DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Diseño de Algoritmos

Aplicación de técnicas 2.0

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Barañao, Facundo	480/11	facundo_732@hotmail.com
Confalonieri, Gisela Belén	511/11	gise_5291@yahoo.com.ar
Mignanelli, Alejandro Rubén	609/11	minga_titere@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Objetivos generales

2. Plataforma de pruebas

Para toda la experimentación se utilizará un procesador Intel Atom, de 4 núcleos a 1.6 GHZ.
El software utilizado será Ubuntu 14.04, y G++ 4.8.2.

3. Problema 1: Dakkar

3.1. Descripción del problema.

Tenemos la intención de competir en una versión particular del Rally Dakkar, y para esto decidimos usar nuestros conocimientos en computación a nuestro a favor. Sabemos que en esta competencia podremos usar una BMX, una motocross y un buggy arenero. Nuestro objetivo es finalizar la competencia en el menor tiempo posible, y contamos con los siguientes datos:

- El circuito se divide en n etapas (numeradas del 1 al n) y en cada etapa sólo se puede usar un vehículo.
- Para cada etapa, se sabe cuánto se tardaría en recorrerla con cada vehículo.
- Tanto la motocross como el buggy arenero tienen un número limitado de veces que se pueden usar, a diferencia de la BMX, y estos números son conocidos.
- La complejidad del algoritmo pedido es de $\mathcal{O}(n \cdot k_m \cdot k_n)$ donde n es la cantidad de etapas, k_m es la cantidad máxima de veces que puedo usar la moto, y k_b la cantidad máxima de veces que puedo usar el buggy.
- La salida de este algoritmo deberá mostrar el tiempo total de finalización de la carrera y una sucesión de n enteros representando el vehículo utilizado en cada etapa (1 para bici, 2 para moto y 3 para buggy).

Ejemplo:
Supongamos un Rally de 3 etapas, en donde puedo usar una vez la moto y una vez el buggy, con los siguientes datos:

Etapa	BMX	Moto	Buggy
1	10	8	7
2	8	3	7
3	15	6	2

Con estos datos, la manera óptima de llevar a cabo la carrera sería la siguiente:

Etapa 1	Etapa 2	Etapa 3	Tiempo total
BMX: 10	Moto: 3	Buggy: 2	15

3.2. Desarrollo de la idea y correctitud.

LLamemos n a la cantidad de etapas del Rally, k_m y k_b a la cantidad máxima de motos y buggys respectivamente que es posible utilizar en el Rally. La idea es no tratar de resolver todas las etapas de un tirón, sino tratar primero de resolver considerando sólo la primer etapa, luego considerando las dos primeras etapas ayudándonos con la resolución de la primera etapa sola, luego considerando las primeras tres etapas ayudándonos con la resolución de las primeras dos etapas, y así hasta llegar a considerar n etapas. Para esto, la resolución que considera las primeras h etapas tendrá una tabla construida a partir de la información de la tabla correspondiente a la resolución de las primeras $h-1$ etapas. Cada una de estas tablas tiene $k_m + 1$ columnas (numeradas del 0 al k_m) y $k_b + 1$ filas (numeradas del 0 al k_b). Para hacer referencia a la tabla que representa a las primeras t etapas (con t algún número natural distinto de 0), la llamaremos la tabla t. Entonces si tomamos $i, j, h \in \mathbb{N}$ tal que $0 \leq i \leq k_b$, $0 \leq j \leq k_m$ y $1 \leq h \leq n$

la idea sería que la posición i, j de la tabla h tenga dos datos: el tiempo óptimo de las primeras h etapas usando a lo sumo i buggies y j motos, y el vehículo que se usaría en la etapa h bajo esas circunstancias.

Determinemos ahora cómo se completarían las tablas (para hacer referencia a una posición de una tabla h, diremos la posición $h_{f,c}$ donde f es el número de fila, y c el número de columna):

- Si $h = 1$
  - En la posición $h_{0,0}$ colocaremos el tiempo de la bicicleta en la etapa h, puesto que sólo puede usarse ese vehículo, e indicaremos que se usó la bici.
  - En la posición $h_{0,1}$ colocaremos el tiempo de la bicicleta o la moto (ambos en la etapa h), el que sea menor, e indicaremos cuál de ellos se utilizó.
  - De la posición $h_{0,2}$ hasta h_{0,k_m} colocaremos exactamente lo que hay en $h_{0,1}$ dado que si consideramos una sola etapa, poder usar una moto o más de una, no es diferencia.
  - En la posición $h_{1,0}$ colocaremos el tiempo de la bicicleta o el buggy (ambos de la etapa h), el que sea menor, e indicaremos cuál de ellos se utilizó.
  - De la posición $h_{2,0}$ hasta $h_{k_b,0}$ colocaremos exactamente lo que hay en $h_{1,0}$ dado que si consideramos una sola etapa, poder usar un buggy o más de uno, no es diferencia.
  - En la posición $h_{1,1}$ colocaremos el tiempo de la bicicleta, el buggy o la moto (de la etapa h), el que sea menor, e indicaremos cuál de ellos se utilizó.
  - El resto de las posiciones de esa tabla, deben tener lo que hay en $h_{1,1}$ dado que si tengo una sola etapa, poder usar una moto y un buggy, o más de alguno de ellos o de ambos, no es diferencia.
- Para todo h tal que $1 < h \leq n$
  - En la posición $h_{0,0}$ colocaremos el tiempo que de la posición $(h-1)_{0,0}$ sumada al de la bici de la etapa h, e indicaremos que se usó la bici. El tiempo óptimo de este caso es correcto, puesto que al no poder usar otros vehículos, el tiempo óptimo de haber recorrido h etapas usando solo la bicicleta, es el tiempo óptimo de haber recorrido $h-1$ etapas sin usar buggys ni motos, sumado a usar una bicicleta en la etapa h. El vehículo elegido es trivialmente correcto, puesto que no hay otra elección posible.
  - Tomando $1 \leq j \leq k_m$, la posición $h_{0,j}$ debe contener el tiempo mínimo entre:
    1. El tiempo de $(h-1)_{0,j}$ sumado al tiempo de la bici en la etapa h.
    2. El tiempo de $(h-1)_{0,(j-1)}$ sumado al tiempo de la moto en la etapa h.

