Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Tarea 1

IIC2133 - Estructuras de datos y algoritmos

Primer semestre, 2016

Entrega: Miércoles 6 de Abril

Objetivos

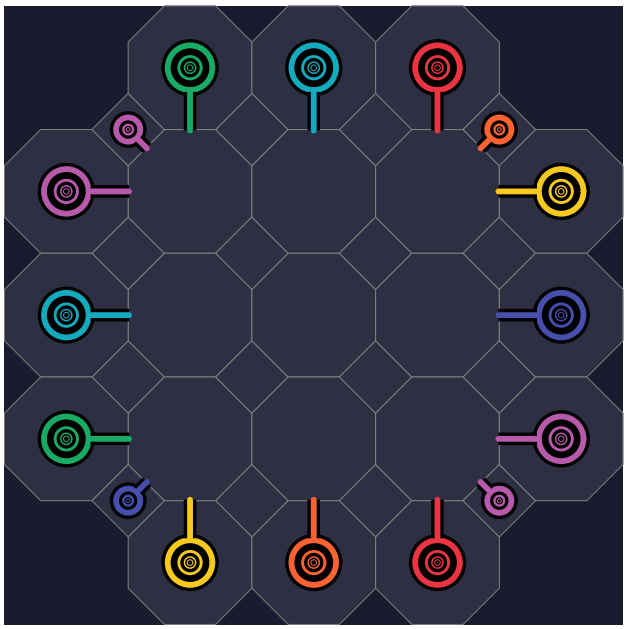
- Usar backtracking para resolver un problema de satisfacción de restricciones.
- Estudiar el desempeño de un algoritmo de backtracking al aplicar diferentes heurísticas y podas.

Introducción: La red de colores

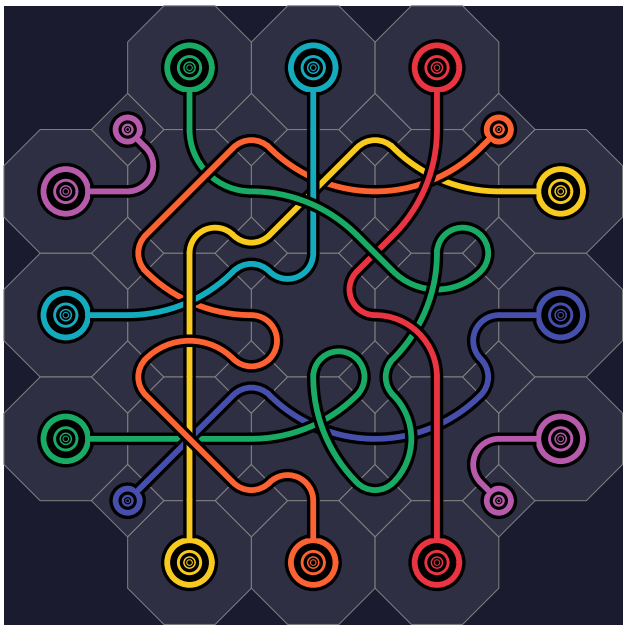
Se desea abastecer por completo una ciudad, para lo cual diferentes compañías se reparten el trabajo. Cada compañía consta de una serie de centrales las cuales sirven como punto de partida o llegada para los repartidores.

Tu deber es trazar la red de rutas que deben seguir los repartidores para cubrir todo el plano de la ciudad, de modo de que dos repartidores no pasen por un mismo lugar, y todas las centrales queden funcionando. Cabe decir que las rutas deben conectar centrales de la misma compañía.

Para eso se te entregará la estructura de la ciudad, que incluirá la posición de cada una de las centrales, junto con la compañía a la cual pertenecen, la cual será representada como un color.



Red de ejemplo



Una posible solución

Problema

Plantearemos lo anterior como un problema de satisfacción de restricciones, por lo que debemos definir los elementos del problema:

Edificio: unidad básica del problema. Puede ser una Central de abastecimiento o un edificio civil, a los cuales llamaremos **cliente**. Las centrales tienen una cantidad de puertos, llamada **capacidad**. Los puertos pueden ser inicio o fin de una ruta, pero no ambos.

Sector: los edificios se encuentran agrupados en **sectores**, los cuales dividen geométricamente el plano de la ciudad. Estos pueden ser octogonales o tetragonales, y contener como máximo **n** edificios, donde **n** es la cantidad de lados que tienen. Cada sector se encuentra conectado con el sector vecino mediante el edificio situado en el límite entre ambos sectores. En caso de que un sector contenga una central de abastecimiento, entonces no contiene más edificios y los llamaremos **núcleo**.

