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

El objetivo principal del siguiente trabajo será la práctica de distintas técnicas para la creación de algoritmos. Dados tres problemas, que representan situaciones muy factibles de la vida real, se buscará resolverlos de forma óptima mediante la aplicación de algoritmos de programación dinámica y planteamientos del problema en base a grafos. Para cada situación, se buscará que el algoritmo no supere cierta cota de complejidad temporal en peor caso.

2. Plataforma de pruebas

Para toda la experimentación se utilizará un procesador Intel Core i3, de 4 nucleos a 2.20 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.

Se llevarán a cabo distintos experimentos para mostrar de manera empírica ciertos aspectos de nuestro algoritmo que, hasta ahora, se encuentran de manera teórica. Queremos mostrar que, efectivamente, nuestro algoritmo tiene una complejidad de $\mathcal{O}(n * km * kb)$.

3.4.1. Test 1

Primeramente, se ejecutó *Dakkar* con 15 instancias generadas de la siguiente manera:

- Se estableció la cantidad de etapas en 50, y se generaron los tiempos por etapa para cada vehículo de manera aleattoria.
- Se generaron las demás instancias, agregando de a 10 etapas con tiempos aleatorios para cada vehículo.
- Se fijó la cantidad de buggies y de motos en 100.

Cada instacia fue ejecutada 20 veces y se consideró el promedio del tiempo incurrido en cada ejecución. Los resultados pueden verse en la Figura 3.

Luego, se tomó una instancia particular de 100 etapas, con los tiempos de cada vehículo generados de manera aleatoria, y se hizo variar la cantidad de buggies y motos de la siguiente manera:

- Se fijó la cantidad de motos en 80 y se hizo variar la cantidad de buggies entre 20 y 140.
- Se fijó la cantidad de buggies en 80 y se hizo variar la cantidad de motos entre 20 y 140.

Nuevamente, cada caso fue ejecutado 20 veces y se consideró el promedio del tiempo incurrido en cada ejecución. Los resultados se ven en las Figuras 5 y 4 respectivamente.

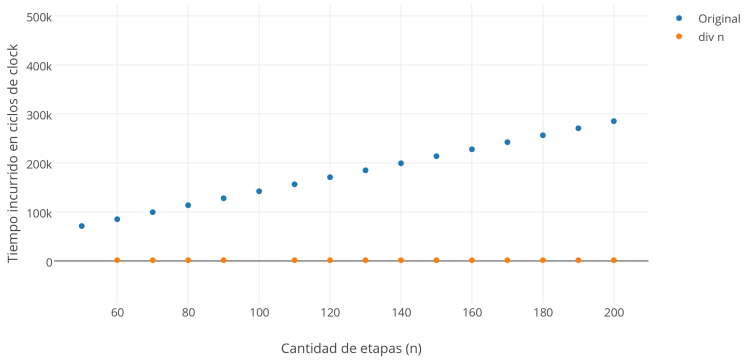

Figura 3: Dakkar - Complejidad con etapas variables

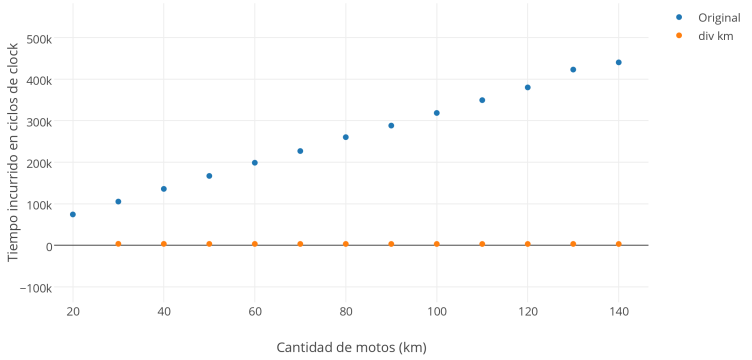

Figura 4: Dakkar - Complejidad con motos variables

Podemos observar en la Figura 3 ,y en las Figuras 5 y 4, teniendo en cuenta las tres variables, cantidad de etapas, cantidad de buggies y cantidad de motos, al dejar constante dos de ellas, el tiempo que incurre

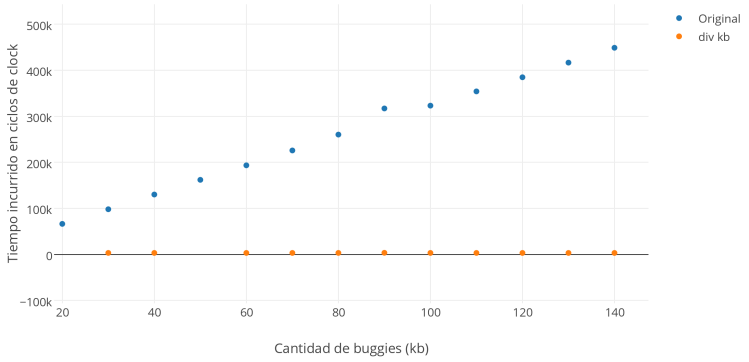

Figura 5: Dakkar - Complejidad con buggies variables

la tercera parecería ser lineal dado que estaríamos observando una recta. Para poder fundamentar que lo que vemos es en efecto, una recta, los gráficos toman la variable cantidad de etapas, y dividen el tiempo incurrido por este valor. Dado que el resultado en ellos es una constante, podemos afirmar en base a estos, que si dejamos dos variables constantes, la tercera crece de forma lineal. Como consecuencia de esto, se ve reafirmado el analisis teórico, y que nuestro algoritmo tiene complejidad temporal $\mathcal{O}(n * km * kb)$.

