Capítulo 4

Búqueda Local y Variantes

4.1 Introducción

En general existe una gran cantidad de problemas que se consideran difíciles de resolver. La dificultad de los problemas está caracterizada por la teoría de complejidad computacional.

Se sabe que muchos problemas son NP-duros, esto es, el tiempo que requieren para resolver una instancia de ese problema crece en el peor de los casos de manera exponencial con respecto al tamaño de la instancia.

Por lo mismo, muchas veces se tienen que solucionar usando métodos de solución aproximados, que regresan buenas soluciones (inclusive a veces óptimas) a bajo costo computacional.

Las heurísticas se pueden usar para resolver problemas de optimización. Muchas de las estrategias de búsqueda podrían usarse, aunque algunas pueden resultar ser demasiado costosas o quedar atrapadas en mínimos locales.

Las *metaheurísticas* se proponen como métodos, determiníticos o estocásticos para salir de mínimos locales.

Algunas de las propiedades deseables para una metahuerística son:

- Simplicidad: debe de ser simple y basada en un principio claro que pueda ser aplicable en general.
- Precisión: debe de estar formulada en términos matemáticos precisos.
- Coherencia: todos los pasos deben de seguir en forma natural los principios de la metaheurística.
- Eficiencia: debe de tomar un tiempo razonable de tiempo.
- Efectividad: debe de encontrar las soluciones óptimas para la mayoría de los problemas de prueba en donde se conoce su solución.
- Robustes: debe de ser consistente en una amplia variedad de instancias.
- Amigable: debe de ser cláramente expresada, fácil de entender y fácil de usar, y con la menor cantidad de parámetros posibles.
- Inovación: debe de poder atacar nuevos tipos de aplicaciones.

La mayoría de la metaherísticas cumplen con solo algunas de las propiedades arriba mencionadas.

Se han propuesto una gran cantidad de ellas. En este curso veremos las principales.

Por otro lado, no existe un claro ganador ya que en general dependen de la naturaleza de los problemas y los objetivos buscados.

La mayoría de los métodos aproximados son: algoritmos basados en construcción o algoritmos basados en búsqueda local.

Los basados en construcción trabajan con soluciones parciales que tratan de completar, mientras los de búsqueda local se mueven en el espacio de soluciones completas.

Los algoritmos de construcción encuentran soluciones de manera incremental empezando con una solución vacía, añadiendo componentes de forma incremental hasta completar una solución.

Los algoritmos de construcción glotones o *greedy* añaden el componente que de acuerdo a cierta heurística produce el máximo beneficio local.

Tabla 4.1: Algoritmo de Búsqueda Local.

Procedimiento Búsqueda Local
    s = genera una solución inicial
    while s no es ótimo local **do**
        $s' \in N(s)$ con $f(s) < f(s')$
        (solución mejor dentro de la vecindad de s)
        $s \leftarrow s'$
    end
    return s

Por ejemplo en el TSP añadir siempre la ciudad más cercana.

Generalmente los algoritmos que construyen soluciones son más rápidos, pero sus soluciones no siempre son buenas.

4.2 Búsqueda Local (*Local Search*)

Búsqueda local es la base de muchos de los métodos usados en problemas de optimización.

Se puede ver como un proceso iterativo que empieza en una solución y la mejora realizando modificaciones locales.

Básicamente empieza con una solución inicial y busca en su vecindad por una mejor solución. Si la encuentra, reemplaza su solución actual por la nueva y continua con el proceso, hasta que no se pueda mejorar la solución actual (ver tabla 4.1).

Cláramente el diseño de la vecindad es crucial para el desempeño de éste, y de muchos otros, algoritmos.

La vecindad son todas las posibilidades de soluciones que se consideran en cada punto.

El cómo se busca la vecindad y cuál vecino se usa en el reemplazo a veces se

conoce como la regla de pivoteo (*pivoting rule*), que en general puede ser:

- Seleccionar el mejor vecino de todos (*best-improvement rule*).
- Seleccionar el primer vecino que mejora la solución (*first-improvement rule*).