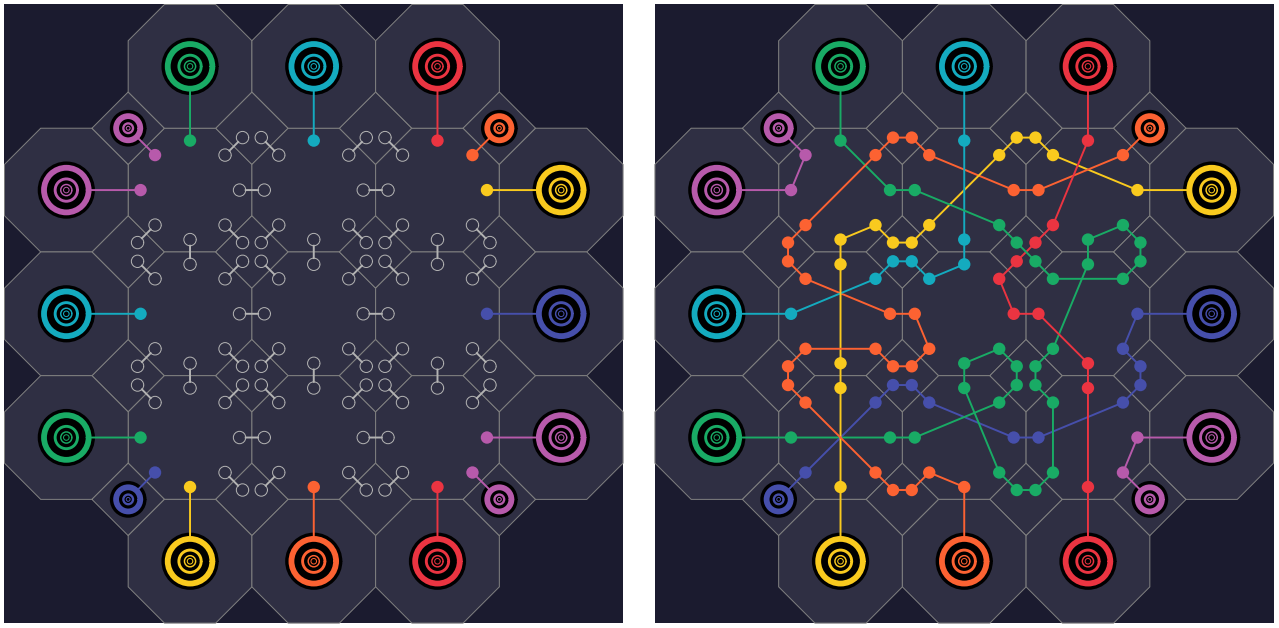
Plano: el plano de la ciudad, contiene la información de los sectores, los cuales a su vez contienen la información de los edificios. Los núcleos son tratados por separado.

Las **restricciones** del problema son las siguientes:

1. Cada cliente debe ser abastecido por alguna de las centrales.
2. Cada cliente debe estar conectado a dos clientes:
   (a) Un cliente de un sector vecino
   (b) Un cliente del mismo sector
3. Si dos clientes están conectados, entonces los atiende el mismo repartidor. Si un cliente está conectado a una central, entonces está abastecido por ésta.

En definitiva:

1. Cada cliente debe de tener un color.
2. Idem
3. Si dos edificios están conectados, entonces tienen el mismo color.



El ejemplo anterior mostrando la estructura detrás del problema. Todas las centrales tienen capacidad 1

Solución

Para resolver este problema, se te proporcionará todo el código en C encargado de la representación y visualización del problema, subido en el repositorio del curso. Deberás completarlo para que el programa sea capaz de definir qué conexiones hacer. Recuerda cumplir con las restricciones mencionadas arriba, considerando además lo siguiente:

- Las conexiones entre edificios son reflejas, es decir, si A está conectado con B, entonces necesariamente B está conectado con A.
- Dado que la restricción 2.(a) está presente siempre, esta conexión ya viene hecha en el plano que se te entregará, por lo que debes decidir únicamente las parejas de edificios que deben ser conectadas para cada sector.

Deberás resolver el problema usando backtracking, diseñando las distintas podas y heurísticas para el problema.

Input & Output

Tu programa se probará en Linux[1] con el siguiente comando:

```
./solver < test.txt
```

La lectura de los archivos de input ya viene resuelta en el código base.

Tu programa deberá imprimir la ciudad, y luego la lista de conexiones que resuelven el problema, sin devoluciones. Para esto deberás usar las funciones provistas por la librería

Análisis

Debes escribir un informe con un breve análisis del problema y el desarrollo de tu solución. Además, debes incluir un análisis de tus resultados.

Esta parte de la tarea es **muy importante**. Debes justificar tus decisiones ("¿Por qué elegiste la heurística h?"). Debes analizar el rendimiento de tus podas y heurísticas ("Se supone que la poda j era muy efectiva. ¿Lo fue finalmente?"), et cétera.