    Para el primer caso indicaremos que se eligió la bici, y para el segundo caso indicaremos que se eligió la moto. Veamos que el tiempo escogido es óptimo:
    1. Si el vehículo que deberíamos tomar en la etapa h fuera la bicicleta, entonces en las primeras $h-1$ etapas pudimos haber usado hasta j motos (dado que no vamos a usar ninguna moto en esta etapa). Como $(h-1)_{0,j}$ indica el tiempo óptimo de usar hasta j motos en las primeras $h-1$ etapas, al sumarle el tiempo de utilizar la bicicleta en la etapa h obtenemos el tiempo óptimo para esta etapa.
    2. Si el vehículo que deberíamos tomar en la etapa h fuera la moto, entonces en las primeras $h-1$ etapas pudimos haber usado hasta $j-1$ motos (dado que vamos a usar una moto en esta etapa). Como $(h-1)_{0,(j-1)}$ indica el tiempo óptimo de usar hasta $j-1$ motos en las primeras $h-1$ etapas, al sumarle el tiempo de utilizar la moto en la etapa h obtenemos el tiempo óptimo para esta etapa.

    Como tomamos el mínimo entre 1) y 2) entonces, y ambos, de ser la respuesta correcta, serían óptimos, entonces la elección resulta óptima en tiempo. Por esto mismo, el vehículo elegido es el correcto. Ver que no se crean conflictos con el buggy, ni tenemos que considerar elegirlo o no, puesto que no podemos usar buggies en estos casos.
  - Tomando $1 \leq i \leq k_b$ la posición $h_{i,0}$ debe contener el tiempo mínimo entre:
    1. El tiempo de $(h-1)_{i,0}$ sumado al tiempo de la bici en la etapa h.

2. El tiempo de $(h-1)_{(i-1),0}$ sumado al tiempo del buggy en la etapa h.

Para el primer caso indicaremos que se eligió la bici, y para el segundo caso indicaremos que se eligió el buggy. Veamos que el tiempo escogido es óptimo:

1. Si el vehículo que deberíamos tomar en la etapa h fuera la bicicleta, entonces en las primeras $h-1$ etapas pudimos haber usado hasta i buggies (dado que no vamos a usar ningún buggy en esta etapa). Como $(h-1)_{i,0}$ indica el tiempo óptimo de usar hasta i buggies en las primeras $h-1$ etapas, al sumarle el tiempo de utilizar la bicicleta en la etapa h obtenemos el tiempo óptimo para esta etapa.
2. Si el vehículo que deberíamos tomar en la etapa h fuera el buggy, entonces en las primeras $h-1$ etapas pudimos haber usado hasta $i-1$ buggies (dado que vamos a usar un buggy en esta etapa). Como $(h-1)_{(i-1),0}$ indica el tiempo óptimo de usar hasta $i-1$ buggies en las primeras $h-1$ etapas, al sumarle el tiempo de utilizar un buggy en la etapa h obtenemos el tiempo óptimo para esta etapa.

Como tomamos el mínimo entre 1) y 2) entonces, y ambos, de ser la respuesta correcta, serían óptimos, entonces la elección resulta óptima en tiempo. Por esto mismo, el vehículo elegido es el correcto. Ver que no se crean conflictos con la moto, ni tenemos que considerar elegirla o no, puesto que no podemos usar motos en estos casos.

- Tomando $1 \leq i \leq k_b$ y $1 \leq j \leq k_m$ la posición $h_{i,j}$ debe contener el tiempo mínimo entre:
  1. El tiempo de $(h-1)_{i,j}$ sumado al tiempo de la bici en la etapa h.
  2. El tiempo de $(h-1)_{i,(j-1)}$ sumado al tiempo de la moto en la etapa h.
  3. El tiempo de $(h-1)_{(i-1),j}$ sumado al tiempo del buggy en la etapa h.

  Para el primer caso indicaremos que se eligió la bici, para el segundo caso indicaremos que se eligió la moto y para el tercer caso indicaremos que se eligió el buggy. Veamos que el tiempo escogido es óptimo:

  1. Si el vehículo que deberíamos tomar en la etapa h fuera la bicicleta, entonces en las primeras $h-1$ etapas pudimos haber usado hasta i buggies y j motos (dado que no vamos a usar ningún buggy ni moto en esta etapa). Como $(h-1)_{i,j}$ indica el tiempo óptimo de usar hasta i buggies y hasta j motos en las primeras $h-1$ etapas, al sumarle el tiempo de utilizar la bicicleta en la etapa h obtenemos el tiempo óptimo para esta etapa.
  2. Si el vehículo que deberíamos tomar en la etapa h fuera la moto, entonces en las primeras $h-1$ etapas pudimos haber usado hasta $j-1$ motos y hasta i buggies (dado que vamos a usar una moto en esta etapa). Como $(h-1)_{i,(j-1)}$ indica el tiempo óptimo de usar hasta i buggies y hasta $j-1$ motos en las primeras $h-1$ etapas, al sumarle el tiempo de utilizar la moto en la etapa h obtenemos el tiempo óptimo para esta etapa.
  3. Si el vehículo que deberíamos tomar en la etapa h fuera el buggy, entonces en las primeras $h-1$ etapas pudimos haber usado hasta $i-1$ buggies y hasta j motos (dado que vamos a usar un buggy en esta etapa). Como $(h-1)_{(i-1),j}$ indica el tiempo óptimo de usar hasta $i-1$ buggies y hasta j motos en las primeras $h-1$ etapas, al sumarle el tiempo de utilizar un buggy en la etapa h obtenemos el tiempo óptimo para esta etapa.

  Como tomamos el mínimo entre 1), 2) y 3), y todos, de ser la respuesta correcta, serían óptimos, entonces la elección resulta óptima en tiempo. Por esto mismo, el vehículo elegido es el correcto.

Entonces, una vez completas las tablas, si queremos saber cuál es el menor tiempo en el que podemos completar el Rally, tan sólo debemos ver el tiempo de n_{k_b,k_m}, y si queremos saber qué vehículos utilizamos en cada etapa, debemos ver qué vehículo se usó en n_{k_b,k_m} y ese vehículo será el que se usó en la etapa n. Luego, de manera general, suponiendo $2 \leq h \leq n$, para extraer el vehículo utilizado en la etapa $h-1$, habiendo extraído el vehículo de la etapa h de la posición $h_{i,j}$ con $0 \leq i \leq k_b$ y $0 \leq j \leq k_m$, por cómo está hecha la tabla, debemos ir a:

- la posición $(h-1)_{i,j}$ si en la etapa h elegimos la bici.

- la posición $(h-1)_{i,j-1}$ si en la etapa h elegimos la moto.
- la posición $(h-1)_{i-1,j}$ si en la etapa h elegimos el buggy.

Y en la posición a la que haya ido, extraer el vehículo utilizado, que será el correspondiente a la etapa $h-1$. Haciendo esto en orden, se obtendría qué vehículo fue utilizado en cada etapa. De esta manera, se resolvería el problema de manera correcta.

