Optimización y metaheurísticas I

Unidad 4: Metaheurísticas - Optimización Discreta

Dr. Jonás Velasco Álvarez

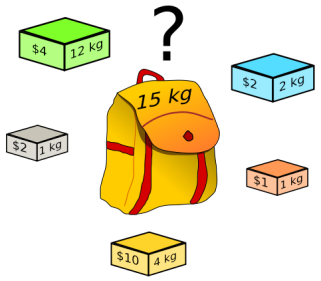
jvelascoa@up.edu.mx

El problema de la mochila

Definición

The knapsack problem. Se tiene un conjunto de n objetos, donde cada objeto j tienen asociado un beneficio p_j y un peso w_j. Por otro lado se tiene una mochila, donde se pueden introducir los objetos, que tiene una capacidad máxima de peso W. Se asume que el peso, el beneficio y la capacidad no son negativos.

El problema consiste en seleccionar un subconjunto de objetos que proporcionen el mayor beneficio, sujeto a la restricción de no exceder la capacidad de la mochila.



Modelo

Variables de decisión

$$x_j = \begin{cases} 1, & \text{si el objeto } j \text{ se introduce en la mochila.} \\ 0, & \text{en otro caso.} \end{cases}$$

donde $j = 1, \ldots, n$.

Restricción de capacidad

$$\sum_{j=1}^{n} w_j x_j \leq W$$

Función objetivo

$$\max\ z = \sum_{j=1}^{n} p_j x_j$$

Aplicaciones

Este tipo de problemas pueden representar diversas situaciones en la vida real tales como:

- operaciones de carga
- selección de proyectos de inversión
- control de presupuestos
- corte de materiales
- diseño de métodos criptográficos

Recocido simulado

Input:
$T, \alpha, K, \epsilon :=$ Parámetros del SA

1: $k \leftarrow 0$
2: Elegir un vector inicial $\mathbf{x}_k$ de manera aleatoria o determinista.
3: **while** $k \leq K \wedge T > \epsilon$ **do**
4: Generar una solución candidata $\hat{\mathbf{x}}$
5: **if** $f(\hat{\mathbf{x}}) > f(\mathbf{x}_k)$ **then**
6: $\mathbf{x}_{k+1} \leftarrow \hat{\mathbf{x}}$
7: **else**
8: $r \leftarrow \mathcal{U}(0,1)$
9: **if** $r < e^{\left(\frac{f(\mathbf{x}_k) - f(\hat{\mathbf{x}})}{T}\right)}$ **then**
10: $\mathbf{x}_{k+1} \leftarrow \hat{\mathbf{x}}$
11: **else**
12: $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k$
13: **end if**
14: **end if**
15: $T \leftarrow \alpha T$
16: $k \leftarrow k + 1$
17: **end while**

Recocido simulado

Input:
$T, \alpha, K, \epsilon :=$ Parámetros del SA

1: $k \leftarrow 0$
2: Elegir un vector inicial $\mathbf{x}_k$ de manera aleatoria o determinista.
3: **while** $k \leq K \wedge T > \epsilon$ **do**
4: Generar una solución candidata $\hat{\mathbf{x}}$
5: **if** $f(\hat{\mathbf{x}}) > f(\mathbf{x}_k)$ **then**
6: $\mathbf{x}_{k+1} \leftarrow \hat{\mathbf{x}}$
7: **else**
8: $r \leftarrow \mathcal{U}(0,1)$
9: **if** $r < e^{\left(\frac{f(\mathbf{x}_k) - f(\hat{\mathbf{x}})}{T}\right)}$ **then**
10: $\mathbf{x}_{k+1} \leftarrow \hat{\mathbf{x}}$
11: **else**
12: $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k$
13: **end if**
14: **end if**
15: $T \leftarrow \alpha T$
16: $k \leftarrow k + 1$
17: **end while**

- Una solución candidata $\hat{\mathbf{x}}$ se obtiene por hacer pequeños cambios a una solución $\mathbf{x}_k$.