En tu análisis deberás desarrollar los siguientes puntos:

- Una explicación de las distintas podas y heurísticas utilizadas en tu solución, justificando su utilidad.
- Un análisis de rendimiento de las distintas podas y heurísticas, considerando tanto tiempo en resolver el problema como cantidad de veces que tuviste que retroceder.
- En particular, debes incluir al menos:
  - 3 podas
  - 3 heurísticas

Entrega

Deberás entregar en tu repositorio de Github una carpeta de nombre **Tarea01**. Y dentro de esta carpeta, una carpeta **Programa** y una carpeta **Informe**. Dentro de *Programa* deberás tener un *Makefile* y una carpeta **src** con todo el código de tu tarea. Ésta se compilará con el comando `make` dentro del mismo directorio, y tu programa deberá generarse en un archivo de nombre `solver`.

En la carpeta *Informe* deberás tener un archivo de nombre *Informe.pdf*.

Se recogerá el estado de la rama `master` de tu repositorio, 1 minuto pasadas las 24 horas del día de entrega. Recuerda dejar ahí la versión final de tu tarea.

[1] En particular, en un notebook con Ubuntu 14.04 LTS 64-bit y 8GB de RAM.

Evaluación

Se otorgará un 60 % de la nota a que tu programa solucione correctamente los archivos de prueba. El otro 40 % vendrá del informe.

Durante las semanas de la tarea se subirán archivos de prueba para que puedas probar tu solución. Estos están separados en 4 categorías de dificultad creciente:

1. Easy: El problema se puede resolver con backtracking básico
2. Normal: El problema necesita de podas para poder ser resuelto en un tiempo razonable.
3. Hard: Además de lo anterior, se necesitan heurísticas.
4. Lunatic: Además de lo anterior, se necesitará preprocesar el mapa de la ciudad, de otro modo el algoritmo se quedará dando vueltas.

Tu código se evaluará corriendo 3 test de cada categoría, por 0.5 puntos c/u. Solo dos de ellos serán publicados.

Cada test tiene un tiempo límite de 10 segundos. De no responder en ese plazo se corta el programa y se pasa al siguiente test, obteniendo 0 puntos.

Además de esto se subirán un ejemplo o dos de informes para que puedas usar como referencia mientras escribes y revisas tu informe

Bonus

Se otorgará una cierta cantidad de bonus a la nota de tu programa o de tu informe en caso de lograr las siguientes metas opcionales:

1. **Rutas optimizadas**: (20 % al *Programa*) este bonus se trata de reducir el tamaño de las rutas más largas. Se busca que la ruta más larga sea lo más cercana posible al tamaño promedio de las rutas de la red. Obtendrás este bonus si tu programa está entre los 10 que más optimicen.
   Para medir esto definiremos α como el cociente entre $l^\star$ y L, donde $l^\star$ es el tamaño promedio de las rutas[2] y L es el tamaño de la ruta más larga. Mientras más cercano a 1 sea α, mejor es la red producida.
2. **No al monopolio**: (20 % al *Programa*) este bonus se trata de balancear la competencia entre las compañías. Se busca que la participación de la compañía que más edificios atiende y la participación de la que menos atiende sean lo más similares posible. Obtendrás este bonus si tu programa está entre los 10 que más reduzcan el monopolio.
   Para medir esto definiremos β como el cociente entre n y N, donde n es el número de edificios que atiende la compañía con menos clientes en la ciudad, y N es el número de edificios que atiende la compañía con más clientes. Mientras más cercano a 1 sea β, mejor es la red producida.
3. **Ortografía perfecta**: (5 % al *Informe*) por tener 0 fallas de ortografía, obtendrás un 5 % de bonus en la nota de tu informe.
4. **Buen uso del espacio y del formato**: (5 % al *Informe*) a juicio del corrector, si tu informe está bien presentado y usa el espacio y formato a favor de entregar la información, obtendrás un 5 % de bonus en tu informe.

Como esta es la primera tarea y tu primer informe será probablemente el más difícil, para esta entrega los bonus de informe se **duplicarán**. Por lo tanto, de obtener (por ejemplo) *Ortografía perfecta*, el bonus será 10 % en lugar de 5 %.

Restricciones y Alcances

Eres libre de modificar el código de la tarea, siempre que no interfieras con el código de la librería. Agregar atributos a los struct es algo que se espera que hagas. Sin embargo, recuerda que el input y output están **estandarizados**: si tu tarea funciona sólo con un input alternativo o entrega un output distinto del que debiera, tu tarea no se evaluará.
.

[2]Es importante destacar que $l^\star$ será constante para cada red, dado que el número de nodos y el de conexiones totales es independiente de la red que se construya.

