaPosiblesTareas.get(i).setIsEstacion(true);  listaPosiblesTareas.get(i).setEstacion(estacionT.getNombre());  listaPosiblesTareas.remove(i);  buscarPosibles();  i = 0;  } else {  i++;  }  }  estacionT.setTiempoSobrante(tiempo);  }  }  public boolean **precedentesUsados**(Tarea tarea) {  for (int i = 0; i < tarea.getPrecedentes().size(); i++) {  if (!ensamblaje.getTarea(tarea.getPrecedentes().get(i)).isIsEstacion()) {  return false;  }  }  return true;  }    public void buscarPosibles() {  for (int i = 0; i < listaTareas.size(); i++) {  if ((precedentesUsados(listaTareas.get(i)))  && (!listaTareas.get(i).isIsEstacion())  && (!listaPosiblesTareas.contains(listaTareas.get(i)))) {  asig++;  listaPosiblesTareas.add(listaTareas.get(i));  }  }  listaPosiblesTareas.sort(Comparator.comparing(Tarea::getTiempo));  Collections.reverse(listaPosiblesTareas);  } | 4n2 + 8n + 4  2n + 2  N  N  N  N \* (n+2)  N\*n  N\*n  N\*n  N\*n  N\*n  N\*n  N\*n  N\*n \* [4n2 + 8n + 4] = 4n4+8n3+4n2  N  N  17n+14n2+8n3+4n4 + 6  2n + 2  N  N  4n + 2  2n+2  N \* [ 4n + 2] = 4n2 + 2n  2N  N  1  1  4n2 + 8n + 4 |
| Total (la suma de todos los pasos) | 4n4 +8n3+ 14n2 +17n +6 |
| Clasificación en notación O Grande | O(n4) |

**Algoritmo #2: Genético**

**Medición Empírica**

| Operaciones | Cantidad de datos de entrada por defecto | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 60 | 120 | 1200 |
| Asignaciones | 3460 | 9622 | 18540 | 114226 | 377692 | 113386023 |
| Comparaciones | 4489 | 14986 | 31279 | 216542 | 750716 | 228154108 |

**Factor de Crecimiento**

|  |  |  |  |
| --- | --- | --- | --- |
| Talla | Factor talla | Factor Asig | Factor Comp |
| 20 a10 | 2 | 2,7809 | 3,3383 |
| 30 a 20 | 1,5 | 1,9268 | 2,0872 |
| 60 a 30 | 2 | 6,161 | 0,069 |
| 120 a 60 | 2 | 3,3065 | 3,47 |
| 1200 a 120 | 10 | 300,207 | 303,915 |

|  |  |  |
| --- | --- | --- |
| Clasificación del comportamiento de las **asignaciones** | O(n2) |  |
| Clasificación del comportamiento de las **comparaciones** | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| Clasificación según su entrada de los datos use la notación  Theta, O Grande, y Omega según corresponda | | | |
| **Entrada de los datos** | **10-20 y 20-30** | **120 - 60** | **1200 ->** |
| Clasificación | Ω(N) | ϴ (n2) | O(n3) |