Recocido simulado

Input:
$T, \alpha, K, \epsilon :=$ Parámetros del SA

1: $k \leftarrow 0$
2: Elegir un vector inicial $\mathbf{x}_k$ de manera aleatoria o determinista.
3: **while** $k \leq K \wedge T > \epsilon$ **do**
4: Generar una solución candidata $\hat{\mathbf{x}}$
5: **if** $f(\hat{\mathbf{x}}) > f(\mathbf{x}_k)$ **then**
6: $\mathbf{x}_{k+1} \leftarrow \hat{\mathbf{x}}$
7: **else**
8: $r \leftarrow \mathcal{U}(0,1)$
9: **if** $r < e^{\left(\frac{f(\mathbf{x}_k) - f(\hat{\mathbf{x}})}{T}\right)}$ **then**
10: $\mathbf{x}_{k+1} \leftarrow \hat{\mathbf{x}}$
11: **else**
12: $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k$
13: **end if**
14: **end if**
15: $T \leftarrow \alpha T$
16: $k \leftarrow k + 1$
17: **end while**

- Una solución candidata $\hat{\mathbf{x}}$ se obtiene por hacer pequeños cambios a una solución $\mathbf{x}_k$.
- **1-flip**. Dado $\mathbf{x}_k = (0, 1, 0, 0)$, las posibles soluciones candidatas $\hat{\mathbf{x}}$ son: $(0, 0, 0, 0)$, $(1, 1, 0, 0)$, $(0, 1, 1, 0)$, $(0, 1, 0, 1)$.

Búsqueda Tabú

Características básicas

- Es un algoritmo de búsqueda local basado en el uso de memoria.
- Es posible aceptar soluciones que empeoran la solución actual.
- La memoria se utiliza para no repetir la trayectoria de búsqueda.
- Existen dos tipos de memoria: memoria reciente y memoria a largo plazo.
- En la memoria se guardan atributos de soluciones.

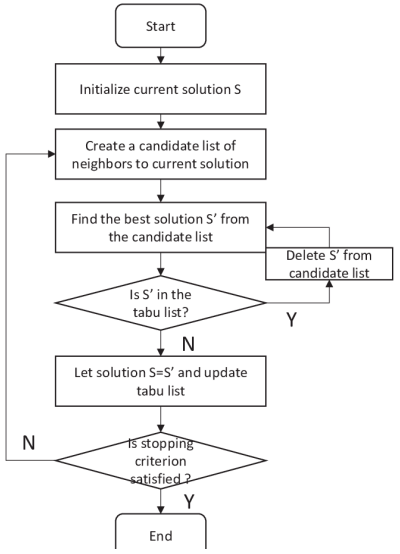
Ideas básicas

- **Movimiento:** Encontrar el valor de la j-ésima variable activa no tabú, tal que $x_j = 0$ y definir $x_j = 1$, siempre que no se viole la restricción mochila. Si no hay un movimiento disponible de este tipo, busque la j-ésima variable activa no tabu más grande de manera que $x_j = 1$ y establezca $x_j = 0$.
- **Atributo tabú:** Es el índice de la variable cuyo valor se cambió en el movimiento más reciente.
- **Regla de activación tabú:** Una variable tabú activa no puede cambiar su valor por ciertas iteraciones de permanencia tabú.
- **Criterio de aspiración:** Remover la variable tabú activa.

Pseudocódigo

1. Seleccionar una solución $\mathbf{x}_{actual} \in \mathcal{F}$ como una solución inicial factible.
2. Seleccionar una nueva solución $\mathbf{x}_{nueva} \in \mathcal{N}(\mathbf{x}_{actual})$
3. Si $f(\mathbf{x}_{nueva}) < f(\mathbf{x}_{actual})$ entonces definir $\mathbf{x}_{actual} \leftarrow \mathbf{x}_{nueva}$
4. Agregar a $\mathbf{x}_{nueva}$ a la lista tabú $\mathcal{T}$
5. Actualizar memoria (por ejemplo, borrar la solución más antigua en $\mathcal{T}$)
6. Repetir los pasos 2 a 5 hasta que un criterio de parada se alcance (por ejemplo, un máximo número de iteraciones)

Pseudocódigo



Problema de la mochila

$$\begin{aligned}
\text{Maximize} \quad & 10x_1 + 14x_2 + 9x_3 + 8x_4 + 7x_5 + 5x_6 + 9x_7 + 3x_8 \\
\text{subject to} \quad & 7x_1 + 12x_2 + 8x_3 + 9x_4 + 8x_5 + 6x_6 + 11x_7 + 5x_8 \leq 38, \\
& x_j = \{0, 1\} \quad \text{for } j = 1, \ldots, 8.
\end{aligned}$$

Table 1. Initial solution.