Anexo

A continuación se presentan puntos que no son parte del enunciado, pero que te servirán quizás como referencia o herramienta para trabajar con tu tarea.

Watcher

En las ayudantías 1 y 2 se trabajó con problemas de backtracking. Para visualizarlas, se usó un programa llamado `Watcher`. Watcher viene dentro del directorio SRC de la tarea. No es necesario que lo modifiques pues viene completo, ni que lo compiles por separado, pues el Makefile incluido lo compilará automáticamente. Para usarlo es igual que para el *Cindy's Puzzle* o el *Sudoku*: sólo ejecuta alguno de los siguientes comandos.

Uso común:

```
./solver < test.txt | ./watcher.exe
```

Para modificar el delay entre cada imagen (*frame*) que muestra el Watcher: (por ejemplo, a 200 mseg)

```
./solver < test.txt | ./watcher.exe 200
```

Para que el Watcher muestre la estructura subyacente al problema:

```
./solver < test.txt | ./watcher.exe -u
```

Ambos parámetros se pueden poner, en cualquier orden:

```
./solver < test.txt | ./watcher.exe 200 -u
./solver < test.txt | ./watcher.exe -u 200
```

En el momento que cierres el Watcher, éste guardará una imagen vectorial (en watcher.pdf). El archivo mostrará el estado de la red que tenía el Watcher antes de cerrarlo. Puedes usar estas imágenes en tu informe.

Ten cuidado ya que el watcher construye los estados según las instrucciones que le imprimas, por lo que lo que verás en la ventana no necesariamente será el estado actual de tu puzzle, en particular si no estás propagando correctamente los colores.

Judge

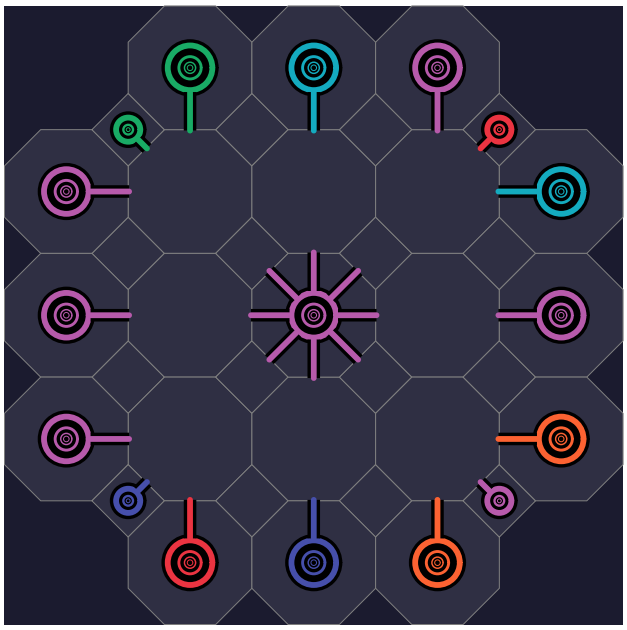
Un nuevo tipo de programa, el `Judge` evalúa el output de tu tarea para determinar si está correcto o no. Este es el programa que se usará para corregir, así que asegurate de probar el output de tu tarea con esto. La forma de compilarlo y usarlo es equivalente al *watcher*:

```
./solver < test.txt | ./judge.exe
```

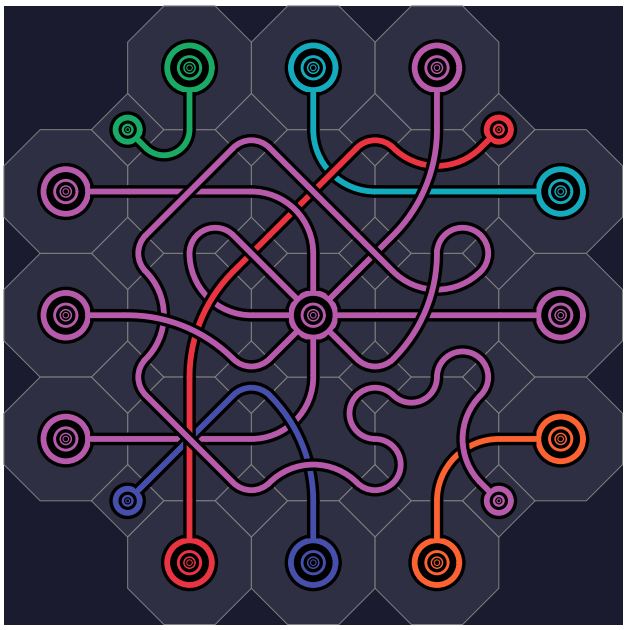
El Judge no recibe parámetros, y en caso de recibir un `undo` entre la lista de decisiones, considerará que falló tu respuesta.