3.3. Análisis de complejidad.

Para empezar a analizar la complejidad de nuestro algoritmo veamos brevemente cómo funciona la función ELECCIÓN. Ésta, por medio de una serie de simples comparaciones entre los tiempos de la bicicleta, la moto y el buggy en $\mathcal{O}(1)$, devuelve el valor y el vehículo que, para la etapa en cuestión, resultan óptimos.

Partiendo de esta base, observemos que nuestro algoritmo se encarga de rellenar n tablas de tamaño $km * kb$, haciendo uso de ELECCIÓN, cuya complejidad ya dijimos, es $\mathcal{O}(1)$. En consecuencia, este proceso posee una complejidad de $\mathcal{O}(n * km * kb)$.

Hasta aquí hemos cubierto la función DAKKAR (Figura 1), la cual como veremos en breve, es la que termina asginando la complejidad total del algoritmo planteado. Esto se debe a que, una vez completadas las n tablas, el tiempo total incurrido se obtiene accediendo a la posición (kb, km) de la última tabla generada, y luego se recorre, dependiendo de la elección del vehículo, una casilla determinada de cada tabla "etapa", agregando el vehículo utilizado en la lista solución. Como tenemos n tablas y tanto acceder a una posición de una tabla como agregar elementos al principio de una lista es $\mathcal{O}(1)$, este proceso toma $\mathcal{O}(n)$.

A continuación, mostramos el resultado, que consiste en recorrer una lista de n elementos y mostrar cada uno de estos en $\mathcal{O}(1)$ (Figura 2).

Para concluir, sea $T(n)$ la complejidad de nuestro algoritmo, tenemos:

$$T(n) = \mathcal{O}(n * km * kb) + 2\mathcal{O}(n)$$
$$T(n) = \mathcal{O}(n * km * kb)$$

Observemos que, por como se planteó nuestra solución, la complejidad en "mejor caso" y en "peor caso" de nuestro algoritmo son iguales. Esto sucede ya que en cualquier caso que se plantee, la función DAKKAR, que por lo dicho anteriormente es la que termina asignando la complejidad de la función, debe rellenar n veces tablas de $km * kb$.

```
DAKKAR(n, km, kb, bicis, motos, buggies)
 1  etapa_1(0,0) ← [bicis_1,bici]
 2  for j = 1 to km
 3      etapa_1(0,j) ← ELECCIÓN(bicis_1,motos_1)
 4  for i = 1 to kb
 5      etapa_1(i,0) ← ELECCIÓN(bicis_1,buggies_1)
 6  for i = 1 to kb
 7      for j = 1 to km
 8          etapa_1(i,j) ← ELECCIÓN(bicis_1,motos_1,buggies_1)
 9  for h = 1 to n
10      etapa_h(0,0) ← [etapa_{h-1}(0,0)+bicis_h, bici]
11      for j = 1 to km
12          etapa_h(0,j) ← ELECCIÓN(etapa_{h-1}(0,j) + bicis_h,etapa_{h-1}(0,j-1) + motos_h)
13      for i = 1 to kb
14          etapa_h(i,0) ← ELECCIÓN(etapa_{h-1}(i,0) + bicis_h,etapa_{h-1}(i-1,0) + buggies_h)
15      for i = 1 to kb
16          for j = 1 to km
17              etapa_h(i,j) ← ELECCIÓN(etapa_{h-1}(i,j) + bicis_h,etapa_{h-1}(i,j-1) + motos_h,
                                        etapa_{h-1}(i-1,j) + buggies_h)
```

Figura 1: Pseudocódigo del algoritmo para Dakkar

```
DAKKAR_SALIDA(etapas, km, kb)
 1  i ← kb
 2  j ← km
 3  res ← lista vacía de vehículos
 4  mostrar tiempo_total ← etapa_n(i,j).tiempo
 5  for h = n to 1
 6      vehículo ← etapa_h(i,j).vehiculo
 7      res ← agregar vehículo al principio
 8      if vehículo == moto
 9          decrementar j
10      else if vehículo == buggy
11              decrementar i
12  Recorrer la lista res y mostrar los elementos
```

Figura 2: Pseudocódigo del armado de la salida de Dakkar

3.4. Experimentación y gráficos.

3.4.1. Test 1

3.4.2. Test 2

4. Problema 2: Zombieland II

4.1. Descripción del problema.

Luego del éxito rotundo del último ataque contra los zombies producido por nuestro anterior algoritmo, la humanidad ve un rayo de esperanza.

En una de las últimas ciudades tomadas por la amenaza Z, se encuentra un científico que afirma haber encontrado la cura contra el zombieismo, rodeado por una cuadrilla de soldados del más alto nivel. Nuestro noble objetivo es, entonces, proveer del camino más seguro (el que genere menos perdidas humanas) al científico y sus soldados de manera tal que logren llegar desde donde están, hasta el bunker militar que se encuentra en esa ciudad. Para esto, se conocen los siguientes datos:

- La ciudad en cuestión tiene la forma clásica de grilla rectangular, compuesta por n calles paralelas en forma horizontal, m calles paralelas en forma vertical dando así una grilla de manzanas cuadradas. Los numeros n y m son conocidos.
- Se conoce la ubicación del científico y del bunker.
- Se conoce la cantidad de soldados que el científico tiene a su disposición.
- Si todos los soldados perecen, el científico no tiene posibilidad de sobrevivir por sí mismo (o sea, no podemos llegar al bunker con 0 soldados).
- Se sabe la cantidad de zombies ubicados en cada calle.
- Si bien nuestros soldados tienen un alto nivel de combate, el enfrentamiento que tuvieron en el último ataque contra los zombies, acabaron con todas sus municiones por lo cual solo tienen sus cuchillos de combate. Esto incide entonces, en que el grupo sólo pasará por una calle sin sufrir pérdidas si hay hasta un soldado por zombie, al menos, y en caso contrario, perderá la diferencia entre la cantidad de soldados, y la cantidad de zombies. En otras palabras, sea z la cantidad de zombies de una calle, y s la cantidad de soldados de que tiene la cuadrilla al momento de pasar por esa calle, si $s \geq z$ entonces la cuadrilla pasa sin perdidas; en caso contrario la cuadrilla pierde $z - s$ soldados.
- Puede no existir un camino que asegure la supervivencia del científico.
- La complejidad del algoritmo pedido es de $\mathcal{O}(s \cdot n \cdot m)$.
- La salida de este algoritmo deberá mostrar la cantidad de soldados que llegan vivos al bunker, seguido de una línea por cada esquina del camino recorrido, representada por dos enteros indicando la calle horizontal y la vertical que conforman dicha intersección. En caso de no existir solución, se mostrará el valor 0.