3.4.2. Test 2

En este caso, se fijó la cantidad de etapas en 100, la cantidad de motos en 30 y la cantidad de buggies en 40, y con estos valores se generaron 100 instancias distintas, con los tiempos de cada etapa elegidos de manera aleatoria. Para cada una de estas instancias, se corrió *Dakkar* 20 veces y se consideró el promedio del tiempo incurrido en cada una. Los resultados pueden verse en la Figura 6.

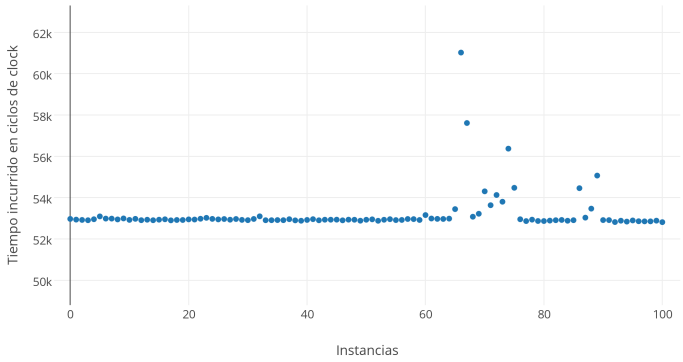

Figura 6: Dakkar - Variables constantes, instancias variables

Podemos observar que manteniendo constantes todas las variables, pero alterando los tiempos de todas las etapas, no se pueden apreciar diferencias notables respecto al tiempo. Esto respalda nuestra teoría acerca de que nuestro algoritmo no depende de los valores, sino de las variables, (o sea que no parecería haber un mejor o peor caso temporal).

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 7, 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 8.

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:

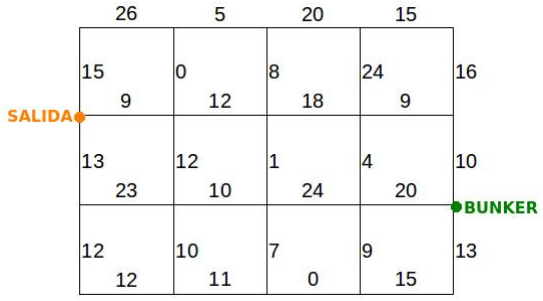

Figura 7: Ejemplo de ciudad para Zombieland II

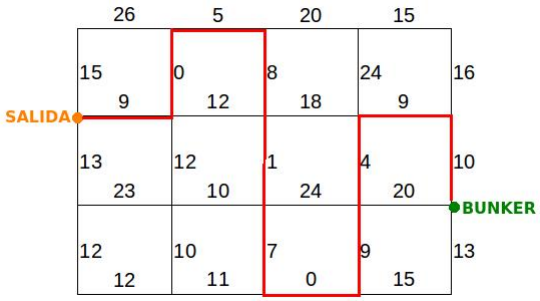

Figura 8: Solución para el problema de la Figura 7

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 9, 10, 11, 12 y 13 muestran la forma de operar del algoritmo descrito. Notar que luego de encontrar los posibles caminos perdiendo hasta 1 soldado (Figura 11), 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 12).