Búsqueda local tiene la ventaja de encontrar soluciones muy rápidamente.

Su principal desventaja es que queda atrapada fácilmente en mínimos locales y su solución final depende fuertemente de la solución inicial.

Búsqueda local es un método determinístico y sin memoria. Dada una misma entrada, regresa la misma salida.

De hecho un conjunto de entradas nos generan la misma salida (mínimo local). Esto se puede ver como un *pozo de atracción*.

Esto sigue siendo muy superior a búsqueda aleatoria y el ingrediente principal es la definición de vecindad que permite moverse de una cierta solución a una mejor solución.

4.2.1 Fast Local Search

Un factor que afecta la eficiencia de los algorimos de búsqueda local es el tamaño de la vecindad. Si se consideran muchos vecinos la búsqueda es muy costosa, especialmente si se requieren muchos pasos antes de llegar a un óptimo local, o si el costo para evaluar la función objetivo es muy alto.

La idea de *fast local search* es ignorar vecinos que probablemente no mejoren la función objetivo.

La vecindad se divide en sub-vecinos a los cuales se les asigna un bit de activación.

La idea es revisar sólo los sub-vecinos cuyo bit de activación sea 1.

Inicialmene todos los bits están activos. Si un sub-vecindario es probado que no tiene movimienos que produzcan alguna mejora, entonces se desactiva.

La división de la vecindad en subvecindades depende del problema. Algo que se puede hacer es revisar cómo se escanean los vecinos y si estos se hacen con *for*s anidados, entonces se pueden juntar los elementos dentro de un loop.

La decisión de si se mejora o no la función objetivo debe de considerar si existe algún cambio en los elementos de la función y no sólo el resultado final.

El principal problema de búsqueda local es que queda atrapado es mínimos locales. Se han propuesto una gran cantidad de algoritmos para aliviar esto. Lo que sigue es una descripción de algunos extensiones a búsqueda local para hacerla más efectiva.

4.2.2 Random Restart y Multistart Methods

La forma más simple para mejorar búsqueda local es repitiendo el proceso varias veces con puntos iniciales aleatorios (*random restart*).

La idea es lograr diversidad y salir de mínimos locales al volver a empezar en otra zona.

Aunque fácil de implementar su efectividad decrece al aumentar la complejidad del problema.

Estudios empíricos y argumentos generales indican que la probabilidad de encontrar una solución de bajo costo con muestreo aleatorio decrece al aumentar el tamaño de búsqueda.

Para los algoritmos de construcción, podemos hablar de estrategias de re-inicio (*multistart*). Estas tienen dos fases: (i) generar (construir) una solución y (ii) tratar de mejorar esa solución (ver tabla 4.2).

Los métodos de re-inicio se pueden sofisticar y en general caracterizar por:

- Uso de memoria: se usa para identificar buenas o malas soluciones anteriores.
- Aleatoriedad: depende de si las soluciones con las que se empiezan se generan aleatoriamente o no.

Tabla 4.2: Algoritmo genérico de re-inicio.

procedimiento MultiStart
while no se satisaface criterio de paro
1) Construye una solución s (generación)
2) Busca una mejor solución s'
Si $s' < s, s \leftarrow s'$
return Mejor Solución

- Grado de reconstrucción: indica los elementos que permanecen fijos entre una generación y la siguiente.

Varias de las técnicas que vamos a ver podemos clasificarlas como técnicas de re-inicio.

4.2.3 Guided Local Search

Búsqueda local guiada (GLS) es una alternativa a búsqueda local para hacerla más efectiva.

La idea básica de GLS es aumentar la función objetivo con penalizaciones.