Ejemplo:

Supongamos una ciudad como muestra la Figura 3, donde los números indican la cantidad de zombies en cada cuadra. Consideremos que la cantidad de soldados al comenzar es 10. La solución para este ejemplo permite llegar al búnker sin perder ningún soldado, siendo el recorrido el indicado en la Figura 4.

4.2. Desarrollo de la idea y correctitud.

Debemos recorrer la ciudad en busca de un camino que permita llegar al búnker con la mayor cantidad de soldados vivos. Sin embargo, probar todos los posibles caminos tomaría más tiempo del que disponemos en un ataque zombie. Por lo tanto, hemos decidido buscar dicho camino de la siguiente manera:

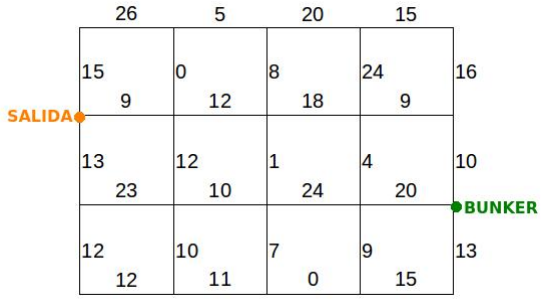

Figura 3: Ejemplo de ciudad para Zombieland II

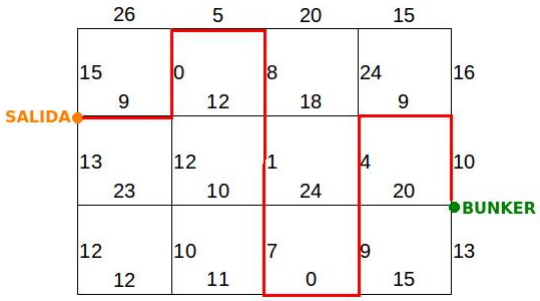

Figura 4: Solución para el problema de la Figura 3

Utilizando *backtraking*, trataremos de encontrar las rutas que salen desde la posición de origen, de manera que ningún soldado muera. Cada cuadra visitada será marcada para que ningún otro camino intente pasar por ahí, y así no tendremos caminos redundantes (es decir, no consideraremos distintos caminos que permitan llegar a una misma esquina con la misma cantidad de soldados vivos), y para cada esquina alcanzada se guardará la esquina anterior de ese camino y la cantidad de soldados vivos hasta ese momento. Si en algún momento se llega a una esquina con alguna cuadra incidente por la cual no se puede avanzar sin pérdidas, vamos a señalizarla (en breve explicaremos para qué).

Si por alguno de estos caminos se llega al búnker, entonces habremos encontrado una solución óptima. Si ninguna de las rutas encontradas es solución, entonces deberemos animarnos a perder soldados. Para ello, primero vamos a arriesgar sólo un soldado, es decir, si teníamos s soldados iniciales, intentaremos llegar al búnker con $s-1$ soldados vivos. Entonces, retomaremos los caminos marcados anteriormente, a partir de las esquinas desde las que no podíamos avanzar sin pérdidas (reanudando cada una en el orden en el que fueron marcadas). Desde ahí, avanzaremos armando los caminos que no nos produzcan más de una baja, y lo haremos de la misma manera que explicamos arriba: guardando la esquina antecesora y la cantidad de soldados vivos en cada esquina atravesada, y sin considerar caminos redundantes. Si al intentar retomar el camino desde una esquina marcada anteriormente, una de las cuadras incidentes produce más de una baja en los soldados, entonces la esquina seguirá estando marcada para alguna etapa posterior.

Nuevamente, si alguno de estos caminos llega al búnker, será la solución. Caso contrario repetiremos el procedimiento intentando llegar con $s-2$ soldados vivos. De forma sucesiva, nos animaremos a perder cada vez un soldado más, y el primer camino encontrado que llegue al búnker será retornado.

Puede suceder que mediante una ruta se llegue a una esquina por la que ya pasó otro camino. Dado que las esquinas marcadas las vamos retomando en el orden en el que fueron marcadas, y que en cada esquina que se retoma se puede perder igual o más número de soldados, entonces el camino que pasó primero por una esquina lo hizo con igual o mayor cantidad de soldados vivos que el segundo. Como no queremos considerar caminos redundantes, si ambas rutas llegan con la misma cantidad de soldados vivos, descartaremos uno de ellos. Y si el primer camino tiene más soldados vivos que el segundo en ese momento, la continuación de ellos podría, o bien, producir pérdidas en los soldados del segundo camino y no en los del primero, o bien producir pérdidas en ambos, pero con menor impacto en el primero que en el segundo. Por lo tanto, resulta conveniente considerar el camino con mayor cantidad de soldados vivos, y por este motivo se decidió que al suceder un cruce de caminos como el expuesto, sólo se tendrá en cuenta aquél que haya ocurrido primero. De esta manera, estamos asegurando que la solución tiene la mayor cantidad de soldados vivos al final.

En caso de llegar al punto de tener un sólo soldado vivo y no encontrar un camino que llegue al búnker, significa que no hay solución.

Una vez que se llega al búnker, se debe "rearmar" el camino correcto. Por lo explicado anteriormente, cada esquina será atravesada por a lo sumo 1 camino, y por lo tanto tendrá a lo sumo 1 esquina antecesora. Entonces basta con consultar consecutivamente ese dato desde la posición del búnker hasta llegar al punto de partida.

Las Figuras 5, 6, 7, 8 y 9 muestran la forma de operar del algoritmo descrito. Notar que luego de encontrar los posibles caminos perdiendo hasta 1 soldado (Figura 7), no existe ningún camino posible

que nos permita avanzar con a lo sumo 2 pérdidas, y por eso se prosigue por un camino que soporte hasta 3 pérdidas (Figura 8).