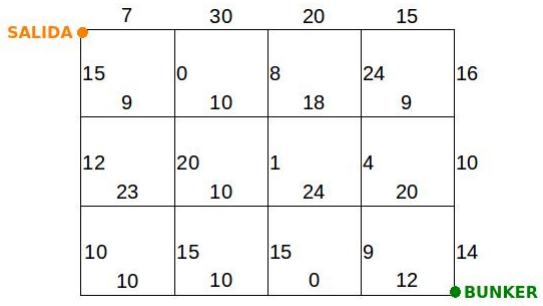

Figura 9: Ejemplo de ciudad - 11 soldados iniciales

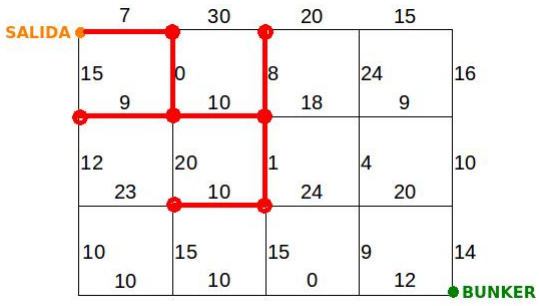

Figura 10: Caminos para la Figura 9 sin pérdidas

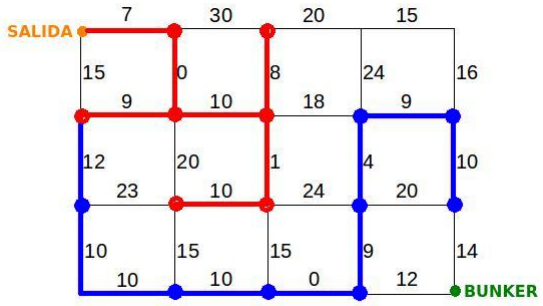

Figura 11: Continuación de Figura 10 con 1 pérdida

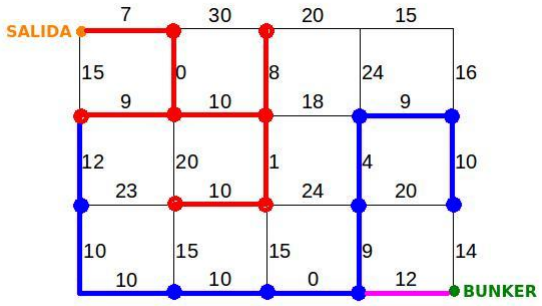

Figura 12: Continuación de Figura 11 con 3 pérdidas

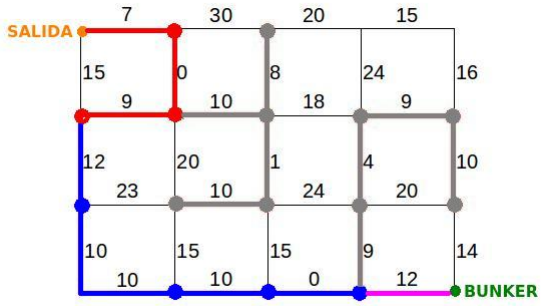

Figura 13: Solución para la Figura 9

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 14) y RECORRIDOS (Figura 15), 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 14: 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 15: Pseudocódigo para la búsqueda de recorridos en la ciudad

4.4. Experimentación y gráficos.

En esta sección se tratará de contrastar el materiar teórico brindado anteriormente, con experimentación.

4.4.1. Test 1

Primero, se fijó la cantidad de calles horizontales y verticales en 29, y la cantidad de soldados en 841. Para estos valores, se generaron 30 instancias donde la cantidad de zombies de cada cuadra fue tomado aleatoriamente. En particular, se generó una instancia donde la solución requiriera recorrer todas las esquinas, y una instancia en la que existiera un camino directo sin zombies, considerándolos peor y mejor caso. El incio se estableció en la esquina superior izquierda, y el bunker en la esquina inferior derecha.

Se corrió nuestro algoritmo con cada una de estas instancias 20 veces, y se tomó el promedio del tiempo incurrido en ellas. Los resultados pueden verse en la Figura 16.

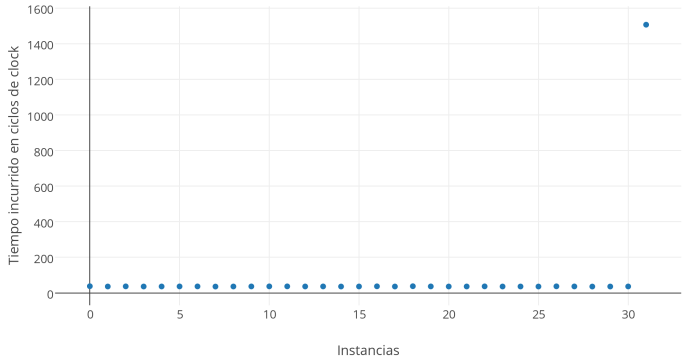

Figura 16: Zombieland II - Instancias aleatorias

En la Figura 16 se puede notar como, manteniendo el tamaño de la ciudad constante y modificando los valores de las distintas calles, se generan resultados extremadamente similares. Se puede apreciar una diferencia abismal entre lo que consideramos el peor caso, y el resto de los casos, siendo el que más tiempo tarda. En cambio el considerado mejor caso, donde hay un claro recorrido donde se pierden 0 soldados, la diferencia no termina siendo realmente apreciable en comparación con las otras instancias generadas de manera aleatoria.

Para resumir, en general, instancias generadas de manera aleatoria no deberían perjudicar demasiado la performance de nuestro algoritmo. Pero sin embargo, parecieran existir instancias particulares que perjudican gravemente a nuestro algoritmo en términos de tiempo.

4.4.2. Test 2

En este caso, se tomó una de las instancias generadas en el Test 1, sabiendo que con la cantidad de soldados iniciales asignados arbitrariamente, existe una solución. Entonces, tomando esa cantidad de soldados, fuimos incrementándola y repitiendo la ejecución para contrastar los tiempos incurridos. Por cada incremento, se ejecutó el algoritmo 20 veces y se tomó el promedio de ellas en cuanto a tiempo. Los resultados se muestran en la Figura 17.

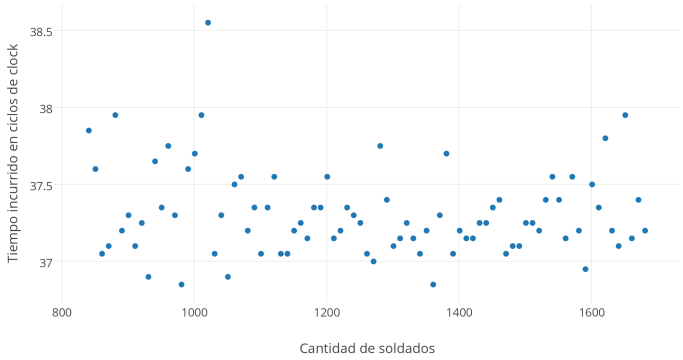

Figura 17: Zombieland II - Variando cantidad de soldados

Lo que nos permite notar la Figura 17 es que a partir de cierta cantidad de soldados, aumentar esta cantidad no resulta en cambios distinguibles para nuestro algoritmo. Esto parecería ser verdad dado que a partir de cierto número de soldados x, si nuestro algoritmo encuentra un camino que le permita perder 0 soldados, ese mismo camino podría ser aplicable a una cantidad soldados y con $y > x$.

Quedaría pendiente contrastar el hecho de que disminuyendo la cantidad de soldados la diferencia sería mucho más apreciable.

