Capítulo 1

El Problema del Agente Viajero

El Problema del Agente Viajero, o TSP por sus siglas en inglés (*Traveler Salesman Problem*), se define como sigue: dado un subconjunto de ciudades en un mapa, ¿cuál es el camino más corto que pasa por todas ellas (si acaso existe)?

1.1. La gráfica

Las ciudades en el mapa y sus conexiones estarán representadas por una gráfica ponderada $G(E, V)$, $E \subset V \times V$; con una función de peso para las aristas $w : E \longrightarrow \mathbb{R}^+$ que representará la distancia entre cada par de ciudades. La gráfica, aunque muy densa, no será completa.

Sea $S \subset V$; S será la instancia de TSP que se querrá resolver. Para hacer más sencilla la optimización de este problema, se utilizará la gráfica completa $G_S(V_S, E_S)$, donde $V_S = S$ y $E_S = \{(u, v) | u, v \in S \text{ y } u \neq v\}$, con la función de peso aumentada $w_S : E_S \longrightarrow \mathbb{R}^+$ que se define como sigue:

Definición 1.1.1 (Función de peso aumentada) *Sea* $w_S : E_S \longrightarrow \mathbb{R}^+$ *tal que:*

$$w_S(u, v) = \begin{cases} w(u, v) & \text{si } (u, v) \in E \\ d(u, v) \times \text{máx}_d(S) & \text{en otro caso} \end{cases}$$

donde $d(u, v)$ *es la* distancia natural *entre dos vértices* u *y* v*; y* $\text{máx}_d(S)$ *será la* distancia máxima *de* S*.*

La razón para definir w_S de esta manera se explicará en la sección 1.3.

1.1.1. Distancia máxima

La distancia máxima de S se define como:

Definición 1.1.2 (Distancia máxima de S) *La* distancia máxima *de S, denotada por* $\text{máx}_d(S)$*, es:*

$$\text{máx}_d(S) = \text{máx}\{w(u,v) | u, v \in S \text{ y } (u,v) \in E\}.$$

En otras palabras, $\text{máx}_d(S)$ es la distancia máxima que existe entre pares de elementos de S tales que estén conectados en G.

1.1.2. Distancia natural

Para calcular la distancia natural entre dos elementos de S, se utilizarán las coordenadas (en latitud y longitud) de las ciudades correspondientes y la fórmula que supone que el planeta Tierra es una esferea perfecta con un radio R de $6,373,000$ metros.

Las coordenadas estarán representadas en grados; la fórmula utiliza radianes, por lo que es necesario convertir grados a radianes con la ecuación 1.1.

$$rad(g) = \frac{g\pi}{180}. \tag{1.1}$$

Definición 1.1.3 (Distancia natural) *La* distancia natural *entre u y v, denotada por $d(u,v)$, está definida por:*

$$d(u,v) = R \times C,$$

donde R es el radio del planeta Tierra en metros (aproximadamente 6,373,000); C está dada por:

$$C = 2 \times \arctan(\sqrt{A}, \sqrt{1-A}),$$

y A está definida por la ecuación 1.2:

$$A = \sin\left(\frac{lat(v) - lat(u)}{2}\right)^2 + \cos(lat(u)) \times \cos(lat(v)) \times \sin\left(\frac{lon(v) - lon(u)}{2}\right)^2. \tag{1.2}$$

La distancia natural no es exactamente la distancia que existe entre dos pares de coordenadas; como la Tierra no es una esfera perfecta, se puede dar un error de unos cuantos kilómetros. Ignoraremos esto para nuestros ejemplos.

1.2. Soluciones

Dada una instancia S de TSP, cualquier permutación de los elementos de S es una solución al problema en G_S, porque G_S es una gráfica completa. Si $S = \{v_1, v_2, v_3\}$, entonces todas las posibles soluciones de S son:

$$\begin{array}{lll} v_1 & v_2 & v_3 \\ v_1 & v_3 & v_2 \\ v_2 & v_1 & v_3 \\ v_2 & v_3 & v_1 \\ v_3 & v_1 & v_2 \\ v_3 & v_2 & v_1 \end{array}$$

Por supuesto v_1, v_2, v_3 es una solución equivalente a v_3, v_2, v_1; esto sin embargo será irrelevante para el sistema.

Aunque todas las permutaciones de S son soluciones válidas en la gráfica G_S, esto no es cierto para G, que es donde realmente se desea resolver el problema. Clasificaremos entonces a todas las permutaciones de S en dos tipos distintos: factibles y no factibles.