Dada una función objetivo g que mapea una solución candidata s a un valor numérico, GLS define una función h (que reemplaza a g) y que es usada por búsqueda local de la siguiente forma:

$$h(s) = g(s) + \lambda \times \sum(p_i \times I_i(s))$$

donde s es la solución candidata, λ es un parámetro de GLS, i varía sobre todos los atributos, p_i es la penalización para el i-ésimo atributo (las p_is se inicializan a 0) e I_i indica si s tiene el i-ésimo atributo (ver tabla 4.3):

$$I_i(s) = \begin{cases} 1 & \text{si } s \text{ tiene el } i\text{-ésimo atributo} \\ 0 & \text{de otra forma} \end{cases}$$

Para que búsqueda local pueda salir de óptimos locales, GLS añade penalizaciones a ciertos atributos.

Tabla 4.3: Guided Local Search

$k \leftarrow 0$
$s_0 \leftarrow$ genera una solición inicial
for $i = 1$ until M **do**
    $p_i = 0$
$h(s) = g(s) + \lambda \times \sum(p_i \times I_i(s))$
while criterio de paro **do**
    $s_{k+1} =$ búsqueda local(s_k, h)
    for $i = 1$ until M **do**
        $util_i(s^*) = I_i(s^*) \times \frac{c_i}{1+p_i}$
    for each i tal que $util_i$ es máxima **do**
        $p_i \leftarrow p_i + 1$
    $k \leftarrow k + 1$
return mejor solución encontrada

La utilidad de penalizar un atributo i dado un óptimo local s^* es:

$$util_i(s^*) = I_i(s^*) \times \frac{c_i}{1 + p_i}$$

donde c_i es el costo del atributo y p_i es la penalización actual del atributo i.

El atributo de mayor utilidad es el penalizado (se incrementa su penalización actual).

Veamos primero como prodría aplicarse al TSP antes de ver algunas variantes.

4.2.3.1 Búsqueda local en TSP

Si queremos aplicar búsqueda local a TSP podemos usar como representación un vector de ciudades.

Se puede generar una solución inicial con un tour aleatorio.

En cuanto al esquema de vecindad (y mejora), la mayoría se basa en movimientos $k - Opt$, donde se eliminan k arcos/ligas del tour actual y se reconecta el tour usando k nuevas ligas.

Esta es la base de las heurísticas en búsqueda local más usadas: 2-Opt, 3-Opt y Lin-Kernighan (LK).

4.2.3.2 Búsqueda local guiada en TSP

Para GLS: (i) El costo de la solución es la suma de las distancias de los arcos en el ciclo, (ii) los atributos son todos los posibles arcos, (iii) el costo de cada atributo es su distancia, y (iv) a cada arco se le asocia una penalización (inicialmente = 0).

La nueva función de evaluación es: $D' = D + \lambda P$, donde D es la distancia, λ es un parámetro y P es una matríz de penalización.

Lo que quiere decir es que cada vez que hagamos un cambio, usando algún $k - Opt$, tenemos que calcular la distancia total usando λ y la penalización asociada a cada arco involucrado en el cambio.

$$util(tour, e_{i,j}) = I_{e_{i,j}}(tour) \cdot \frac{d_{i,j}}{1 + p_{i,j}}$$

donde $I_{e_{i,j}}(tour) = 1$ si $e_{i,j} \in tour$ y es 0 si no está.

Lo único que falta definir es el parámetro λ, que generalmente depende del problema que se quiera resolver.

Se ha encontrado que buenos valores de λ son: $\lambda = \alpha \times \frac{g(\text{mínimo local})}{\text{no. atrib. en mín. local})}$.

En ese caso α es más fácil de sintonizar. Para el agente viajero se encontró que un rango de $\frac{1}{8} \leq \alpha \leq \frac{1}{2}$ da buenos resultados.

GLS se puede usar en combinación con otras estrategias.

En particular, la combinación de *fast local search* con GLS es relativamente directa, solo se tienen que asociar atributos con sub-vecindad.

Por ejemplo, en el TSP se podría tener un bit de activación por ciudad, indicando que ciudades conviene considerar en la vecindad.

4.3 Variable Neighbourhood Search

La idea básica es ir cambiando en forma sistemática la vecindad al momento de realizar la búsqueda.

Sea N_k un conjunto finito de vecindades predefinidas y $N_k(x)$ el conjunto de soluciones en la k-ésima vecindad de x.