4.4.3. Test 3

Para este experimento, se tomaron las instancias generadas en el Test 1, cuando tuvieran solución, y se realizaron "bloqueos" de manera de impedir la llegada de los soldados al búnker. Estos "bloqueos" consistieron en modificar la cantidad de zombies en 3 puntos particulares:

- En las calles aledañas al bunker.
- En las calles aledañas al inicio.
- En las calles que permitiran pasar de la mitad superior hacia la mitad inferior de la ciudad.

Cada caso fue ejecutado 20 veces, y se consideró el promedio en tiempo. Los resultados pueden verse en la Figura 18.

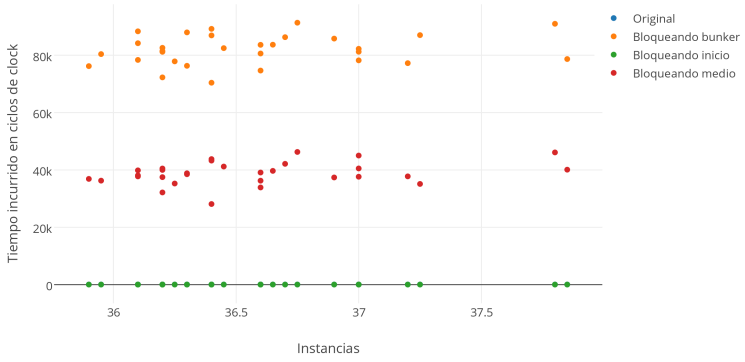

Figura 18: Zombieland II - Instancias con bloqueos

Resulta evidente que, de bloquear los posibles movimientos desde el punto de partida, nuestro algoritmo presentaría un tiempo de ejecución extremadamente pequeño, al no poder analizar ningun camino. En el caso de bloquear el bunker, como resulta necesario en este caso llegar al final para notar que no hay un camino posible, esto desemboca en tener que analizar más caminos que de costumbre, dado que los posibles caminos que eran posibles sin los bloqueos, ya no están disponibles. Finalmente, notamos que al realizar un bloqueo a la mitad de la ciudad, los tiempos resultantes son intermedios a los casos anteriores.

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 19, 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 20 muestra gráficamente la salida correcta.

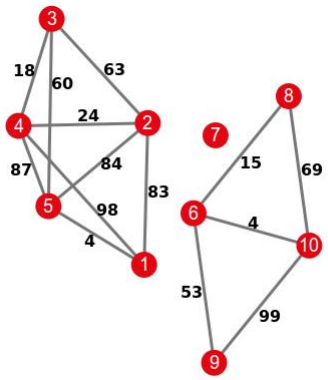

Figura 19: Ejemplo de pozos y posibles tuberías

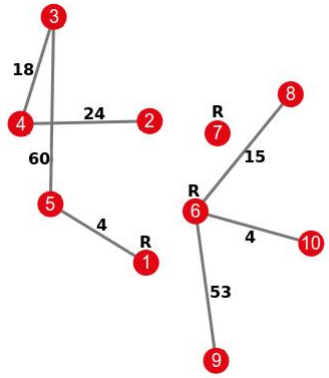

Figura 20: Solución para el problema de la Figura 19

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.

Uno de los primeros pasos a realizar de nuestro algoritmo es la organización de los datos que poseemos (posibles tuberías con sus respectivos costos). Como ya hemos mencionado en el apartado "Desarrollo de la la idea y correctitud", nos gustaría tener estos datos ordenados. Para eso los almacenaremos en una lista de incidencias la cual ordenaremos de manera creciente segun su precio, con un costo total de, siendo x la cantidad de tuberías posibles $\mathcal{O}(x \cdot log(x))$.

Una vez finalizado esto, pasamos a una de las funciones claves de nuestro algoritmo, PETROLEO. Esta utiliza la función PLAN para, como su nombre lo indica, armar el plan que nos permita, para cada componente conexa del grafo original, conectar sus pozos con el menor costo posible, acumulando el gasto total. Con el plan ya terminado, lo que le resta hacer a PETROLEO es colocar una refinería en cada uno de los pozos aislados y en el pozo representante de cada componente conexa, agregando el costo de cada nueva refinería al gasto total. Dejando de lado la función PLAN, es fácil ver que la complejidad de la función consta de revisar cada nodo y hacer ciertas operaciones menores $\mathcal{O}(1)$, además de inicializar algunas variables $\mathcal{O}(1)$, llegando a un costo de $\mathcal{O}(n)$.

Ahora veamos que complejidad nos presenta la función PLAN. Es necesario, para empezar, contar con un arreglo para guardar el tamaño de las componentes conexas, siendo inicializado cada elemento como 0 $\mathcal{O}(n)$ (podrían haber n componentes conexas). Luego para cada elemento de $aristas$ se realiza el proceso de completar las distintas componentes conexas eligiendo sus representantes $\mathcal{O}(1)$, modificando el tamaño de las componentes conforme se modifican $\mathcal{O}(1)$. Un caso interesante que puede darse durante este procedimiento es cuando dos componentes conexas se ven relacionadas por una nueva arista, de ser así, el conjunto más pequeño se ve absorbido por el más grande copiándose sus elementos. Sería correcto notar que, de ser un grafo conexo, la suma de los elementos copiados, cualquiera sea el caso (dos componentes conexas que se ven unidas al final, muchas componentes conexas pequeñas que se terminan uniendo, etc) la suma de los elementos copiados termina siendo, a lo sumo, $n - 1$.

