Facultad de Ingeniería de la UBA

75.29 Teoría de Algoritmos
Trabajo Práctico N°3

Cuatrimestre: Segundo de 2014

Fecha de entrega: 5 de Diciembre de 2014

Alumnos:

Apellido y Nombre	Padrón	Email
Parnisari, María Inés	92235	maineparnisari@gmail.com
Buffevant, César	76700	buffevant@gmail.com

Enunciado

PLANIFICACIÓN DE ÓRDENES DE TRABAJO
Dado un conjunto de n trabajos que hay que realizar en una máquina determinada, tales que cada trabajo a_i requiere t_i tiempo de máquina para ser completado y produce un beneficio b_i y tiene asociado un vencimiento v_i
Y considerando que un trabajo no puede interrumpirse una vez que se inició, y la máquina solo puede ejecutar un único trabajo a la vez y el beneficio b_i se percibe solo si el trabajo a_i se completa antes de su vencimiento v_i (caso contrario el beneficio es 0).
Dados los siguientes escenarios:

a) Todos los tiempos t_i son enteros entre 1 y n. Los vencimientos v_i también son enteros.

- Escribir un programa que encuentre la planificación buscada en tiempo polinomial (programación dinámica).
- Calcular el orden de la solución encontrada.

Los datos vienen dado en un archivo de texto donde cada línea contiene la tupla (valores separados por coma): t_i, b_i, v_i por cada trabajo.
La salida es una secuencia de enteros, ordenados según el orden en que deben ejecutarse cada uno de los trabajos.

b) Los tiempos t_i y los vencimientos v_i son reales (arbitrarios)
¿Existe una planificación tal que todos los trabajos se completan y el beneficio total es de al menos K?
Demostrar que responder a la pregunta enunciada es NP-completo:

- Calcular el orden del algoritmo verificador (que dada una planificación comprueba si es

solución o no)

- Reducir un problema NP-completo al problema bajo estudio.

Programación Dinámica

El algoritmo se basa en la suposición de que las tareas vienen ordenadas por vencimiento creciente. En la implementación ésto no es requerido ya que la lista de tareas se ordena antes de generar el plan.

La idea básica del algoritmo es chequear que se pueda mejorar la planificación siempre que haya tiempo antes del vencimiento de la tarea. La idea de que haya tiempo disponible es capturada por la fórmula:

$$t = min(tiempo, vencimiento\ tarea) - duración\ tarea$$

Si t es negativo entonces no hay tiempo para programar la tarea. Si t es positivo entonces se puede mejorar la planificación programando la tarea en el tiempo t.
La ecuación de recurrencia es:

$$OPT(i, t) = OPT(i-1, t) \text{ cuando } t_i > min(t, v_i)$$

$$OPT(i, t) = max(OPT(i-1, t), b_i + OPT(i-1, min(t, v_i) - t_i)) \text{ cuando } t_i \leq min(t, v_i)$$

donde,

- t_i es la duración de la tarea i
- v_i es el vencimiento de la tarea i
- b_i es el beneficio de la tarea i
- OPT(i, t) es la solución óptima hasta el tiempo t teniendo i tareas.

La clase Task representa una tarea, mientras que la clase Planner tiene el método que implementa el algoritmo.

Casos de Prueba

Se incluyen test unitarios para los siguientes casos:

- Todas las tareas tienen vencimientos escalonados por lo que todas son programadas en orden.
- Todas las tareas tiene vencimientos escalonados pero en el orden inverso (la tarea N vence primero y la tarea 1 vence última), por lo que todas las tareas son programadas al revés.
- Todas las tareas tienen un vencimiento mayor que la suma de las duraciones por que lo todas las tareas se programan.
- Hay una tarea de mayor beneficio que se solapa con dos tareas, por lo que las dos tareas no se programan al elegirse la tarea de mayor beneficio.
- Hay una tarea solapada de menor beneficio la cual es ignorada por otras dos de mayor beneficio.

Calculo de Ordenes

Lectura del archivo

La lectura del archivo es O(n) ya que se leen una vez cada línea del mismo. En cada línea hay una tarea.