Redes interesantes

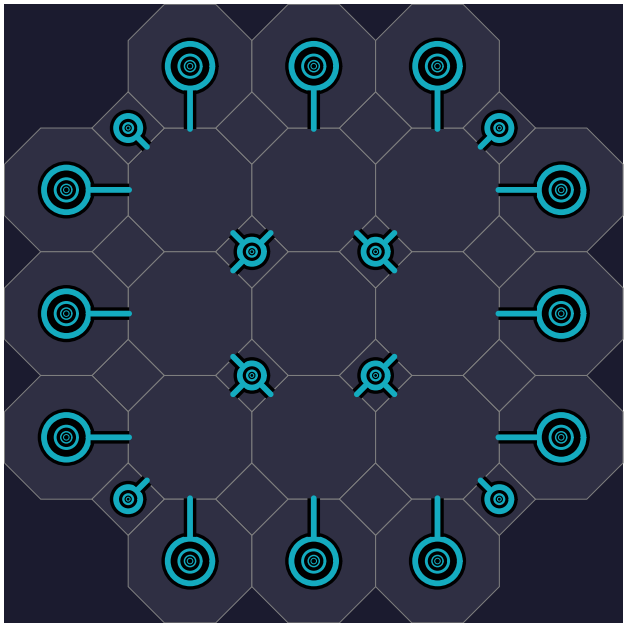
Este problema tiene escenarios muy variados. A continuación te presentamos redes para ciudades con planos que te podría interesar conocer.