Hablemos ahora del peor caso posible para nuestro algoritmo. En este, para cada pozo disponible tenemos una cantidad de tuberías posibles $n - 1$ (cada pozo se puede conectar con todos los demás). En este contexto nuestra lista de incidencias tendría un tamaño de $n \cdot (n - 1)$ (crearla y ordenarla costaría $\mathcal{O}(n^2 \cdot log(n^2))$ la cual PLAN tendría que recorrer con los costos antes mencionados, llegando así a una complejidad de $\mathcal{O}(n \cdot (n - 1)) + \mathcal{O}(n - 1)$. Esto puede verse como $\mathcal{O}(n \cdot n) + \mathcal{O}(n - 1)$.

Por ende, siendo T(n) la complejidad de nuestro algoritmo, en el peor caso queda definida por la suma de los costos de las funciones PETROLEO PLAN y los costos de organizar los datos y mostrarlos(tomado como $\mathcal{O}(n)$:

$$T(n) = 2\mathcal{O}(n \cdot n) + 2\mathcal{O}(n) + \mathcal{O}(n-1) + \mathcal{O}(n^2 \cdot log(n^2))$$
$$T(n) = 2\mathcal{O}(n^2) + \mathcal{O}(n^2 \cdot log(n^2))$$
$$T(n) = \mathcal{O}(n^2 \cdot log(n^2))$$
$$T(n) = \mathcal{O}(n^2 \cdot (log(n) + log(n))$$
$$T(n) = \mathcal{O}(n^2 \cdot log(n))$$

Entonces nuestra solución tiene una complejidad estrictamente menor que $\mathcal{O}(n^3)$

```
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 21: Pseudocódigo del armado del plan

```
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 22: Pseudocódigo de Petróleo

5.4. Experimentación y gráficos.

A continuación, buscaremos contrastar el análisis teórico previo con experimentación. De esta manera, intentaremos verificar de manera empírica que la complejidad de nuestro algoritmo es $\mathcal{O}(n^2 \cdot log(n))$.

5.4.1. Test 1

Para esta experimentación, fijamos el valor de una refinería en 50 y creamos instancias de posibles tuberías, asignando pesos aleatorios, partiendo de 10 nodos e incrementando este valor consecutivamente. Para cada instancia generada, se corrió *Petróleo* 20 veces y se tomó el promedio de los tiempos incurridos. Los resultados se ven en la Figura 23.

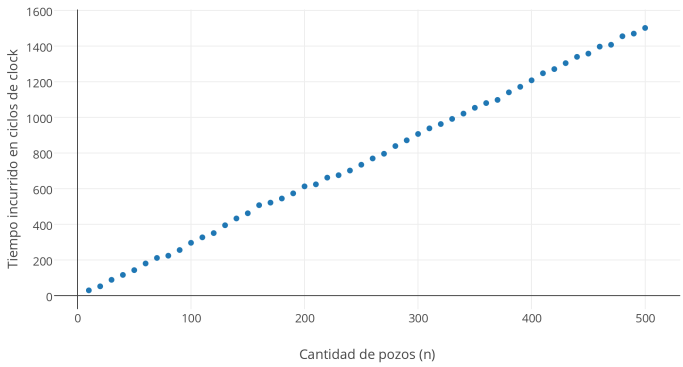

Figura 23: Petróleo - Cantidad de pozos variable

A simple vista la Figura 23 no parece muy clara en cuanto a si es una recta, una curva, o una curva

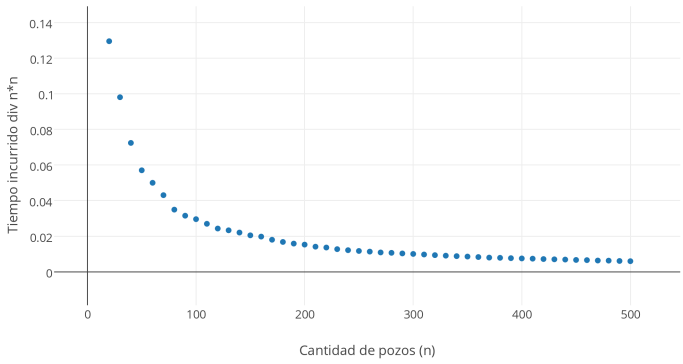

Figura 24: Petróleo - Cantidad de pozos variable (div n*n)

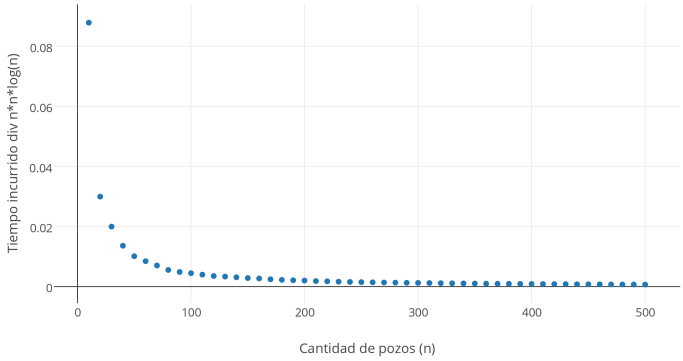

Figura 25: Petróleo - Cantidad de pozos variable (div n*n*log(n))

muy pronunciada. Por esto, recurrimos a observar la Figura 24 y observamos que los valores parece estabilizarse en una constante muy cercana al cero. Si bien en principio se ve una abrupta caida de tiempo, hay que tener en cuenta que los valores de la Figura 23 se mueven de 0 a 1600 mientras que los valores de la figura 24 se mueven de 0 a 0.15, por lo que consideramos estos valores “despreciables” y consideramos que el numero se aproxima a una constante. Para apoyar esta decisión, la figura 25 muestra el mismo compartamiento que la figura 24, solo que con numeros aún más chicos, lo que tiene coherencia, puesto que estan divididos por log n. Esto parecería indicar que la complejidad de nuestro algoritmo es de $\mathcal{O}(n^2)$, y no de $\mathcal{O}(n^2 \cdot log(n))$ como indica nuestro analisis teórico.