La mayoría de los algoritmos de búsqueda local usan una sola vecindad.

Las vecindades a veces se pueden inducir a partir de una o más métricas que se tengan en el espacio de solución.

VNS se basa en tres hechos simples:

1. Un mínimo local con respecto a una estructura de vecindad no lo es necesariamente con respecto a otra.
2. Un mínimo global es un mínimo local con respecto a todas las posibles estructuras de vecindades.
3. En muchos problemas el mínimo local con respecto a una o varias estructuras de vecindad están relativamente cerca.

La última observación es empírica e implica que un mínimo local muchas veces nos de información acerca del óptimo global.

Se pueden hacer tres estrategias: (i) determinística, (ii) estocástica, y (iii) combinación de las dos.

Variable neighborhood descent (VND): es un método determinístico. La idea es buscar sistemáticamente en diferentes esquemas de vecindad hasta llegar a un mínimo local que es mínimo con respecto a todos los esquemas de vecindad (ver tabla 4.4).

Tabla 4.4: Algoritmo de búsqueda de vecindad variable decendente.

Sea N_k, para $k = 1, \ldots, k_{max}$ un conjunto de estructuras
de vecindad a seguir en ese orden.
repeat until *no mejoría*
Sea $k \leftarrow 1$
repeat until $k = k_{max}$
encuentra al mejor vecino x' de x ($x' \in N_k(x)$).
si la solución obtenida x' es mejor que la anterior (x),
set $x \leftarrow x'$ y $k \leftarrow 1$
sino $k \leftarrow k + 1$

Reduced VNS (RVNS): es un método aleatorio. Es exactamente igual que la anterior, sólo que los vecinos en cada vecindad son seleccionados aleatoriamente (ver table 4.5).

VNS básico (VNS): combina cambios determinísticos y estocásticos. La idea es seleccionar un punto aleatorio en cierta vecindad y luego aplicarle a ese punto búsqueda local. Como en los casos anteriores, si se encuentra una mejor solución, se re-emplaza la solución actual y se empieza otra vez con el primer esquema de vecindad. Si no se busca una mejora en el siguiente esquema de vecindad (ver tabla 4.6).

Se han propuesto diferentes variantes al algoritmo básico:

- Buscar dentro de los mejores N vecinos.
- Aceptar una mejor solución con cierta probabilidad.
- Una vez que se encontró una solución siguiendo un esquema de vecindad particular, irse a regiones alejadas a tratar de encontrar una mejor solución (*Skewed VNS*).
- Hacer VNS paralelo (PVNS)

En esta, así como en otras técnicas que vamos a hacer, se han propuesto esquemas híbridos, e.g., VNS y búsuqeda Tabú, VNS y GRASP, VNS con satisfacción de restricciones, etc.

Tabla 4.5: Algoritmo de búsqueda de vecindad variable reducida.

Sea N_k, para $k = 1, \ldots, k_{max}$ un conjunto de estructuras
de vecindad a usar en la búsqueda.
Define criterio de paro
repeat until *criterio de paro*
Sea $k \leftarrow 1$
repeat until $k = k_{max}$
genera un punto aleatorio (x') en la k-ésima vecindad de x ($x' \in N_k(x)$).
si la solución obtenida x' es mejor que la anterior (x),
set $x \leftarrow x'$ y $k \leftarrow 1$
sino $k \leftarrow k + 1$

Tabla 4.6: Algoritmo de búsqueda de vecindad variable básico.

Sea N_k, para $k = 1, \ldots, k_{max}$ un conjunto de estructuras
de vecindad a usar en la búsqueda.
Define criterio de paro
repeat until *criterio de paro*
Sea $k \leftarrow 1$
repeat until $k = k_{max}$
genera un punto aleatorio (x') en k-ésima vecindad de x ($x' \in N_k(x)$).
aplica algún método de búsqueda local usando
a x' como punto inicial (sea x'' la solución encontrada)
si la solución obtenida x'' es mejor que la anterior (x),
set $x \leftarrow x''$ y $k \leftarrow 1$
sino $k \leftarrow k + 1$

4.4 Búsqueda Local Iterativa (*Iterated local search*)

Vimos en búsqueda local que podemos definir pozos de atracción.