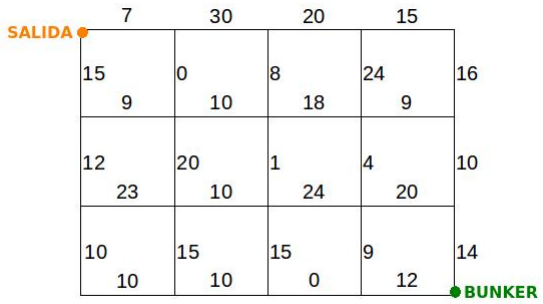

Figura 5: Ejemplo de ciudad - 11 soldados iniciales

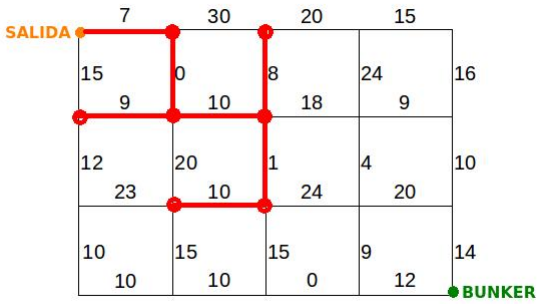

Figura 6: Caminos para la Figura 5 sin pérdidas

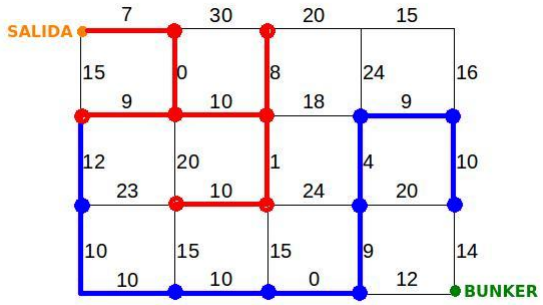

Figura 7: Continuación de Figura 6 con 1 pérdida

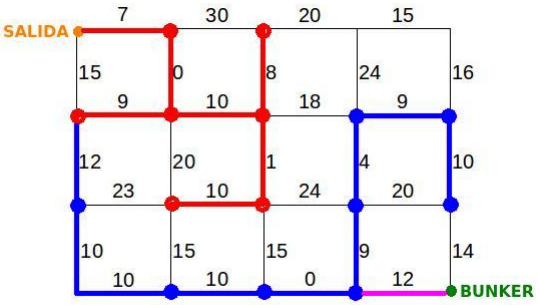

Figura 8: Continuación de Figura 7 con 3 pérdidas

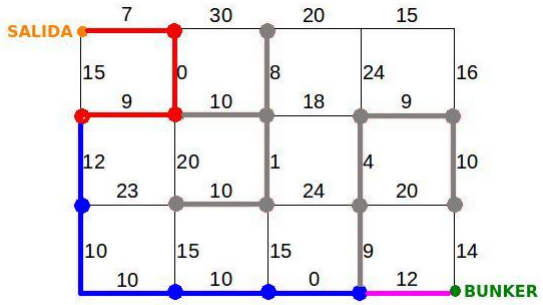

Figura 9: Solución para la Figura 5

4.3. Análisis de complejidad.

Llamemos n la cantidad de calles horizontales y m la cantidad de calles verticales. Tomando $1 \leq i \leq n$ y $1 \leq j \leq m$, la esquina (i, j) será la esquina formada por la intersección de las calles i y j. Para cada esquina definimos los siguientes atributos:

- **Arriba**: cuadra a atravesar para moverse a la esquina $(i-1, j)$. Se considerará inválido cuando $i = 1$.
- **Abajo**: cuadra a atravesar para moverse a la esquina $(i+1, j)$. Se considerará inválido cuando $i = n$.
- **Izquierda**: cuadra a atravesar para moverse a la esquina $(i, j-1)$. Se considerará inválido cuando $j = 1$.

- **Derecha**: cuadra a atravesar para moverse a la esquina $(i, j + 1)$. Se considerará inválido cuando $j = m$.
- **Origen**: nombre de la esquina antecesora, en caso de existir. Si la esquina aún no ha sido visitada, permanecerá vacío.
- **Parcial**: cantidad de soldados vivos al llegar a la esquina desde **Origen**. Si la esquina aún no ha sido visitada, permanecerá vacío.

Veremos ahora la complejidad de dos funciones distintas, ZOMBIELAND (Figura 10) y RECORRIDOS (Figura 11), ambas imprescindibles para nuestra solución.

En primer lugar, deberemos organizar la información de la ciudad, es decir, la cantidad de calles horizontales (n) y verticales (m) y la cantidad de zombies en cada cuadra. Para esto utilizaremos una matriz de $n \times m$, y para $1 \leq i \leq n$ y $1 \leq j \leq m$ la posición (i, j) de la matriz representará a la esquina (i, j). Teniendo esta matriz, completaremos cada posición con los atributos antes mencionados (**arriba, abajo, izquierda** y **derecha** tendrán la cantidad de zombies de dicha cuadra). El costo de completar esta matriz es, entonces, $\mathcal{O}(n \cdot m)$.

Analicemos ahora ZOMBIELAND, la cual toma como parámetro la matriz con los datos de la ciudad,el punto de partida, la posición del búnker y la cantidad de soldados. Luego se procede a inicializar una lista que indicará las esquinas marcadas y a setear como tope la cantidad de soldados con la que pretendemos llegar al búnker. Este tope irá decrementando a medida que nos animemos a perder un soldado más, y se ciclará tantas veces como sean necesarias hasta encontrar un camino o determinar que no hay camino posible. Notemos que en el peor caso tendríamos que ciclar $s - 1$ sólo para encontrarnos con que no hay un camino posible (o sea, intentamos llegar con un único soldado vivo y aún así no encontramos una ruta). En cada iteración, se ejecuta la función RECORRIDOS partiendo de la primera esquina de la lista de marcadas. Por ahora asumamos que dicha función tiene un costo de $\mathcal{O}(n \cdot m)$. Siendo así vemos que en el peor de los casos, nuestra función itera $s - 1$ veces y cada una con un costo de $\mathcal{O}(n \cdot m)$, llegando a una complejidad total de $\mathcal{O}(s \cdot n \cdot m)$.

Pasemos entonces a la función RECORRIDOS. Ésta parte de una posición tomada como parámetro y analiza las distintas posibilidades de movimientos (dirigirse hacia **arriba, abajo, izquierda, derecha**), revisando siempre que estos sean viables, es decir, que la cantidad de zombies de esa cuadra respete la cota de soldados a perder y que no se haya pasado anteriormente por dicha cuadra. Como se trata de simples comparaciones, este chequeo es $\mathcal{O}(1)$. Si en efecto es un movimiento válido, se actualiza el tope y los atributos de la esquina ($\mathcal{O}(1)$) y se llama a la recursión sobre la esquina destino, armando así los recorridos.

Un aspecto crucial de esta función es que no permite repetir caminos y como consecuencia de esto una esquina puede ser visitada un número acotado de veces. Cada cuadra que recorremos es señalizada para no ser transitada nuevamente, determinando así 2 maneras distintas para pasar por una misma esquina, en el caso de que los zombies lo permitan. Por otro lado, si llegada a una esquina y la cantidad de zombies en las posibles direcciones causara que se violara el tope de soldados, esta esquina se visita 1 vez y se la marca para ser revisada en el futuro, con otros posibles topes. Entonces en una misma ciclada, en la que estemos dispuestos a perder x cantidad de soldados, una misma esquina se chequea a lo sumo 2 veces. En el peor caso se deben recorrer todas las esquinas de la ciudad, y como cada una no puede verse más de un número determinado de veces, diremos que la complejidad es $\mathcal{O}(n \cdot m)$.