Definición 1.2.1 (Solución factible) *Sea $S\{v_1, \ldots, v_k\}$ una instancia de TSP y $P = v_{\rho(1)}, v_{\rho(2)}, \ldots, v_{\rho(k)}$ una permutación de los elementos de S. P será una solución* factible *si y sólo si $(v_{\rho(i-1)}, v_{\rho(i)}) \in E$, para $i = 2, \ldots, k$.*

En otras palabras, una permutación será factible si y sólo si las aristas entre dos elementos consecutivos de la permutación existen en E (siempre existen en E_S, porque G_S es completa).

Si al menos una arista entre dos elementos consecutivos de la permutación no existe en E, la solución será no factible.

1.3. Función de costo

La optimización de TSP necesitará una función de costo que se usará para evaluar qué tan buenas (o no) son las soluciones que el sistema encuentre. Como se quiere minimizar la distancia de la trayectoria que pase por todos los vértices, definimos w_s en la sección 1.1 de tal forma que las aristas que estén en E_S, pero no en E, tengan un peso *mucho mayor* que las aristas en E. Esto alentará al sistema a tratar de descartar aristas que no estén en G para entonces encontrar soluciones factibles.

Para poder comparar, aunque sea burdamente, las evaluaciones a soluciones factibles de instancias de TSP diferentes, vamos a normalizar la función de costo. Para esto necesitaremos un normalizador.

Definición 1.3.1 (Normalizador) *Para cada par no ordenado $u, v \in S$, si $(u, v) \in E$, agregamos la distancia $w(u, v)$ a una lista L y ordenamos L de mayor a menor. Sea $L' = L$ si la longitud de L es menor que $|S| - 1$; o la sublista de L con sus primeros $|S| - 1$ elementos en otro caso.*

El normalizador *de S (denotado por $\mathcal{N}(S)$) está definido como:*

$$\mathcal{N}(S) = \sum_{d \in L'} d.$$

En otras palabras, y suponiendo que $|S| = k$, el normalizador de S es la suma de las $k-1$ aristas más pesadas en E formadas de elementos de S.

Toda permutación de ciudades en S tal que sea una solución factible de TSP tendrá, por definición, una longitud menor o igual a $\mathcal{N}(S)$. El normalizador nos permitirá, como su nombre indica, normalizar nuestra función de costo de tal manera que todas las soluciones factibles de S se evalúen entre 0 y 1; y que todas las soluciones no factibles se evalúen con un valor mayor a 1.

Con el normalizador y la función de peso aumentada podemos definir la función de costo:

Definición 1.3.2 (Función de costo) *Sea $S \subset V$ una instancia de TSP; la función de costo f de una permutación $P = v_{\rho(1)}, \ldots, v_{\rho(k)}$ de los elementos de S se define como:*

$$f(P) = \frac{\sum_{i=2}^{k} w_S(v_{\rho(i-1)}, v_{\rho(i)})}{\mathcal{N}(S)}.$$

Ésta es la función que se buscará optimizar.

1.4. Vecinos

Además de la función de costo, las optimizaciones que se cubrirán en este libro suelen utilizar la estrategia de *búsqueda local* (*local searching*), lo cual implica conocer el *vecindario* de una solución. Para esto se necesita definir el *vecino* de una solución de TSP.

Definición 1.4.1 (Solución vecina) *Sea S una instancia de TSP y $P = v_{\rho(1)}, \ldots, v_{\rho(k)}$ y $P' = v_{\sigma(1)}, \ldots, v_{\sigma(k)}$ dos permutaciones de los elementos de S.*

Diremos que P y P' son vecinas si y sólo si existen dos índices $1 \leq s, t \leq k$, $s \neq t$, tales que:

- $v_{\rho(i)} = v_{\sigma(i)}$ *si* $i \notin \{s, t\}$, *y*
- $v_{\rho(s)} = v_{\sigma(t)}$ *y* $v_{\rho(t)} = v_{\sigma(s)}$.

En otras palabras, dos soluciones serán vecinas si intercambiando dos elementos de la primera se obtiene la segunda.

Capítulo 2

Aceptación por Umbrales

La heurística de Recocido Simulado (*Simulated Annealing*) fue propuesta por Scott Kirkpatrick, Daniel Gelatt y Mario Vecchi en 1983. La idea de la heurística es emular la técnica metalúrgica de *recocido* que se utiliza para reducir los defectos en metales.

Gunter Dueck y Tobias Scheuer crearon en 1990 la variante del Recocido Simulado que denominaron Aceptación por Umbrales (*Threshold Accepting*), que en general se considera más sencilla de implementar.

La heurística de Aceptación por Umbrales será la que se verá en este capítulo.