El problema con búsqueda local es que queda atrapada en mínimos locales y volver a empezar desde varios puntos aleatorios no soluciona el problema.

En realidad lo que se necesita es que se haga un muestreo *sesgado*, guiando la búsqueda de forma más efectiva.

La idea principal de búsqueda local iterativa es tratar de generar vecinos que nos generen pozos de atracción vecinos (o al menos diferentes).

Una posibilidad es generar un camino de vecinos seleccionados aleatoriamente, $s_1, s_2, \ldots, s_i$, donde s_{j+1} es vecino de s_j.

Determinar entonces, el primer vecino que pertenece a otro pozo de atracción. Con esto, podríamos ir generando pozos de atracción vecinos. Sin embargo, es un proceso computacionalmente muy costoso, por la cantidad de llamadas a búsqueda local que se requieren.

En lugar de la noción de vecinos en pozos locales, podemos tomar una noción de vecinos más débil.

Dado un mínimo local, se crea una *perturbación* que pertenezca al espacio de estados legales y que nos genere un nuevo estado.

En ese nuevo estado se aplica búsqueda local. Si el óptimo local encontrado pasa una prueba de aceptación (es mejor), entonces, se acepta, si no se regresa al punto anterior (ver table 4.7).

Búsqueda local iteractiva produce un muestreo con un sesgo adecuado, siempre y cuando las perturbaciones no sean ni muy grandes ni muy pequeñas.

Perturbaciones grandes nos regresan a *random restart*, mientras que perturbaciones pequeñas no nos sacan del pozo de atracción.

Normalmente búsqueda local iteractiva no es reversible (no regresamos a los mismos sub-espacios explorados antes), sin embargo, si se siguen perturba-

Tabla 4.7: Algoritmo de Búsqueda Local Iterativa

procedimiento *Búsqueda Local Iterativa*
s_0 = genera solución inicial
$s*$ = búsqueda local (s_0)
repeat
    s' = perturbación(s^*,historia)
    $s*'$ = búsqueda local (s')
    $s*$ = criterio de aceptación($s*$, $s*'$,historia)
until criterio de terminación

ciones determinísticas se pueden seguir ciclos cortos.

Por lo mismo a las perturbaciones se les deben incluir aspectos aleatorios o ser adaptivas para evitar ciclarse.

El algoritmo de ILS tiene varios componentes que pueden afinarse.

Solución Inicial: puede ser aleatoria o ser la obtenida con un algoritmo *greedy* o glotón (normalmente dá mejores resultados).

Criterio de Aceptación: Si no mantemos historia de lo ocurrido, el criterio de aceptación simplemente considera la diferencia en costos entre $s*$ y $s*'$.

El criterio de aceptación en tal caso puede ser determinístico o se puede ajustar empíricamente conforme se soluciona el problema (por ejemplo, como se usa en recocido simulado).

El criterio de aceptación se usa para establecer un balance entre explotación (intensificación) y exploración (diversificación).

Se favorece la explotación si sólo se aceptan mejores soluciones.

Se favorece la exploración si siempre se hacen perturbaciones independientemente de su costo.

Si después de un cierto tiempo no se han logrado mejores resultados, se pueden empezar ILS desde una nueva solución.

Método de Perturbación: puede contener tanta cantidad dependiente del dominio como se quiera/tenga.

El criterio básico es que el punto inicial obtenido a partir de la perturbación debe de ser un excelente punto inicial para búsqueda local.

La perturbación es el esquema que tiene ILS para escapar de mínimos locales.

La fuerza de la perturbación está definida por el número de cambios realizados sobre la solución.

Por ejemplo en TSP es el número de ligas que se cambian en un tour.

Muchas veces un movimiento aleatorio sobre un nivel superior usado en la estructura de la vecindad es suficiente.