Para finalizar se procede a armar el resultado y a mostrarlo. Para lo primero partimos desde el búnker y haciendo uso del atributo **origen**, que nos indica desde qué esquina se llegó a la esquina en cuestión, "desandamos" el camino realizado. Por cada retroceso se inserta la esquina nueva en una lista, hasta llegar al punto de partida, en un proceso que en el peor caso (recorriendo todas las esquinas de la ciudad) toma $\mathcal{O}(n \cdot m)$. Para mostrar el resultado se recorre la lista armada anteriormente y se muestra cada esquina recorrida junto con la cantidad de soldados con la cual finalizamos el recorrido, así que volvemos a tener $\mathcal{O}(n \cdot m)$.

Por lo tanto, la complejidad total del algoritmo es $\mathcal{O}(s \cdot n \cdot m)$

```
ZOMBIELAND(ciudad, soldados, inicio, bunker)
 1  tope ← soldados
 2  marcados ← lista vacía de esquinas
 3  marcados ← agregar atrás inicio
 4  res ← false
 5  while tope > 0 y ¬res
 6      while marcados siga teniendo elementos de la etapa anterior
 7          esquina ← desencolar primer elemento de marcados
 8          res ← res + RECORRIDOS(ciudad, marcados, soldados, esquina, bunker, tope)
 9      decrementar tope
10  if ¬res
11      soldados ← 0
12  else
13      soldados ← tope + 1
```

Figura 10: Pseudocódigo de Zombieland II

```
Bool RECORRIDOS(ciudad, marcados, soldados, esquina, bunker, tope)
 1  res ← false
 2  if esquina == bunker
 3      return true
 4  if se puede ir hacia esquina.derecha
 5      if se mueren soldados cumpliendo tope
 6          soldados ← actualizar
 7      siguiente ← esquina a la que se llega
 8      inhabilitar esquina.derecha y siguiente.izquierda
 9      if siguiente.origen y siguiente.parcial están vacíos
10          siguiente.origen ← esquina
11          siguiente.parcial ← soldados
12      res ← res + RECORRIDOS(ciudad, marcados, soldados, siguiente, bunker, tope)
13      if res
14          return res
15  if se puede ir hacia esquina.abajo
16      if se mueren soldados cumpliendo tope
17          soldados ← actualizar
18      siguiente ← esquina a la que se llega
19      inhabilitar esquina.abajo y siguiente.arriba
20      if siguiente.origen y siguiente.parcial están vacíos
21          siguiente.origen ← esquina
22          siguiente.parcial ← soldados
23      res ← res + RECORRIDOS(ciudad, marcados, soldados, siguiente, bunker, tope)
24      if res
25          return res
26  if se puede ir hacia esquina.arriba
27      if se mueren soldados cumpliendo tope
28          soldados ← actualizar
29      siguiente ← esquina a la que se llega
30      inhabilitar esquina.arriba y siguiente.abajo
31      if siguiente.origen y siguiente.parcial están vacíos
32          siguiente.origen ← esquina
33          siguiente.parcial ← soldados
34      res ← res + RECORRIDOS(ciudad, marcados, soldados, siguiente, bunker, tope)
35      if res
36          return res
37  if se puede ir hacia esquina.izquierda
38      if se mueren soldados cumpliendo tope
39          soldados ← actualizar
40      siguiente ← esquina a la que se llega
41      inhabilitar esquina.izquierda y siguiente.derecha
42      if siguiente.origen y siguiente.parcial están vacíos
43          siguiente.origen ← esquina
44          siguiente.parcial ← soldados
45      res ← res + RECORRIDOS(ciudad, marcados, soldados, siguiente, bunker, tope)
46      if res
47          return res
48  if no se pudo avanzar en alguno de los sentidos por exceso de zombies
49      marcados ← agregar al final esquina
50  return false
```

Figura 11: Pseudocódigo para la búsqueda de recorridos en la ciudad

4.4. Experimentación y gráficos.

4.4.1. Test 1

4.4.2. Test 2

5. Problema 3: Refinando petróleo

5.1. Descripción del problema.

Tenemos en cierta zona, pozos de petróleo que necesita ser refinado. Para que un pozo pueda refinar su petróleo, necesita o bien una refinería ubicada junto a él, o bien conectarse mediante tuberías a otro pozo que o tenga una refinería, o esté conectado mediante tuberías a algun pozo que puede refinar su petróleo. Nuestro objetivo es armar un plan de construcción que decida dónde construir una refinería y dónde construir un sistema de tuberías, de manera tal que todos los pozos puedan refinar su petróleo, minimizando el costo. Para esto, se tienen los siguientes datos:

- Se conoce la cantidad de pozos en cuestión.
- Se conoce el costo de construir una refinería (este costo es fijo, y no varía según el pozo).
- Dada la geografía del lugar, no todo par de pozos se puede comunicar por una tubería directamente, y por decisiones de administración, una tubería no puede bifurcarse a mitad de camino entre un pozo y otro. Igualmente, conocemos todos los pares de pozos que pueden conectarse con una tubería, y el costo de ésta en caso de decidir construirse (a diferencia de las refinerías, las tuberías dependen del par de pozos que se quiere conectar).
- La complejidad del algoritmo debe ser estrictamente menor a $\mathcal{O}(n^3)$.
- La salida de este algoritmo debe contener una línea con el costo total de la solución, la cantidad de refinerías y la cantidad de tuberías a construir, seguido de una línea con los números de pozos en los que se construirán refinerías, más una línea con dos números por cada tubería a construir, representando el par de pozos conectados.

Ejemplo:
Supongamos una zona petrolera como muestra la Figura 12, donde los números de las aristas representan el costo de construir dicha tubería. Consideremos que el costo de construir una refinería es 75.

La solución para este ejemplo requiere construir 3 refinerías y 7 tuberías, cuyo costo total es 403. La Figura 13 muestra gráficamente la salida correcta.