5.4.2. Test 2

Para esta etapa, se crearon diversas instancias y se evaluaron de la siguiente manera:

- Se generaron 2 instancias separadas y se corrieron por separado.
- Luego se fusionaron ambos grafos y se corrió como una sola instancia.

Finalmente se tomaron los promedios de cada caso. Los resultados se ven en la Figura 26, donde las primeras dos columnas representan las instancias separadas y la tercera representa la fusión.

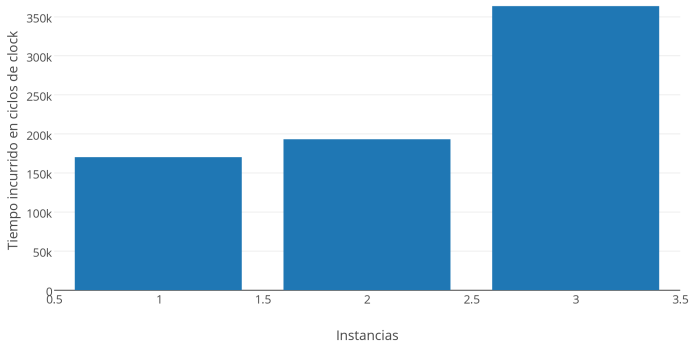

Figura 26: Petróleo - Fusión de grafos

En la figura 26 observamos que al fusionar sistemas de tuberías, el tiempo obtenido es aproximadamente la suma de las dos tuberías fusionadas en cuestión. Esto apoya nuestra teoría de que nuestro algoritmo trabaja en paralelo las distintas componentes conexas con el metodo de kruscal, puesto que no hay diferencia notable entre correr dos sistemas de tuberías distintos uno a la vez, o los dos juntos.

6. Apéndice 1: acerca de los tests

Para reproducir los tests mencionados en el presente informe, se adjunta con la entrega el script *tests_nuestros.sh*. Además, se provee el script *compilar.sh* para compilar los programas, y el archivo *testing.cpp* que se ocupa de correr los tests para los tres problemas. Estos dos últimos se llaman desde *tests_nuestros.sh*. El script mencionado se ocupa de realizar las compilaciones necesarias y correr cada una de las pruebas. Los resultados se guardarán en un directorio llamado *Resultados_tests_nuestros*.

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

```
Carrera dakkar(int n, int km, int kb, int* bici, int* moto, int* buggy)
{
        int h;
        int i;
        int j;
        Optimo aux;
        Carrera etapas(n, Etapa(kb+1, Vec(km+1)));

        // Lleno los datos de la primera etapa (etapas[0])

        // posicion (0,0)
        etapas[0][0][0] = make_pair(bici[0],1);

        // posiciones (0,1) a (0,km)
        aux = eleccion(bici[0],moto[0],-1);
        for(j = 1; j <= km; j++){
                etapas[0][0][j] = aux;
        }

        // posiciones (1,0) a (kb,0)
        aux = eleccion(bici[0],-1,buggy[0]);
        for(i = 1; i <= kb; i++){
                etapas[0][i][0] = aux;
        }

        // demas posiciones
        aux = eleccion(bici[0],moto[0],buggy[0]);
        for(i = 1; i <= kb; i++){
                for(j = 1; j <= km; j++){
                        etapas[0][i][j] = aux;
                }
        }

        // Armo las demas etapas (etapas[1] a etapas[n-1])
        for(h = 1; h < n; h++)
        {
                // posicion (0,0)
                etapas[h][0][0] = make_pair(etapas[h-1][0][0].first + bici[h],1);

                // posiciones (0,1) a (0,km)
                for(j = 1; j <= km; j++){
                        aux = eleccion(etapas[h-1][0][j].first + bici[h],etapas[h-1][0][j-1].
                            first + moto[h],-1);
                        etapas[h][0][j] = aux;
                }

                // posiciones (1,0) a (kb,0)
                for(i = 1; i <= kb; i++){
                        aux = eleccion(etapas[h-1][i][0].first + bici[h],-1,etapas[h-1][i
                            -1][0].first + buggy[h]);
                        etapas[h][i][0] = aux;
                }

                // demas posiciones
                for(i = 1; i <= kb; i++)
                {
                        for(j = 1; j <= km; j++)
                        {
```

```
                                aux = eleccion(etapas[h-1][i][j].first + bici[h],etapas[h-1][i
                                    ][j-1].first + moto[h],etapas[h-1][i-1][j].first + buggy[h
                                    ]);
                                etapas[h][i][j] = aux;
                        }
                }
        }

        return etapas;
}


int armo_salida(Carrera etapas, int km, int kb, int n, list<int>& salida)
{
/* Funcion que recorre las matrices de etapas y determina el tiempo
 * total incurrido y el vehiculo utilizado en cada etapa.  Empieza
 * en la ultima etapa y termina con la primera.
 */
        int i = kb;
        int j = km;
        int res = etapas[n-1][i][j].first;      // tiempo total incurrido

        for(int h = n-1; h >= 0; h--)
        {
                // apilo vehiculo
                salida.push_front(etapas[h][i][j].second);

                // acomodo indices
                if(etapas[h][i][j].second == 2){
                        j--;
                }else if(etapas[h][i][j].second == 3){
                        i--;
                }
        }

        return res;
}
```