**Medición Analítica**

|  |  |
| --- | --- |
| Código fuente | líneas ejecutadas |
| public void lineaEnsamblaje() {  buscarPosibles();  int estacionesS = listaEstaciones.size();  for (Estacion estacion : listaEstaciones) {  int ciclo = ensamblaje.getTiempoCiclo();  if (estacionesS == listaTareas.size() + 1) {  int x = (int) ((Math.random()) \* (listaPosiblesTareas.size()));  estacion.getTareas().add(listaPosiblesTareas.get(x).getNombre());  estacion.setTiempoSobrante(ciclo - listaPosiblesTareas.get(x).getTiempo());  listaPosiblesTareas.get(x).setIsEstacion(true);  listaTareas.remove(listaPosiblesTareas.get(x));  listaPosiblesTareas.remove(x);  buscarPosibles();  } else {  int i = 0;  while (i < listaPosiblesTareas.size()) {  int x = (int) ((Math.random()) \* (listaPosiblesTareas.size()));  if (ciclo >= listaPosiblesTareas.get(x).getTiempo()) {  if (estacionesS == listaTareas.size() + 1) {  break;  }  estacion.setTarea(listaPosiblesTareas.get(x).getNombre());  ciclo -= listaPosiblesTareas.get(x).getTiempo();  listaPosiblesTareas.get(x).setIsEstacion(true);  listaPosiblesTareas.get(x).setEstacion(estacion.getNombre());  listaTareas.remove(listaPosiblesTareas.get(x));  listaPosiblesTareas.remove(x);  buscarPosibles();  i = 0;  } else {  asig++;  }  }  estacion.setTiempoSobrante(ciclo);  }  estacionesS--;  }  }  public boolean **precedentesUsados**(Tarea tarea) {  for (int i = 0; i < tarea.getPrecedentes().size(); i++) {  if (!ensamblaje.getTarea(tarea.getPrecedentes().get(i)).isIsEstacion()) {  return false;  }  }  return true;  }  public **void buscarPosibles**() {  for (int i = 0; i < listaTareas.size(); i++) {  if ((precedentesUsados(listaTareas.get(i)))  && (!listaTareas.get(i).isIsEstacion())  && (!listaPosiblesTareas.contains(listaTareas.get(i)))) {  listaPosiblesTareas.add(listaTareas.get(i));  }  }  }  public void **crearEstaciones**(Ensamblaje ensamblaje) {  int estac = (int) (ensamblaje.getLineaEnsamblaje().size() \* 0.7);  for (int i = 0; i < estac; i++) {  listaEstaciones.add(new Estacion("Estacion" + String.valueOf(i + 1), ensamblaje.getTiempoCiclo()));  }  }  public void **crearPadres**() {  limpiarListas();  for (int i = 0; i < 5; i++) {  limpiarListas();  lineaEnsamblaje();  Padre padre = new Padre(listaEstaciones, "Padre " + (i + 1));  listaPadres.add(padre);  }  crearHijos();  }  public void **limpiarListas**() {  listaTareas.clear();  for (Tarea x : ensamblaje.getLineaEnsamblaje()) {  x.setIsEstacion(false);  }  for (Tarea x : ensamblaje.getLineaEnsamblaje()) {  listaTareas.add(x);  }  listaEstaciones.clear();  listaPosiblesTareas.clear();  crearEstaciones(ensamblaje);  }  public void **crearHijos**() {  ArrayList<Padre> padres = fitnessPadre(this.listaPadres);  this.listaPadres = padres;  Hijo hijo1 = crearHijo(padres.get(0), padres.get(1));  Hijo hijo2 = crearHijo(padres.get(1), padres.get(2));  Hijo hijo3 = crearHijo(padres.get(0), padres.get(2));  hijo1.setNombre("Hijo1");  hijo2.setNombre("Hijo2");  hijo3.setNombre("Hijo3");  this.listaHijos.add(hijo1);  this.listaHijos.add(hijo2);  this.listaHijos.add(hijo3);  Hijo mejorHijo = fitnessHijo(listaHijos);  mutacion(mejorHijo);  }  public Hijo **crearHijo**(Padre padre1n, Padre padre2n) {  ArrayList<Estacion> esta1 = new ArrayList<>();  ArrayList<Estacion> esta2 = new ArrayList<>();  for (Estacion estacion : padre1n.getPadre()) {  ArrayList<String> tareas = new ArrayList<>();  tareas.addAll(estacion.getTareas());  Estacion nueva = new Estacion(estacion.getNombre(), estacion.getTiempoSobrante());  nueva.setTareas(tareas);  esta1.add(nueva);  }  for (Estacion estacion : padre2n.getPadre()) {  ArrayList<String> tareas = new ArrayList<>();  tareas.addAll(estacion.getTareas());  Estacion nueva = new Estacion(estacion.getNombre(), estacion.getTiempoSobrante());  nueva.setTareas(tareas);  esta2.add(nueva);  }  Padre padre1 = new Padre(esta1, padre1n.getNombre());  Padre padre2 = new Padre(esta2, padre2n.getNombre());  ArrayList<String> noUsadosTemp = new ArrayList<>();  ArrayList<String> usados = new ArrayList<>();  ArrayList<Estacion> estacionesH = new ArrayList<>();  for (Estacion estacion : padre1.getPadre()) {  Estacion estacionN = new Estacion(estacion.getNombre(), ensamblaje.getTiempoCiclo());  estacionesH.add(estacionN);  }  Hijo hijo = new Hijo("", padre1.getNombre(), padre2.getNombre(), estacionesH);  for (int i = 0; i < hijo.getHijo().size(); i++) {  for (String tarea : noUsadosTemp) {  if  (hijo.getHijo().get(i).getTiempoSobrante() >= ensamblaje.getTarea(tarea).getTiempo()) {  hijo.getHijo().get(i).getTareas().add(tarea);  hijo.getHijo().get(i).setTiempoSobrante(hijo.getHijo().get(i).getTiempoSobrante() - ensamblaje.getTarea(tarea).getTiempo());  usados.add(tarea);  }  }  for (String tarea : usados) {  if (noUsadosTemp.contains(tarea)) {  noUsadosTemp.remove(tarea);  }  }  for (int t = 0; t < padre1.getPadre().get(i).getTareas().size(); t++) {  if (usados.contains(padre1.getPadre().get(i).getTareas().get(t))) {  int tiempoSobrante = padre1.getPadre().get(i).getTiempoSobrante() + ensamblaje.getTarea(padre1.getPadre().get(i).getTareas().get(t)).getTiempo();  