Como no se sabe de antemano cuál puede ser una buena perturbación, se podría tratar de explotar la historia, ya sea en el contexto de Tabu search o de VNS (visto en la sección 4.3).

Velocidad: Se ha visto que el tiempo empleado en k búsquedas locales dentro de búsqueda local iterativa, es mucho menor que hacer k búsquedas locales con re-inicios aleatorios.

Búsqueda Local: El método usado de búsqueda local en general puede ser cambiado por prácticamente cualquiera de los que estamos viendo y vamos a ver.

4.5 Greedy Randomized Adaptive Search (GRASP)

GRASP es un método de multi-inicio (*multi-start*) o iterativo en donde cada iteración tiene dos fases: (i) construcción y (ii) búsqueda local (ver tabla 4.8).

En la fase de construcción se encuentra una solución factible cuya vecindad es investigada hasta llegar a un mínimo local.

La mejor solución es guardada.

Sea el conjunto candidato (C), todos los elementos que se pueden incorporar

Tabla 4.8: Algoritmo de Búsqueda GRASP.

procedimiento GRASP(Max_Itera,Semilla)
for $k = 1, \ldots, Max\_Itera$ **do**
    s = Greedy Randomized Construction(Semilla)
    s = búsqueda local (s)
    actualiza solución(s,Mejor Solución)
return Mejor Solución

Tabla 4.9: Algoritmo de Greedy Ransomized Construction

procedimiento Greedy Randomized Construction(Semilla)
$s \leftarrow 0$
Inicializa el conjunto candidato: $C \leftarrow E$
Evalua el costo incremental $c(e) \forall e \in C$
while $C \neq 0$ **do**
    $c^{min} \leftarrow min\{c(e)|e \in C\}$
    $c^{max} \leftarrow max\{c(e)|e \in C\}$
    $RCL \leftarrow \{e \in C|c(e) \leq c^{min} + \alpha(c^{max} - c^{min})\}$
    selecciona un elemente t de RCL aleatoriamente
    $s \leftarrow s \cup \{t\}$
    actualiza el conjunto candidato C
    re-evalua los costos incrementales $c(e) \forall e \in C$
return s

a la solución parcial que se está construyendo sin que se pierda la factibilidad.

La selección del elemento a incorporar se determina mediante una evaluación *greedy* de todos los elementos candidatos (e.g., incremento en la función de costo al aumentar algún elemento).

Esta evaluación crea una lista de candidados restringida (RCL) formada por los mejores elementos (i.e., los que incrementen menos la función de costo).

Esta lista puede estar restringida en cuanto a la cantidad de elementos (cardinalidad) o en cuanto a la calidad de los elementos.

La calidad se puede restringir de acuerdo la siguiente cota: $c(e) \in \{c^{min}, c^{min} + \alpha(c^{max} - c^{min})\}$

Con $\alpha = 0$ se vuelve un algoritmo completamente *greedy*, mientras que con $\alpha = 1$ se vuelve una estrategia aleatoria.

α sirve para establecer un balance entre costo computacional y calidad de las soluciones.

El elemento que se incorpora en la construcción de la solución, se selecciona aleatoriamente dentro de RCL.

Una vez incorporado se actualiza la lista de candidados y se re-evaluan los costos incrementales.

El algoritmo de búsqueda local puede ser con el mejor candidato (*best-improvement*) o el primer candidato (*first-improvement*) que mejore la solución actual.

Algunas extensiones se ocupan principalmente por como controlar los posibles valores de α.

- Dejar un valor fijo.
- Variar de acuerdo a la calidad de las soluciones, por ejemplo, dentro de un rango de valores y favoreciendo con el tiempo aquellos valores que dan mejores resultados.
- Seleccionar aleatoriamente dentro de un cierto rango.

- Seleccionar aleatoriamente dentro de una distribución decreciente y no uniforme.

También se han hecho extensiones para no seleccionar el elemento de RCL aleatoriamente, sino siguiendo ciertas preferencias de acuerdo al lugar que ocupan.

Por otro lado se han hecho combinaciones o mejoras usando *path relinking*, pero esto lo veremos más adelante.