2.1. Preliminares

Dado un problema $\mathscr{P}$ de optimización y clasificado como *NP*-duro, sea S el conjunto de posibles soluciones a una instancia de $\mathscr{P}$. Se supondrá que se tiene una función $f : S \to \mathbb{R}^+$ (llamada *función objetivo*), tal que $0 \leq f(s) < \infty$ para cualquier $s \in S$. Dadas $s, s' \in S$, si $f(s) < f(s')$, entonces se considerará a la solución s mejor que a la solución s'.

Si, de alguna manera bien definida, se puede convertir una solución s en una solución distinta s' (y obviamente la operación inversa convierte s' en s), se considerarán a s y s' como soluciones *vecinas*. Esta relación determina una gráfica, que *no necesariamente* es conexa.

La idea central de la aceptación por umbrales es, dada una temperatura inicial $T \in \mathbb{R}^+$ y una solución inicial s (ambas obtenidas de alguna manera), de forma aleatoria buscar una solución vecina s' tal que $f(s') \leq f(s) + T$, y entonces actualizar s para que sea s'; en este caso diremos que la solución s' es *aceptada*. Se continúa de esta manera mientras la temperatura T es disminuida paulatinamente siguiendo una serie de condiciones; el proceso termina cuando $T < \varepsilon$ (para $\varepsilon > 0$ muy pequeña); cuando se han generado un determinado número de soluciones aceptadas; o cuando otra serie de condiciones es satisfecha.

Los algoritmos no deterministas de la heurística serán analizados en la siguiente sección.

2.2. La heurística

Como se mencionaba arriba, la heurística busca soluciones mejores a la solución actual, o peores dentro de un umbral determinado por la temperatura. Dicha temperatura va decrementando su valor, y cuando sea menor a ε, la heurística termina.

Las condiciones para decrementar la temperatura T estarán dadas por el comportamiento de lo que se llamarán lotes; un lote será un número determinado de soluciones aceptadas. Sea $L \in \mathbb{N}^+$, determinado por vías experimentales; el algoritmo (no determinista) para calcular un lote se puede ver en el procedimiento 1.

Procedimiento 1 Calcula lote

```
procedure CALCULALOTE(T, s)                    ▷ Temperatura T, solución s
    c ← 0
    r ← 0.0
    while c < L do
        s' ← VECINO(s)          ▷ VECINO(s) obtiene un vecino aleatorio de s
        if f(s') ≤ f(s) + T then                ▷ Se acepta la solución vecina s'
            s ← s'
            c ← c + 1
            r ← r + f(s')
    return r/L, s   ▷ Promedio de las soluciones aceptadas y última solución aceptada
```

Es necesario notar que el **while** puede no terminar; si VECINO no puede encontrar una solución vecina de s que sea aceptada, entonces el número de soluciones aceptadas puede nunca llegar a L, por lo que la condición de paro puede no alcanzarse. Deben existir medidas que permitan al ciclo terminar si el lote no puede completarse: lo más común es tener un número máximo de intentos, significativamente mayor que L.

Sabiendo calcular lotes, se puede definir el algoritmo principal de la aceptación por umbrales en el procedimiento 2. Como este algoritmo utiliza CALCULALOTE, tampoco es determinista.

El valor ε ($0 \leq \varepsilon < \infty$) es otro cero virtual determinado de antemano. El valor φ ($0 < \varphi < 1$) es el factor de enfriamiento, y determina qué tan lento o rápido la temperatura T va disminuyendo; un valor muy cercano a 1 hará que el sistema se enfríe muy lentamente, lo que permitirá encontrar muy buenas soluciones, pero que causará que la ejecución del mismo sea excesivamente tardada. Inversamente, si φ es muy cercano a 0, el sistema se enfriará muy rápido y la ejecución será igualmente rápida; pero probablemente las soluciones encontradas serán de muy baja calidad.

Procedimiento 2 Aceptación por umbrales

```
procedure AceptacionPorUmbrales(T, s)  ▷ Temperatura T y solución s iniciales
    p ← 0
    while T > ε do
        q ← ∞
        while p ≤ q do                 ▷ Mientras no haya equilibrio térmico
            q ← p
            p, s ← CalculaLote(T, s)
        T ← φT
```

Obviamente, a lo largo de la ejecución del sistema se guarda en una variable global a la solución s mínima, para que esa sea la que se entregue como resultado al finalizar la corrida; en aras de mantener sencillo el pseudocódigo, se omiten las instrucciones que inicializan y actualizan esta variable global.

La calidad del sistema (cuánto tiempo tarda, qué tan buenas son las soluciones que produce) está determinada por todos los parámetros libres del mismo: el tamaño L de lote; el número máximo de veces que se intentará aceptar una solución vecina de la solución actual al tratar de completar un lote; el factor de enfriamiento φ; y el cero virtual ε para la temperatura.