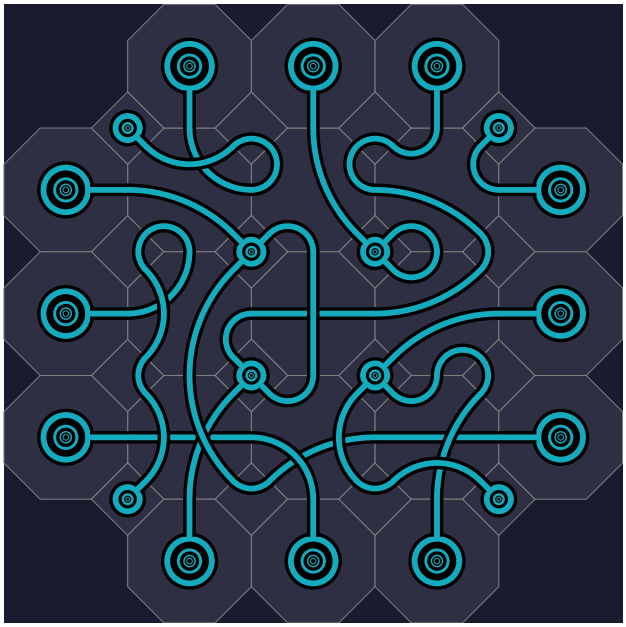


Ciudad que tiene un núcleo en su interior, de capacidad 8

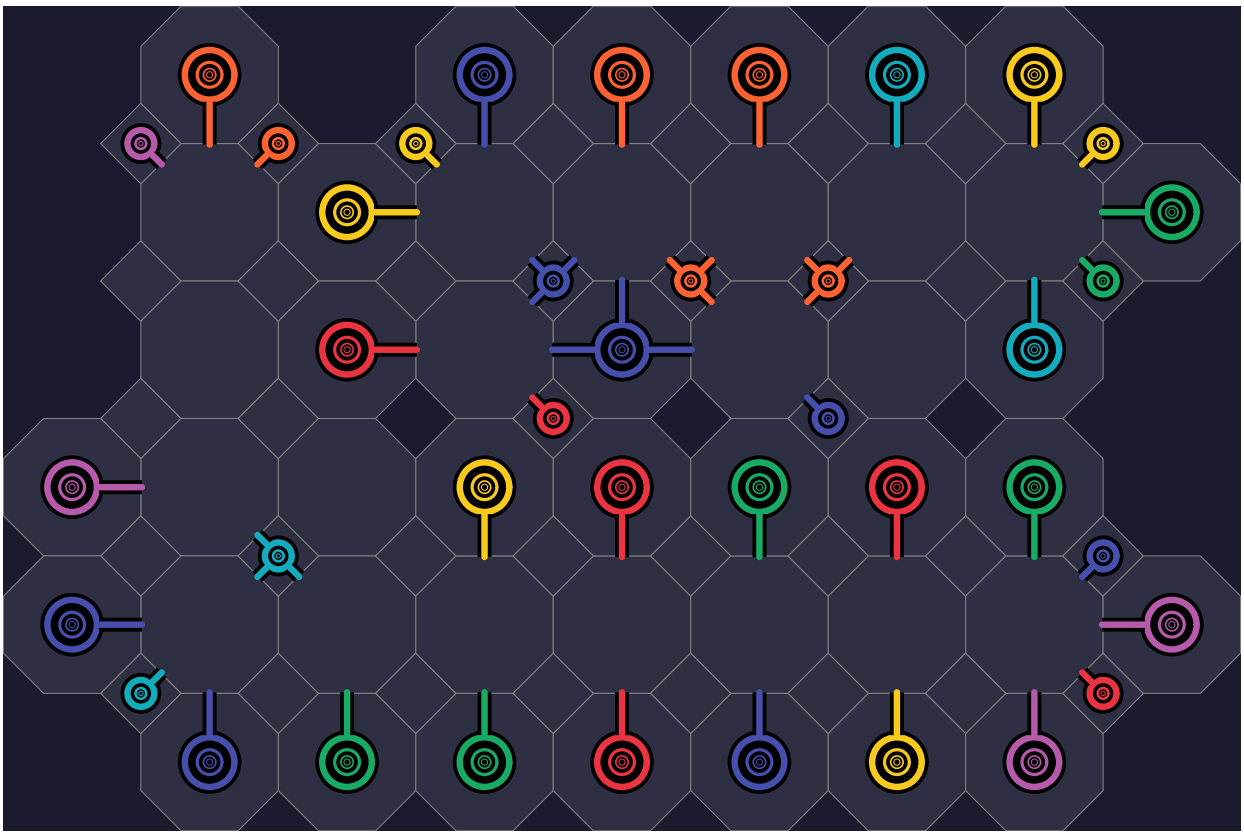


Una posible solución