Planificación

El ordenamiento de la lista de tareas se incluyó como primer parte del algoritmo, la primer fase del algoritmo es O(NlogN) donde N es la cantidad de tareas.
Definimos W como el máximo vencimiento.
La implementación de programación dinámica itera por todas la tareas (filas en la matriz de resultados) y por todas las columnas (los tiempo entre 1 y W) actualizando la matriz de resultados. Todas las operaciones sobre la matriz son O(1). Luego de generar la matriz de resultados hay que generar la planificación recorriendo dicha matriz desde la tarea n en el tiempo W hacía atrás. Al recorrer toda la lista de tareas de la planificación es a lo sumo O(N) , por lo que el orden total del algoritmo es O(NLogN + NW + N) = O(NLogN + N(W + 1)).

Demostración NP-Completo

Llamemos X a nuestro problema bajo estudio.
Un algoritmo verificador de X debería chequear que la planificación es válida, todas las tareas se ejecutan y el beneficio total es de al menos K.

Supongamos que tengo un diccionario D con todas las tareas, indexadas por un entero. Una solución a X es una planificación de la forma (i, Ti), donde i es el indice identificando la tarea y Ti es el tiempo en el que comienza la tarea.

```
Input:
	K // goal
	plan // planificación
	D // diccionario con todas las tareas
last_end ←  0
benefit ← 0
for j = 0 to length(plan)
{
	task ← D[plan[j].i]
	check last_end  <= plan[j].T // la tarea no puede ejecutarse antes de la anterior.
	if (task.v >= plan[j].T + task.t)
	{
		benefit ← benefit + t.b
	}
	last_end ← plan[j].T + task.t  // actualizo last_end
	remove(plan[i].i, D) // remuevo la tarea del diccionario.
```

```
}
check length(D) == 0 // verifico que todas las tareas se hayan ejecutado
check benefit >= K // verifico que el beneficio sea de al menos K.
```

Considerando que remover un item del diccionario es O(n), y estamos iterando por todos los items de la planificación, el algoritmo verificador es O(n2). Por lo tanto existe un verificado de orden polinomial para X. Ergo, X es NP.

Ahora para completar la verificación hay que reducir un problema NP-Completo a X. El problema ha utilizar es Subset Sum: Dado un conjunto S de números, existe un subconjunto de S tal que la suma de sus elementos sea igual a W:

$$S = \{s_i,\ 0 \leq i \leq N\}\ ¿\exists S_0 \subseteq S : \sum_{s_j \in S_0} s_j = W\ ?$$

¿Como construimos un problema solucionable por X, que resuelva el problema Subset Sum? X tiene tres parámetros a definir: las tareas, K (beneficio mínimo) y n (tiempo total de completitud de las tareas). Definimos estos parámetros de la siguiente manera:

- tareas: (s_i, W, s_i)
- K = W
- n = $\sum_{s_i \in S} s_i = M$

Todas las tareas que se terminen antes de W van a sumar s_i al beneficio total, y todas las que terminen después van a contribuir con 0. W parte la planificación en dos, las tareas que terminan antes, y las tareas que termina después. Llamemos P_0 al conjunto de tareas que terminan antes de W, y P_1 a las que terminan después. La sumatoria de tiempos de las tareas en P_0 es

$$\sum_{P_0} s_i \leq W \quad (1)$$

Y el beneficio total es

$$\sum_{P_0} s_i \geq K = W \quad (2)$$

Combinando (1) y (2), obtenemos que si existe una planificación donde el beneficio sea de al menos K, entonces existe un subconjunto P_0 tal que $\sum_{P_0} s_i = W$.

Al demostrar que subset sum es reducible a X y que X es NP (por existir una verificación polinomial), entonces X es NP-Completo.