No hay un consenso en la literatura de cómo deben calcularse estos valores; en general se recurre a la experimentación computacional, ejecutando corridas del sistema con distintos valores para los parámetros y analizando los resultados obtenidos.

Otro factor importante es el procedimiento VECINO, que genera una solución vecina **aleatoria** a partir de una solución dada. Dicho vecino debe ser obtenido de manera aleatoria con distribución uniforme, para poder garantizar una búsqueda no sesgada en nuestro espacio de soluciones.

Como se busca tener resultados reproducibles, el generador de números aleatorios (RNG, por sus siglas en inglés) debe ser cuidadosamente mantenido junto con la semilla con la que es inicializado, para poder reproducir los resultados; además de que facilita la paralelización del sistema al poder ejecutar distintas corridas con semillas diferentes.

Por último, no se ha mencionado cómo calcular la solución inicial, ni cómo obtener una temperatura inicial adecuada; de manera similar a φ, la temperatura inicial puede ser muy alta (en cuyo caso la ejecución del sistema tardará demasiado) o muy baja (en cuyo caso las soluciones obtenidas sean probablemente de baja calidad). El escenario inicial es dependiente del problema en cuestión; pero sí existen algoritmos para calcular una temperatura inicial satisfactoria. Esto se cubre en la siguiente sección.

2.3. La temperatura inicial

Como se mencionó arriba, la temperatura inicial afecta en gran medida el comportamiento del sistema; se quiere una temperatura inicial suficientemente alta para evitar caer en un mínimo local muy temprano; pero suficientemente baja para que el sistema no tarde demasiado en terminar.

La temperatura inicial se puede determinar a través de experimentación computacional (como ocurre con todos los parámetros del sistema), pero se cuenta con un algoritmo que ha sido utilizado exitosamente en un sistema de distritación electoral.

La idea del algoritmo es obtener una temperatura T que aumente la probabilidad de que la heurística pueda desplazarse rápidamente por el espacio de búsqueda (al menos al inicio), sin que la misma sea excesivamente grande. Para esto se realizará una búsqueda binaria de una T tal que, con alta probabilidad, acepte un porcentaje P elevado de soluciones vecinas de un escenario inicial, pero que no acepte *todas*. En la distritación electoral un porcentaje $.85 \leq P \leq .95$, es utilizado.

Se puede ver el algoritmo en el procedimiento 3:

Procedimiento 3 Temperatura inicial

```
procedure TEMPERATURAINICIAL(s, T, P)
    p ← PORCENTAJEACEPTADOS(s, T)
    if |P − p| ≤ ε_P then
        return T
    if p < P then
        while p < P do
            T ← 2T
            p ← PORCENTAJEACEPTADOS(s, T)
        T_1 ← T/2
        T_2 ← T
    else
        while p > P do
            T ← T/2
            p ← PORCENTAJEACEPTADOS(s, T)
        T_1 ← T
        T_2 ← 2T
    return BUSQUEDABINARIA(s, T_1, T_2, P)
```

El algoritmo PORCENTAJEACEPTADOS se puede ver en el procedimiento 4, y el algoritmo BUSQUEDABINARIA en el procedimiento 5.

Como el procedimiento 4 una vez más utiliza el algoritmo VECINO, tampoco es determinístico, y tampoco lo es TEMPERATURAINICIAL.

El valor ε_P es otro cero virtual que permite detener el algoritmo más rápido, dado que es muy poco probable que el resultado de PORCENTAJEACEPTADOS sea idéntico a P.

Procedimiento 4 Porcentaje aceptados

```
procedure PORCENTAJEACEPTADOS(s, T)
    c ← 0
    for i = 1 to N do
        s' ← VECINO(s)
        if f(s') ≤ f(s) + T then
            c ← c + 1
            s ← s'
    return c/N
```

Procedimiento 5 Búsqueda binaria

```
procedure BUSQUEDABINARIA(s, T_1, T_2, P)
    T_m ← (T_1 + T_2)/2
    if T_2 − T_1 < ε_P then
        return T_m
    p ← PORCENTAJEACEPTADOS(s, T_m)
    if |P − p| < ε_P then
        return T_m
    if p > P then
        return BUSQUEDABINARIA(s, T_1, T_m, P)
    else
        return BUSQUEDABINARIA(s, T_m, T_2, P)
```

Puede parecer contradictorio que para buscar una temperatura inicial se necesite proveer una temperatura incial; sin embargo, la búsqueda binaria funciona con una complejidad en tiempo (relativa) logarítmica, lo que permite cambiar rápidamente su valor sin importar demasiado cuál es el original, siempre y cuando $0 \leq T \leq 2^k$ para una k muy grande. En la distritación electoral una $T = 8$ funcionó aparentemente bien.