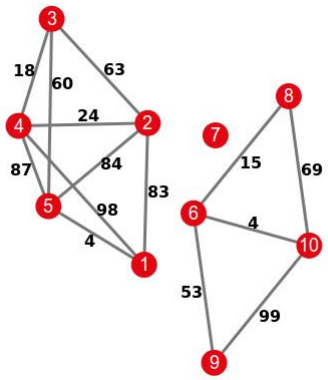

Figura 12: Ejemplo de pozos y posibles tuberías

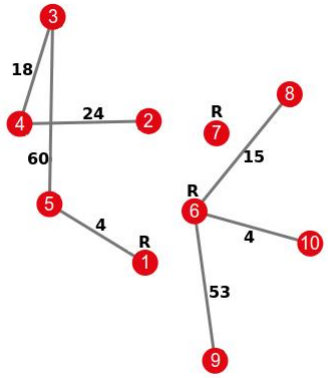

Figura 13: Solución para el problema de la Figura 12

5.2. Desarrollo de la idea y correctitud.

En primer lugar, notemos que el problema a resolver puede ser interpretado con grafos, considerando los pozos petroleros como nodos, y las posibles tuberías y sus costos como aristas con peso. Tomando el

problema de esta manera, podemos replantear el ejercicio como la búsqueda de un subgrafo que contenga a todos los nodos y escoja las aristas que minimicen el gasto.

Observemos que, para que un pozo pueda refinar su petroleo, debe o bien tener una refinería o bien estar conectado por tuberías a algún pozo que tenga refinería. Entonces, en principio, ambas opciones producen el mismo efecto. Además, una vez que el pozo en cuestión puede refinar su petroleo, cualquier pozo que se conecte a él mediante una tubería también podrá hacerlo.

Asumamos por un momento que todas las potenciales tuberías tienen un costo menor o igual que construir una refinería, y que el grafo formado por los pozos y las tuberías posibles fuera conexo. En este caso, el problema se puede traducir como la búsqueda de un árbol generador mínimo (AGM), ya que se trata de un subgrafo que recorre todos los nodos minimizando la suma de los pesos de las aristas. Teniendo esta conexión entre los pozos, bastaría con tomar uno de ellos y colocar allí una refinería. Notemos que esto es correcto, ya que si tenemos todos los nodos conectados, con una sóla refinería es suficiente para que todos ellos refinen su petróleo, colocar más de una sólo incrementaría el costo y no mejoraría la situación. A su vez, si se tienen tuberías con el mismo costo que una refinería, colocar una tubería o una refinería es una cuestión de decisión, puesto que los efectos de ambas opciones son los mismos, por lo dicho en A, y los costos entre una y otra alternativa no varían (nosotros elegiremos utilizar las tuberías).

Sin embargo, sabemos que no todo par de pozos puede conectarse mediante una tubería, y esto puede dar lugar a un grafo inicial no conexo. De todos modos, si seguimos considerando que las tuberías son menor o igual de costosas que una refinería, bastaría con encontrar un AGM para cada componente conexa del grafo inicial, y colocar una refinería en cada uno de ellos. Esto minimizaría el gasto en cada componente y, por lo tanto, en toda la zona petrolera. Veamos que esto es correcto puesto que las distintas componentes conexas no pueden conectarse por medio de tuberías, y se las puede tratar como problemas disjuntos.

Ahora bien, nada garantiza que toda potencial tubería tenga costo menor o igual al de una refinería. Supongamos que tenemos un plan que permite refinar el petróleo de todos los pozos al menor costo, y que este plan tiene al menos una tubería con costo mayor al de una refinería. Siendo r el costo de una refinería, $t > r$ el costo de alguna tubería de la solución mencionada, y A y B los nodos en los que incide dicha tubería, pueden darse los siguientes casos:

1. Ambos pozos poseen una refinería. En este caso, si quitamos la tubería de costo t, A y B siguen pudiendo refinar su petróleo, al igual que todos los pozos conectados a ellos mediante tuberías. Es decir, la estructura resultante permite que todos los nodos refin en su petróleo y como quitamos una tubería, el costo del plan será menor. Esto es absurdo, pues partimos del hecho de que la solución inicial era óptima.

2. Sólo uno de los pozos tiene una refinería (digamos, A). Entonces tanto A como los nodos conectados a él pueden refinar su petróleo sin problemas. Si quitamos la tubería de costo t puede suceder que:

   a) B sigue pudiendo refinar su petróleo en otro lugar. Entonces, al quitar la tubería de costo t obtuvimos una nueva solución, y como quitamos una tubería, el costo del plan será menor. Esto es absurdo, pues partimos del hecho de que la solución inicial era óptima.

   b) B refinaba su petróleo sólo a través de A y ahora ya no puede hacerlo. Si agregamos una refinería en el pozo B, volveríamos a refinar el petróleo de B y el de todos los nodos que se conectan con él. Y como $t > r$, el costo del nuevo plan sería menor, lo cual es absurdo, pues partimos del hecho de que la solución inicial era óptima.

3. Ninguno de los dos pozos tienen refinería. Esto quiere decir que, mediante tuberías, A y B conectan a pozos que sí tienen refinería. Notemos que, si quitamos la tubería de costo t, no es posible que ambos dejen de poder refinar petróleo, pues eso significaría que aún con dicha tubería no podían hacerlo, y eso es absurdo.

   a) Si quitamos la tubería de costo t y ambos pozos siguen conectados a otros nodos con refinería, entonces ambos siguen refinando su petróleo junto con todos los nodos que llegan hasta ellos, y como quitamos una tubería, el costo del plan será menor. Esto es absurdo, pues partimos del hecho de que la solución inicial era óptima.

b) Si al quitar la tubería de costo t alguno de los pozos, digamos A, deja de poder refinar su petróleo, podemos colocar una refinería allí y entonces tanto A como los pozos conectados a él vuelven a refinar su petróleo. Y como $t > r$, el costo del nuevo plan sería menor, lo cual es absurdo, pues partimos del hecho de que la solución inicial era óptima.

Dado que todas las posibilidades que contemplaban una tubería de mayor costo que una refinería dentro de una solución óptima cayeron en un absurdo, concluímos que ninguna solución óptima puede contener una tubería de mayor costo que una refinería, y por esto, no las tendremos en cuenta. Por lo tanto, todas las tuberías a considerar tienen un costo menor o igual a una refinería, y como dijimos anteriormente, puede resolverse mediante la búsqueda de AGM.

Para buscar el AGM, nos hemos basado en el algoritmo de *Kruskal*. Dado que este algoritmo opera sobre grafos conexos, lo hemos adaptado para poder procesar grafos que no necesariamente son conexos. Buscaremos que la solución hallada sea un bosque en el cual cada árbol sea un AGM de cada componente conexa inicial.

Para esto, primero acomodaremos todas las posibles tuberías ordenándolas crecientemente por costo. Así, de manera golosa, iremos tomándolas y colocándolas en la solución, garantizándonos un costo mínimo. A su vez, para asegurarnos de que el resultado sea un bosque, utilizaremos una tubería sólo si no forma circuitos con las colocadas anteriormente. Dado que al ir colocando aristas se van conectando distintos grupos de nodos, vamos a pensarlos como conjuntos disjuntos, de manera de poder consultar si dos nodos forman parte de un mismo conjunto o no. Entonces, al tomar una arista podemos encontrarnos con los siguientes casos:

1. La arista conecta dos nodos que no forman parte de ningún conjunto. En este caso, asociaremos a ambos como parte de un nuevo conjunto.