Codigo Fuente

Tasks.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace TP3.Model
{
	public class Task
	{
		public Task(int id, int duration, int deadline, int profit)
		{
			this.ID = id;
			this.Duration = duration;
			this.Deadline = deadline;
			this.Profit = profit;
		}

		public int ID
		{
			get;
			private set;
		}

		public int Duration
		{
			get;
			private set;
		}

		public int Deadline
		{
			get;
			private set;
		}

		public int Profit
		{
			get;
			private set;
		}
	}
}
```

PlannerReader.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Threading.Tasks;

namespace TP3.Model
{
	public class PlannerReader
	{
		public List<Task> Read(StreamReader file)
		{
			var tasks = new List<Task>();
			var id = 1;
			while (!file.EndOfStream)
			{
				var line = file.ReadLine();
				var parts = line.Split(',');
				if (parts.Length == 3)
				{
					tasks.Add(new Task(id, Int32.Parse(parts[0]),
Int32.Parse(parts[1]), Int32.Parse(parts[2])));
					id++;
				}
			}
			return tasks;
		}
	}
}
```

Planner.cs

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace TP3.Model
{
	public class Planner
	{
		/// <summary>
		/// Este método recorre la matriz de resultados, armando el plan.
		/// </summary>
		/// <remarks>
		/// El orden es O(N) Se puede ver que la recursión siempre le resta 1 al indice
de la tarea hasta llegar a 0.
		/// </remarks>
		/// <param name="i">Indice de la tarea.</param>
		/// <param name="t">Vencimiento.</param>
		/// <param name="plan">Lista con el plan.</param>
		/// <param name="M">Matriz de resultados.</param>
		/// <param name="tasks">lista de tareas.</param>
		private static void TraceBackPlan(int i, int t, List<Task> plan, int[,] M,
List<Task> tasks)
		{
			if (i == 0) return;
			if (M[i, t] == M[i - 1, t])
			{
				TraceBackPlan(i - 1, t, plan, M, tasks);
			}
			else
			{
				var task = tasks[i-1];
				var tt = Math.Min(t, task.Deadline) - task.Duration;
				TraceBackPlan(i - 1, tt, plan, M, tasks);
				// agrego la tarea al plan.
				plan.Add(task);
			}
		}

		/// <summary>
		/// Este método devuelve una lista ordenada de tareas con el orden de ejecución.
		/// </summary>
		/// <remarks>
		/// El orden es O(NLogN + N(W + 1)) donde N es la cantidad de tareas y W es el
vencimiento máximo.
		/// </remarks>
		/// <param name="tasks">Lista desordenada de tareas.</param>
		/// <returns>Plan de ejecucion.</returns>
		public static IEnumerable<Task> GetPlan(List<Task> tasks)
		{
```

```
                        // O(n log n)
                        var orderedTasks = tasks.OrderBy(t => t.Deadline).ToList();
                        // lista para devolver el plan.
                        var plan = new List<Task>(orderedTasks.Count);
                        var maxDeadline = orderedTasks.Last().Deadline; // O(1)
                        //matriz para mantener los resultados
                        var M = new int[orderedTasks.Count+1, maxDeadline + 1];

                        // O(N*W)
                        for (var i = 1; i <= orderedTasks.Count; i++)
                        {
                                var task = orderedTasks[i-1];
                                // O(W)
                                for (var d = 1; d <= maxDeadline; d++)
                                {
                                        var t = Math.Min(d, task.Deadline) - task.Duration;
                                        if (t < 0)
                                        {
                                                // si no tengo tiempo entre el tiempo actual y el
deadline de la tarea,
                                                // decarto la tarea - el beneficio no cambia
                                                M[i, d] = M[i-1, d];
                                        }
                                        else
                                        {
                                                // si tengo tiempo entre el tiempo actual y el
deadline de la tarea,
                                                // me quedo con el máximo entre el beneficio
anterior o el actual mas el anterior en el otro tiempo.
                                                M[i, d] = Math.Max(M[i-1, d], task.Profit + M[i-1,
t]);
                                        }
                                }
                        }
                        // genero el plan a partir de la matriz de resultados - O(N)
                        TraceBackPlan(orderedTasks.Count, maxDeadline, plan, M, orderedTasks);

                        return plan;
                }
        }
}
```

Program.cs

```
using System.Configuration;
using System.Diagnostics;
using System.IO;
using TP3.Model;

namespace TP3.ConsoleApplication
{
	class MainClass
	{
		public static void Main(string[] args)
		{
			if (args.Length != 1)
			{
				System.Console.WriteLine ("Por favor, especifique un archivo de
entrada.");
				return;
			}
			System.Console.WriteLine ("Planificacion:");
			var reader = new PlannerReader ();
			using (var file = new FileStream (args [0], FileMode.Open)) {
				var tasks = reader.Read (new StreamReader(file));
				var plan = Planner.GetPlan (tasks);
				foreach (var t in plan) {
					System.Console.Write (t.ID + " ");
				}
			}
		}
	}
}
```

Test.cs