padre1.getPadre().get(i).setTiempoSobrante(tiempoSobrante);  padre1.getPadre().get(i).getTareas().remove(padre1.getPadre().get(i).getTareas().get(t));  }  }  for (int t = 0; t < padre2.getPadre().get(i).getTareas().size(); t++) {  if (usados.contains(padre2.getPadre().get(i).getTareas().get(t))) {  int tiempoSobrante = padre2.getPadre().get(i).getTiempoSobrante() + ensamblaje.getTarea(padre2.getPadre().get(i).getTareas().get(t)).getTiempo();  padre2.getPadre().get(i).setTiempoSobrante(tiempoSobrante);  padre2.getPadre().get(i).getTareas().remove(padre2.getPadre().get(i).getTareas().get(t));  }  }  if (padre1.getPadre().get(i).getTiempoSobrante() >= padre2.getPadre().get(i).getTiempoSobrante()) {  for (String tarea : padre1.getPadre().get(i).getTareas()) {  if (!usados.contains(ensamblaje.getTarea(tarea))) {  hijo.getHijo().get(i).getTareas().add(tarea);  hijo.getHijo().get(i).setTiempoSobrante(hijo.getHijo().get(i).getTiempoSobrante() - ensamblaje.getTarea(tarea).getTiempo());  usados.add(tarea);  }  }  for (String tarea : padre2.getPadre().get(i).getTareas()) {  if (!usados.contains(ensamblaje.getTarea(tarea))) {  noUsadosTemp.add(tarea);  }  }  } else {  for (String tarea : padre2.getPadre().get(i).getTareas()) {  if (!usados.contains(ensamblaje.getTarea(tarea))) {  hijo.getHijo().get(i).getTareas().add(tarea);  hijo.getHijo().get(i).setTiempoSobrante(hijo.getHijo().get(i).getTiempoSobrante() - ensamblaje.getTarea(tarea).getTiempo());  usados.add(tarea);  }  }  for (String tarea : padre1.getPadre().get(i).getTareas()) {  if (!usados.contains(ensamblaje.getTarea(tarea))) {    noUsadosTemp.add(tarea);  }  }  }  }  return hijo;  }  public void **mutacion**(Hijo hijo) {  listaTareas.clear();  listaPosiblesTareas.clear();  for (Tarea x : ensamblaje.getLineaEnsamblaje()) {  listaTareas.add(x);  }  ArrayList<Estacion> estacionesH = new ArrayList<>();  for (Estacion estacion : hijo.getHijo()) {  Estacion estacionN = new Estacion(estacion.getNombre(), ensamblaje.getTiempoCiclo());  estacionesH.add(estacionN);  }  buscarPosiblesMutantes();  Hijo hijoMutado = new Hijo(hijo.getNombre(), hijo.getNombreP1(), hijo.getNombreP2(), estacionesH);  ArrayList<String> usados = new ArrayList<>();  for (Estacion estacionM : hijoMutado.getHijo()) {  for (Estacion estacionN : hijo.getHijo()) {  for (String tarea : estacionN.getTareas()) {  if ((!usados.contains(tarea))  && (estacionM.getTiempoSobrante() >= ensamblaje.getTarea(tarea).getTiempo())  && (listaPosiblesTareas.contains(ensamblaje.getTarea(tarea)))) {  usados.add(tarea);  estacionM.getTareas().add(tarea);  estacionM.setTiempoSobrante(estacionM.getTiempoSobrante() - ensamblaje.getTarea(tarea).getTiempo());  ensamblaje.getTarea(tarea).setIsMutado(true);  buscarPosiblesMutantes();  }  }  }  }  this.prodigio = hijoMutado;  }  public ArrayList<Padre**> fitnessPadre**(ArrayList<Padre> padres) {  for (Padre padre : padres) {  padre.sobranteT();  }  padres.sort(Comparator.comparing(Padre::getSobrante));  ArrayList<Padre> nuevosP = new ArrayList<>();  nuevosP.add(padres.get(0));  nuevosP.add(padres.get(1));  nuevosP.add(padres.get(2));  return nuevosP;  }  public Hijo **fitnessHijo**(ArrayList<Hijo> hijos) {  for (Hijo hijo : hijos) {  hijo.sobranteT();  }  hijos.sort(Comparator.comparing(Hijo::getSobrante));  return hijos.get(0);  }  public boolean **precedentesUsadosM**(Tarea tarea) {  for (int i = 0; i < tarea.getPrecedentes().size(); i++) {  if (!ensamblaje.getTarea(tarea.getPrecedentes().get(i)).isIsMutado()) {  return false;  }  }  return true;  }  public void **buscarPosiblesMutantes**() {  for (int i = 0; i < listaTareas.size(); i++) {  if ((precedentesUsadosM(listaTareas.get(i)))  && (!listaTareas.get(i).isIsMutado())  && (!listaPosiblesTareas.contains(listaTareas.get(i)))) {  listaPosiblesTareas.add(listaTareas.get(i));  }  }  } | 2n2+7n+2  1  2n+2  n  n  n  n  n  n  n  n  n  2n3+7n2+2n  n  n2 + 2n  n \* n  n\*n  n\*n  n\*n  n\*n  n\*n  n\*n  n\*n  n\*n  2n4+7n3+2n2  n \* n  n\*n  n  n  2n4+2n3+23n2  +25n+5  -----------------------------  2n+n  n  2n+2  2n+2  n (2n+2)  3n  2n2+7n+2  ----------------------------  1  2n+2  3n  5n+3  11n+10  2n+2  11n2+10n  2n5+2n4+23n3  +25n2+5n)  1  1  16n4+24n3  +112n2  +135n+86  2n5+18n4+47n3  148n2+163n+100  1  2n + 2  n  2n+2  n  1  1  5n+3  **11n+10**  3n+7  1  4n3+36n2+37n+16  4n3+36n2+37n+16  4n3+36n2+37n+16  1  1  1  1  1  1  3n+3  4n4+10n3+4n2  +15n+21  16n4+24n3  +112n2  +135n+86  1  1  2n+2  n  n  n  n  n  2n+2  n  n  n  n  n  1  1  1  1  1  2n+2  n  n  1  2n+2  2n2+2n)  n2  n2  n2  n2  2n2+2n  N2  N2  2n2+2n  n2  n2  n2  n2  2n3+2n  n2  n2  n2  n2  n  n  2n2+2n  n2  n2  n2  n2  2n2+2n  n2  n2  2n2+2n  n2  n2  n2  n2  2n3+2n)  n2  n2  4n3+36n2+37n+16  1  1  2n+2  n  1  2n+2  n  n  4n+2  1  1  2n+2  (2n2+2n)  (2n3+2n2)  N3  N3  N3  N3  N3  N3  (4n4+2n3)  1  4n4+10n3+4n2  +15n+14  2n+2  n  1  1  1  1  1  3n+7  2n+2  n  1  3n+3  2n+2  n  3n+2  2n+2  n  n  4n+2 |
| Total (la suma de todos los pasos) | 2n5+18n4+47n3  148n2+163n+100 |
| Clasificación en notación O Grande |  |