2. La arista conecta un nodo perteneciente a un conjunto con un nodo que no petenece a ninguno. En este caso, agregaremos al segundo nodo al conjunto al que pertenece el primero.

3. La arista conecta dos nodos pertenecientes a algún conjunto.

   a) Si cada uno pertenece a un conjunto diferente, realizaremos la unión de ambos, ya que la nueva arista los relaciona.

   b) Si ambos pertenecen al mismo conjunto, entonces la arista estaría formando un circuito. En este caso, no agregaremos dicha tubería.

Al terminar de recorrer todas las posibles tuberías, habremos conectado todos los nodos posibles al menor precio.

Notemos que si el grafo inicial es conexo, la forma de proceder es exactamente el algoritmo de *Kruskal*, y sabemos que éste encuentra un AGM. Veamos qué sucede si el grafo inicial no es conexo. Sea c una componente conexa del grafo inicial, y sea e la primera arista que forme parte de ella. Puede suceder que las siguientes k aristas colocadas también pertenezcan a c, y que además generen su AGM; esto quiere decir que hemos encontrado un AGM mediante el algoritmo de *Kruskal* aplicado a esta componente en particular. Por otro lado, puede ocurrir que para generar dicho AGM, se coloquen las aristas correspondientes a c intercaladas con aristas pertenecientes a otras componentes, pero si miramos sólo a esta componente veremos que también se lleva a cabo el algoritmo de *Kruskal* en ella, sólo que con ciertas "pausas" en las que se completan otras componentes. Por lo tanto, podemos decir que nuestra manera de formar el bosque de AGM realiza *Kruskal* en forma "paralela" para todas las componentes conexas del grafo inicial. Observemos que esta bien usado el termino "paralela", puesto que en ningún momento podemos estarnos encargando de más de una componente a la vez, ya que cada componente conexa por definición, no tiene aristas en común con las demás, y en ningún momento podemos no estarnos encargando de alguna componente conexa, puesto que cada arista debe pertenecer al menos a una de ellas.

Entonces, si al terminar de mirar todas las aristas, todos los nodos pertenecen al mismo conjunto, significa que el grafo inicial era conexo, y deberemos agregar sólo una refinería para poder procesar todo el petróleo. Si los nodos quedan agrupados en distintos conjuntos, éstos estarán representando las

distintas componentes conexas del grafo inicial, la forma en que los pozos quedaron conectados será el bosque de AGM que buscábamos y bastará con construir una refinería por conjunto para poder procesar todo el petróleo. Por último, puede ocurrir que existan nodos que no fueron incorporados a ningún conjunto; esto significa que esos pozos desde un principio no tenían comunicación con los demás, por lo cual van a requerir la construcción de una refinería en cada uno.

Para poder mostrar el plan obtenido al finalizar el algoritmo, cada tubería agregada será contada y guardada, al igual que cada refinería que fuese necesaria, para poder conocer cuántas y cuáles tuberías y refinerías se necesitarán. También se irá acumulando el costo de cada tubería a construir, a lo cual se sumará el costo de construir cada refinería, obteniendo así el costo total del plan.

5.3. Análisis de complejidad.

Para implementar nuestro algoritmo de manera eficiente, cada conjunto de pozos tendrá un representante y cada nodo conocerá al representante de su conjunto. De esta manera, si dos nodos tienen al mismo representante significa que ambos pertenecen al mismo conjunto. En caso de que el nodo no haya sido aún incluído en un conjunto, este valor estará indefinido.

5.4. Experimentación y gráficos.

5.4.1. Test 1

5.4.2. Test 2

5.4.3. Test 3

```
PLAN(aristas, n, tuberias, costo_total)
 1  cant_tub ← 0
 2  tam_conjuntos ← arreglo de n elementos
 3  tam_conjuntos ← inicializar en 0
 4  for cada arista
 5      A, B ← nodos incididos por la arista
 6      if A y B pertenecen al mismo conjunto
 7          descartar arista
 8      else
 9          if A y B no pertenecen a ningún conjunto
10              A ← colocar A como representante de conjunto
11              B ← colocar A como representante de conjunto
12              tam_conjuntos[A] ← 2
13          else if A pertenece a un conjunto y B no
14                  p ← representante de conjunto de A
15                  B ← colocar p como representante de conjunto
16                  incrementar tam_conjuntos[p]
17          else if B pertenece a un conjunto y A no
18                  p ← representante de conjunto de B
19                  A ← colocar p como representante de conjunto
20                  incrementar tam_conjuntos[p]
21          else if A y B pertenecen a distintos conjuntos
22                  p ← representante de conjunto de A
23                  q ← representante de conjunto de B
24                  if tam_conjuntos[p] < tam_conjuntos[q]
25                      for cada nodo del conjunto representado por p
26                          colocar q como representante de conjunto
27                      tam_conjuntos[q] ← tam_conjuntos[q] + tam_conjuntos[p]
28                  else
29                      for cada nodo del conjunto representado por q
30                          colocar p como representante de conjunto
31                      tam_conjuntos[p] ← tam_conjuntos[p] + tam_conjuntos[q]
32          tuberias ← agregar arista
33          actualizar costo_total
34          incrementar cant_tub
35  return cant_tub
```

Figura 14: Pseudocódigo del armado del plan

6. Apéndice 1: acerca de los tests

```
PETROLEO(aristas, n, costo_refineria)
 1  costo_total ← 0
 2  tuberias ← lista vacía de aristas
 3  cant_tub ← PLAN(aristas, n, tuberias, costo_total)
 4  cant_ref ← 0
 5  refinerias ← lista vacía de nodos
 6  vistos ← arreglo de n posiciones
 7  for cada nodo
 8      if no pertenece a ningún conjunto
 9          refinerias ← agregar nodo
10          incrementar cant_ref
11      else
12          p ← representante del conjunto al que pertenece el nodo
13          if p todavía no fue visto
14              vistos ← marcar posición p
15              refinerias ← agregar nodo
16              incrementar cant_ref
17  costo_total ← costo_total + costo_refineria × cant_ref
```

Figura 15: Pseudocódigo de Petróleo

7. Apéndice 2: secciones relevantes del código

En esta sección, adjuntamos parte del código correspondiente a la resolución de cada problema que consideramos más **relevante**.

7.1. Código del Problema 1

7.2. Código del Problema 2

7.3. Código del Problema 3