```
using NUnit.Framework;
using System;
using System.Linq;
using System.Collections.Generic;
using TP3.Model;

namespace TP3.Test
{
	[TestFixture()]
	public class PlannerTest
	{
		[Test()]
		public void AllTasksAreScheduledWhenNoDeadline()
		{
			var tasks = new List<Task>
			{
				new Task(1, 1, 100, 10),
				new Task(2, 1, 100, 10),
				new Task(3, 1, 100, 10),
				new Task(4, 1, 100, 10),
				new Task(5, 1, 100, 10)
			};

			var plan = Planner.GetPlan(tasks);
			Assert.AreEqual(1, plan.ElementAt(0).ID);
			Assert.AreEqual(2, plan.ElementAt(1).ID);
			Assert.AreEqual(3, plan.ElementAt(2).ID);
			Assert.AreEqual(4, plan.ElementAt(3).ID);
			Assert.AreEqual(5, plan.ElementAt(4).ID);
		}

		[Test()]
		public void AllTasksAreScheduledWhenIncreasingDeadline()
		{
			var tasks = new List<Task>
			{
				new Task(1, 1, 2, 10),
				new Task(2, 1, 3, 10),
				new Task(3, 1, 4, 10),
				new Task(4, 1, 5, 10),
				new Task(5, 1, 6, 10)
			};

			var plan = Planner.GetPlan(tasks);
			Assert.AreEqual(1, plan.ElementAt(0).ID);
			Assert.AreEqual(2, plan.ElementAt(1).ID);
			Assert.AreEqual(3, plan.ElementAt(2).ID);
			Assert.AreEqual(4, plan.ElementAt(3).ID);
```

```
        Assert.AreEqual(5, plan.ElementAt(4).ID);
}

[Test()]
public void AllTasksAreReversedWhenDecreasingDeadline()
{
        var tasks = new List<Task>
        {
                new Task(1, 1, 5, 10),
                new Task(2, 1, 4, 10),
                new Task(3, 1, 3, 10),
                new Task(4, 1, 2, 10),
                new Task(5, 1, 1, 10)
        };

        var plan = Planner.GetPlan(tasks);
        Assert.AreEqual(5, plan.ElementAt(0).ID);
        Assert.AreEqual(4, plan.ElementAt(1).ID);
        Assert.AreEqual(3, plan.ElementAt(2).ID);
        Assert.AreEqual(2, plan.ElementAt(3).ID);
        Assert.AreEqual(1, plan.ElementAt(4).ID);
}

[Test()]
public void OverlappedTasksWithLowerBenefitAreIgnored()
{
        var tasks = new List<Task>
        {
                new Task(1, 1, 1, 10),
                new Task(2, 1, 2, 10),
                new Task(3, 1, 3, 10),
                new Task(4, 1, 4, 10),
                new Task(5, 2, 3, 30)
        };

        var plan = Planner.GetPlan(tasks);
        Assert.AreEqual(1, plan.ElementAt(0).ID);
        Assert.AreEqual(5, plan.ElementAt(1).ID);
        Assert.AreEqual(4, plan.ElementAt(2).ID);
}

[Test()]
public void OverlappedTasksWithHigherBenefitsAreScheduled()
{
        var tasks = new List<Task>
        {
                new Task(1, 1, 1, 10),
                new Task(2, 1, 2, 20),
                new Task(3, 1, 3, 20),
                new Task(4, 1, 4, 10),
```

```
                    new Task(5, 2, 3, 10)
                };

                var plan = Planner.GetPlan(tasks);
                Assert.AreEqual(1, plan.ElementAt(0).ID);
                Assert.AreEqual(2, plan.ElementAt(1).ID);
                Assert.AreEqual(3, plan.ElementAt(2).ID);
                Assert.AreEqual(4, plan.ElementAt(3).ID);
            }
        }
}
```