7.2. Código del Problema 2

```
bool recorridos(Mapa& ciudad, list<pair <pos,int> >& cola, int soldados, pos posicion, pos
    bunker, int& cont, int tope)
{
/* Funcion que recorre el mapa buscando llegar al bunker */

        bool res;
        pos pos_aux;
        pair <pos,int> bp;
        int soldAr;
        int soldAb;
        int soldI;
        int soldD;

        int i = posicion.horizontal;
        int j = posicion.vertical;

        if(posicion == bunker)  // si ya estoy en el bunker, me voy
        {
                return true;
        }

        if(ciudad[i][j].derecha != -1)  // si es calle valida
        {
                // si puedo pasar por esa calle
                if((ciudad[i][j].derecha <= soldados) || (2*soldados - ciudad[i][j].derecha >=
                     tope))
                {
```

```
                    if(ciudad[i][j].derecha <= soldados){
                            // si no se muere nadie
                            soldD = soldados;
                    }else{
                            // si se me mueren soldados
                            soldD = 2*soldados - ciudad[i][j].derecha;
                    }

                    // aviso que ya pase por esta cuadra
                    ciudad[i][j].derecha = -1;
                    ciudad[i][j+1].izquierda = -1;

                    // esquina a la que llego
                    pos_aux.horizontal = i;
                    pos_aux.vertical = j+1;

                    if(soldD > ciudad[i][j+1].parcial)      // si vale la pena
                    {
                            // guardo de donde vine
                            ciudad[i][j+1].origen = posicion;

                            // cantidad de soldados vivos hasta aca
                            ciudad[i][j+1].parcial = soldD;
                    }

                    // recursion
                    res = recorridos(ciudad, cola, soldD, pos_aux, bunker,cont,tope);
                    if(res){ return res; }
            }
    }

    if(ciudad[i][j].abajo != -1)    // si es calle valida
    {
            // si puedo pasar por esa calle
            if((ciudad[i][j].abajo <= soldados) || (2*soldados - ciudad[i][j].abajo >=
                tope))
            {
                    if(ciudad[i][j].abajo <= soldados){
                            // si no se muere nadie
                            soldAb = soldados;
                    }else{
                            // si se me mueren soldados
                            soldAb = 2*soldados - ciudad[i][j].abajo;
                    }

                    // aviso que ya pase por esta cuadra
                    ciudad[i][j].abajo = -1;
                    ciudad[i+1][j].arriba = -1;

                    // esquina a la que llego
                    pos_aux.horizontal = i+1;
                    pos_aux.vertical = j;

                    if(soldAb > ciudad[i+1][j].parcial)     // si vale la pena
                    {
                            // guardo de donde vine
                            ciudad[i+1][j].origen = posicion;

                            // cantidad de soldados vivos hasta aca
                            ciudad[i+1][j].parcial = soldAb;
                    }

                    // recursion
                    res = recorridos(ciudad, cola, soldAb, pos_aux, bunker,cont,tope);
                    if(res){ return res; }
            }
    }
```

```
if(ciudad[i][j].arriba != -1)   // si es calle valida
{
        // si puedo pasar por esa calle
        if((ciudad[i][j].arriba <= soldados) || (2*soldados - ciudad[i][j].arriba >=
            tope))
        {
                if(ciudad[i][j].arriba <= soldados){
                        // si no se muere nadie
                        soldAr = soldados;
                }else{
                        // si se me mueren soldados
                        soldAr = 2*soldados - ciudad[i][j].arriba;
                }

                // aviso que ya pase por esta cuadra
                ciudad[i][j].arriba = -1;
                ciudad[i-1][j].abajo = -1;

                // esquina a la que llego
                pos_aux.horizontal = i-1;
                pos_aux.vertical = j;

                if(soldAr > ciudad[i-1][j].parcial)     // si vale la pena
                {
                        // guardo de donde vine
                        ciudad[i-1][j].origen = posicion;

                        // cantidad de soldados vivos hasta aca
                        ciudad[i-1][j].parcial = soldAr;
                }

                // recursion
                res = recorridos(ciudad, cola, soldAr, pos_aux, bunker,cont,tope);
                if(res){ return res; }
        }
}

if(ciudad[i][j].izquierda != -1)        // si es calle valida
{
        // si puedo pasar por esa calle
        if((ciudad[i][j].izquierda <= soldados) || (2*soldados - ciudad[i][j].
            izquierda >= tope))
        {
                if(ciudad[i][j].izquierda <= soldados){
                        // si no se muere nadie
                        soldI = soldados;
                }else{
                        // si se me mueren soldados
                        soldI = 2*soldados - ciudad[i][j].izquierda;
                }

                // aviso que ya pase por esta cuadra
                ciudad[i][j].izquierda = -1;
                ciudad[i][j-1].derecha = -1;

                // esquina a la que llego
                pos_aux.horizontal = i;
                pos_aux.vertical = j-1;

                if(soldI > ciudad[i][j-1].parcial)      // si vale la pena
                {
                        // guardo de donde vine
                        ciudad[i][j-1].origen = posicion;

                        // cantidad de soldados vivos hasta aca
                        ciudad[i][j-1].parcial = soldI;
                }
```

```
                        // recursion
                        res = recorridos(ciudad, cola, soldI, pos_aux, bunker,cont,tope);
                        if(res){ return res; }
                }
        }

        if(ciudad[i][j].arriba > soldados || ciudad[i][j].abajo > soldados ||
                ciudad[i][j].izquierda > soldados || ciudad[i][j].derecha > soldados)
        {
                bp = make_pair(posicion,soldados);
                cola.push_back(bp);
                cont++;
        }

        return false;
}


void zombieland(Mapa& ciudad, list<pair <pos,int> >& cola, int& soldados, pos bunker)
{
        int contador;
        int cont_aux = 1;
        int sold_aux;
        int tope = soldados;
        bool res = false;
        pos posicion;

        while(tope > 0 && !res) // todavia tengo soldados y no encontre solucion
        {
                contador = cont_aux;    // cant de elementos a mirar en la cola
                cont_aux = 0;

                while(contador > 0 && !res)     // todavia tengo elementos para ver y no
                    encontre solucion
                {
                        posicion = (cola.front()).first;
                        sold_aux = (cola.front()).second;
                        cola.pop_front();
                        contador--;
                        res = res || recorridos(ciudad, cola, sold_aux, posicion, bunker,
                            cont_aux, tope);
                }

                tope--;
        }

        if(!res){
                soldados = 0; // se mueren todos
        }
        else{
                soldados = tope + 1; // soldados que llegan vivos
        }

        return;
}
```