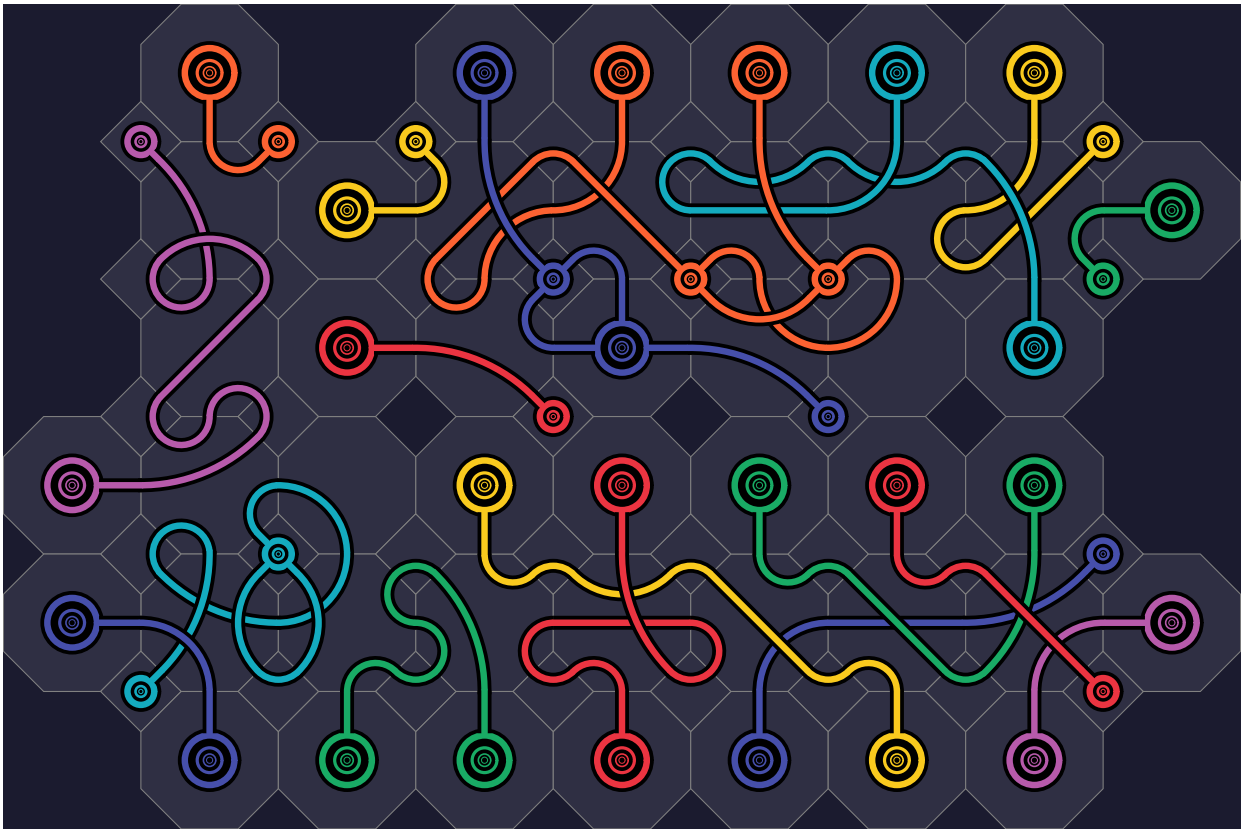


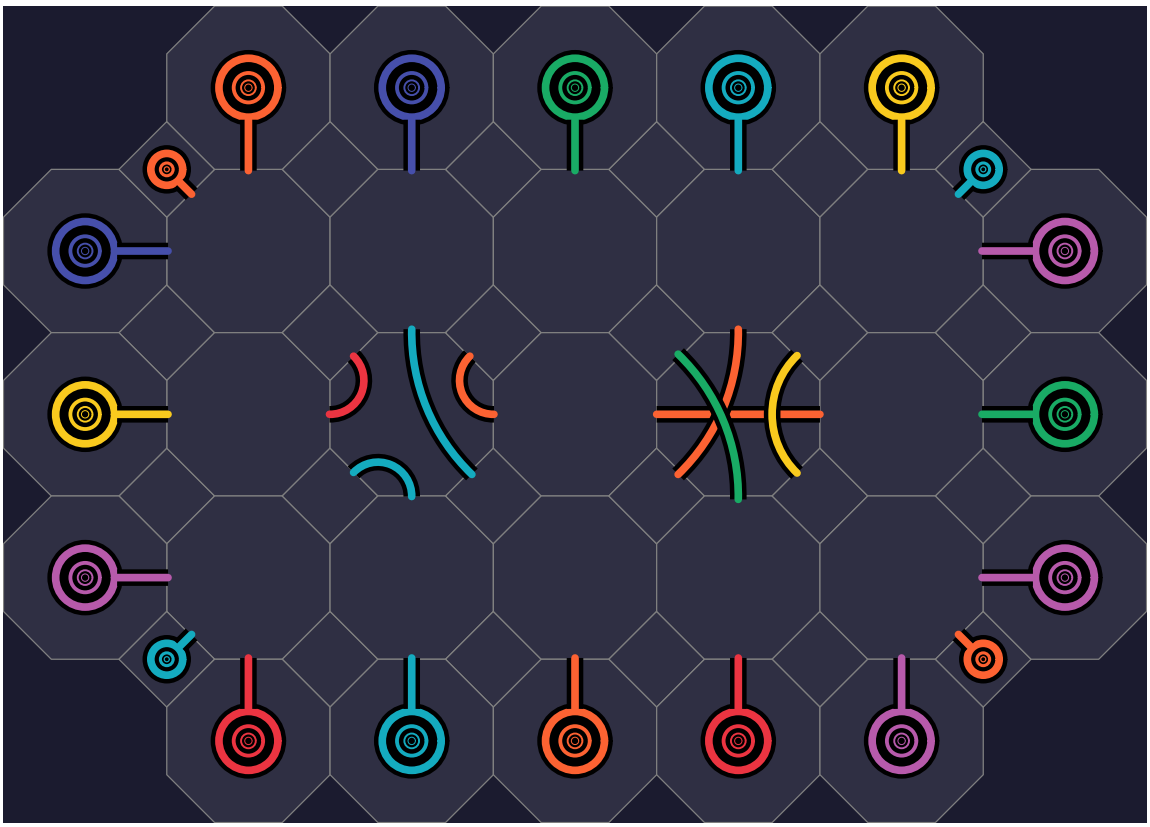


Ciudad con sólo una compañía y núcleos internos de capacidad 3.