**Graficas**

**Conclusiones:**

Gracias a las mediciones realizadas a los algoritmos creados (Medición empírica y medición analítica), hemos podido enterarnos del consumo de datos de cado método, así como cual fue más eficiente y a que ritmo crecían los cálculos que generaba con respecto a la cantidad de datos introducidos.

Realizando estas comparaciones se pudo concluir que el algoritmo dinámico era el que crecía de menor manera con respecto a la cantidad de datos procesados, por otro lado, el algoritmo genético aumentaba un poco más.

Además, el algoritmo dinámico fue de medición O(n4) y el algoritmo genético O(n5), es por esto que se deduce que el algoritmo dinámico es más eficiente en comparación al algoritmo genético.

**Recomendaciones**

Con la elaboración de este proyecto y la resolución de los problemas planteados de ha permitido adquirir una serie de conocimientos fundamentales para el desarrollo y análisis de algoritmos.

Gracias a estos nuevos aprendizajes podemos realizar unas series de recomendaciones las cuales pueden ser de mucha utilidad a la hora de realizar un algoritmo eficiente.

Entre ellos podemos resaltar la importancia de una buena medición de un algoritmo, gracias a esto podremos supervisar los niveles de crecimiento de estos y concluir si es eficiente y óptimo para cumplir su función.

Siempre es primordial crear algoritmos eficientes capaces de consumir los mínimos recursos del computador, esto puede agilizar el funcionamiento del algoritmo y evitar la saturación de datos en el procesador.

Nunca está de más investigar, es por esto que siempre es bueno consultar si existe algún algoritmo eficiente ya creado que podamos adaptar a nuestro problema, esto nos puede ahorrar tiempo el cual podemos invertir mejorando otras áreas del proyecto.

**Literatura citada**

Design and Analysis of Algorithms, Massachusetts Institute of Technology

Profs. Erik Demaine, Srini Devadas, and Nancy Lynch

Algoritmos Genéticos. (s.f.). Recuperado 06 noviembre, 2019, de <http://www.nce.ufrj.br/GINAPE/VIDA/alggenet.htm>

José Leandro González González, J. L. G. G. (2013, 11 junio). Propuesta de algoritmo de clasificación genética | González González | Revista Cubana de Ingeniería. Recuperado 10 noviembre, 2019, de <http://rci.cujae.edu.cu/index.php/rci/article/view/158>