7.3. Código del Problema 3

```
int plan(Incidencia& inc, int n, list<Arista>& tuberias, int& costo_total,vector<int>& padres)
{
/* Funcion que arma el arbol generador minimo con las tuberias y
 * determina las componentes conexas a la vez.  Devuelve la cantidad de
 * tuberias colocadas.
 */
        int costo;
        int p1;
```

```
        int p2;
        int res = 0;
        Arista edge;
        vector< pair<int,list<int>>> hijos(n);

        for(int i = 0; i < n; i++){     // todos los nodos empiezan con 0 hijos
                hijos[i].first = 0;
        }

        for(Incidencia::iterator it1 = inc.begin(); it1 != inc.end(); it1++)
        {
                edge = it1->first;
                costo = it1->second;
                p1 = padres[edge.first];
                p2 = padres[edge.second];
                if((p1 != p2) || (p1 == -1))
                {
                        if(p1 != p2)    // padres distintos
                        {
                                if((p1 != -1) && (p2 != -1))    // si ninguno es -1
                                {
                                        if(hijos[p1].first < hijos[p2].first){
                                                swap(p1,p2);    // p1 se queda con el que
                                                    tiene mas hijos
                                        }

                                        for(list<int>::iterator it2 = (hijos[p2].second).begin
                                            (); it2 != (hijos[p2].second).end(); it2++)
                                        {       // paso los hijos de p2 como hijos de p1
                                                padres[*it2] = p1;
                                                (hijos[p1].second).push_back(*it2);
                                                hijos[p1].first++;
                                        }
                                        // pongo a p2 como hijo de p1
                                        padres[p2] = p1;
                                        (hijos[p1].second).push_back(p2);
                                        hijos[p1].first++;

                                }else if(p1 == -1){     // si p1 es -1
                                        // pongo a p1 como hijo de p2
                                        padres[edge.first] = p2;
                                        (hijos[p2].second).push_back(edge.first);
                                        hijos[p2].first++;
                                }else{  // si p2 es -1
                                        // pongo a p2 como hijo de p1
                                        padres[edge.second] = p1;
                                        (hijos[p1].second).push_back(edge.second);
                                        hijos[p1].first++;
                                }
                        }else if (p1 == -1){    // si ambos son -1
                                // pongo como padre de ambos a p1
                                padres[edge.first] = edge.first;
                                padres[edge.second] = edge.first;
                                (hijos[edge.first].second).push_back(edge.second);
                                hijos[edge.first].first++;
                        }

                        // actualizo variables
                        costo_total = costo_total + costo;
                        tuberias.push_back(edge);
                        res++;
                }
        }

        return res;
}
```

```
int petroleo(Incidencia inc,list<int>& refinerias,list<Arista>& tuberias,int n,int& cant_ref,
    int& cant_tub, int costo_ref)
{
/* Funcion que elabora el plan, determina las refinerias a colocar y
 * calcula el costo total.  Devuelve el costo total.
 */
        int p;
        int res = 0;
        vector<int> padres(n,-1);
        vector<bool> aux(n,false);

        // elaboro el plan de tuberias
        cant_tub = plan(inc,n,tuberias,res,padres);

        // determino refinerias
        for(int i = 0; i < n; i++)
        {
                if(padres[i] == -1)     // nodo solitario
                {       // coloco refineria
                        refinerias.push_back(i);
                        cant_ref++;
                }else{
                        p = padres[i];
                        if(!aux[p])     // nueva componente conexa
                        {       // coloco refineria
                                aux[p] = true;
                                refinerias.push_back(i);
                                cant_ref++;
                        }
                }
        }

        // actualizo costo total
        res = res + costo_ref*cant_ref;
        return res;
}
```