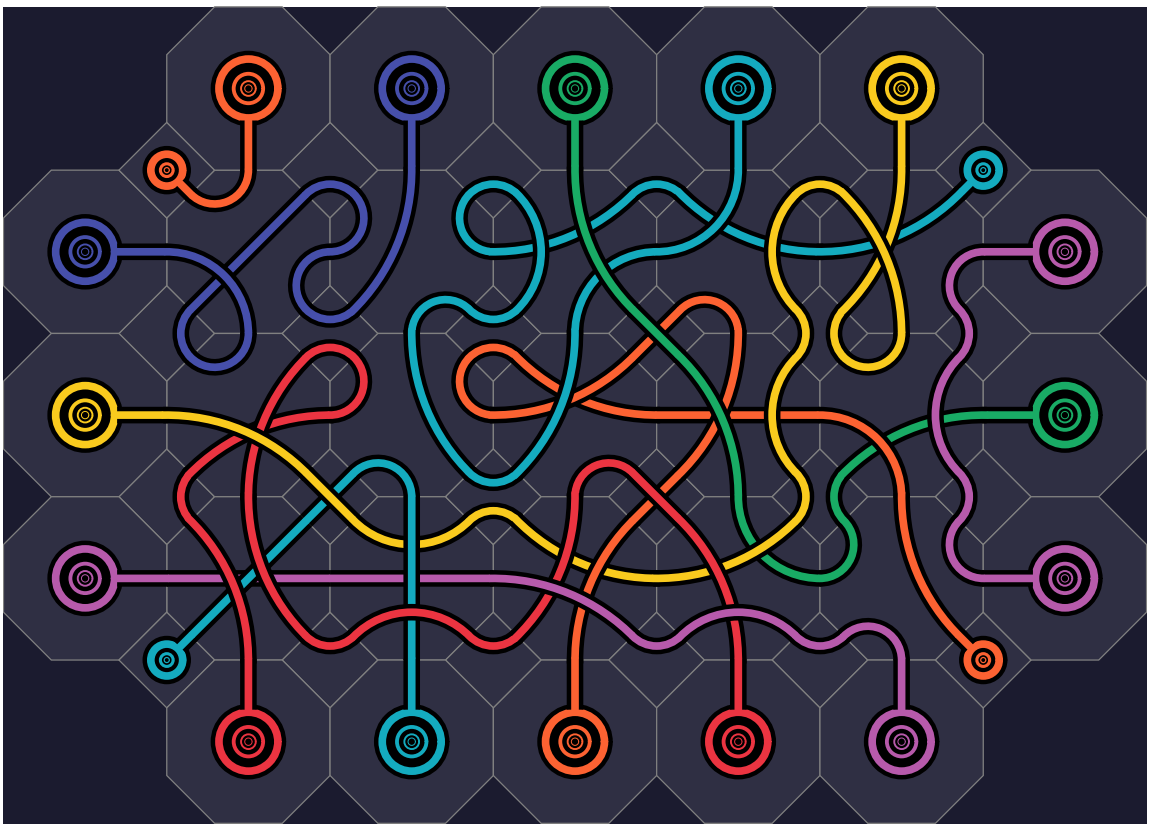


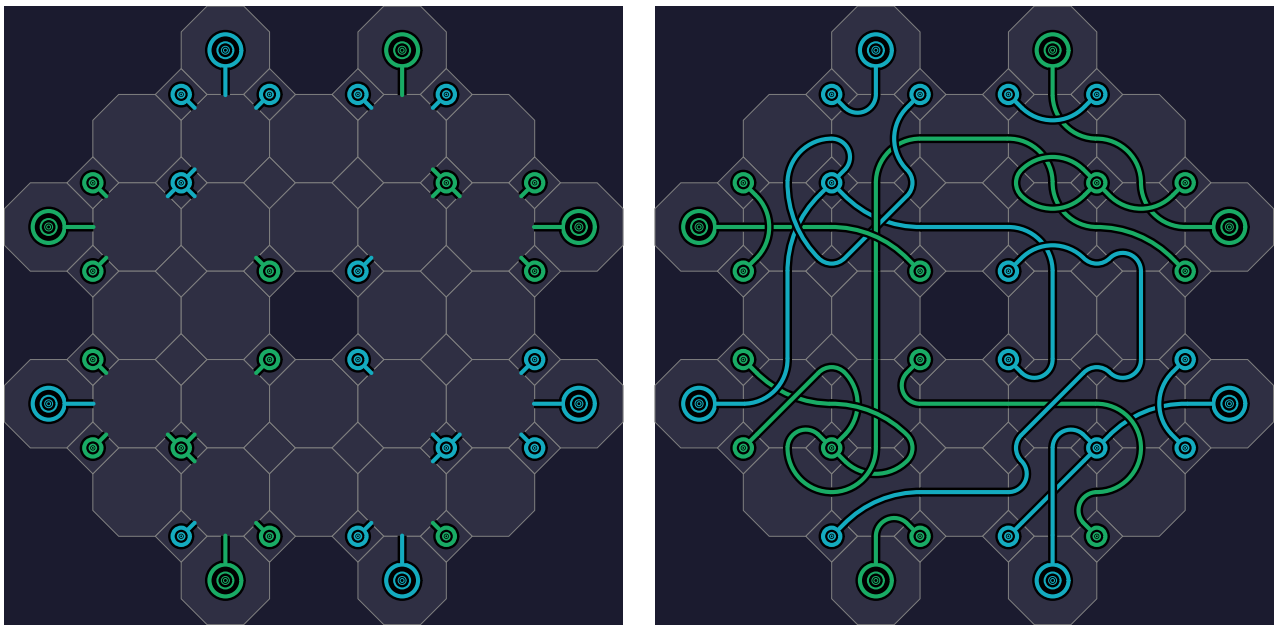
Ciudad que tiene secciones separadas, y una posible solución.





Ciudad en la que las redes ya están parcialmente trazadas





Ciudad con una geometría interesante.