Variable	Profit	Weight	Ratio	Value	Profit	Weight
1	10	7	1.43	1	10	7
2	14	12	1.17	0	0	0
3	9	8	1.13	0	0	0
4	8	9	0.89	1	8	9
5	7	8	0.88	1	7	8
6	5	6	0.83	1	5	6
7	9	11	0.82	0	0	0
8	3	5	0.60	1	3	5
Total	—	—	—	—	33	35

Problema de la mochila

Table 2. Neighborhood of initial solution.

No.	Move	Neighbor	Profit	Weight	Feasible?
1	$x_1 = 0$	(4, 5, 6, 8)	23	28	Yes
2	$x_2 = 1$	(1, 2, 4, 5, 6, 8)	47	47	No
3	$x_3 = 1$	(1, 3, 4, 5, 6, 8)	42	43	No
4	$x_4 = 0$	(1, 5, 6, 8)	25	26	Yes
5	$x_5 = 0$	(1, 4, 6, 8)	26	27	Yes
6	$x_6 = 0$	(1, 4, 5, 8)	28	29	Yes
7	$x_7 = 1$	(1, 4, 5, 6, 7, 8)	42	46	No
8	$x_8 = 0$	(1, 4, 5, 6)	30	30	Yes

Problema de la mochila

Table 3. Iterations of the STM TS with *TabuTenure* = 2.

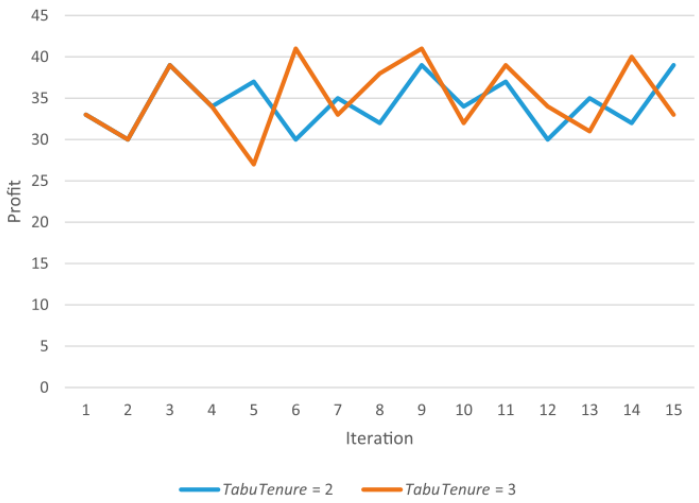
Iteration	Current solution	Profit	Weight	Tabu active	Move
1	(1, 4, 5, 6, 8)	33	35		8
2	(1, 4, 5, 6)	30	30	8	3
3	(1, 3, 4, 5, 6)	**39**	38	3 8	6
4	(1, 3, 4, 5)	34	32	6 3	8
5	(1, 3, 4, 5, 8)	37	37	8 6	5
6	(1, 3, 4, 8)	30	29	5 8	6
7	(1, 3, 4, 6, 8)	35	35	6 5	8
8	(1, 3, 4, 6)	32	30	8 6	5
9	(1, 3, 4, 5, 6)	39	38	5 8	6
10	(1, 3, 4, 5)	34	32	6 5	8
11	(1, 3, 4, 5, 8)	37	37	8 6	5
12	(1, 3, 4, 8)	30	29	5 8	6
13	(1, 3, 4, 6, 8)	35	35	6 5	8
14	(1, 3, 4, 6)	32	30	8 6	5
15	(1, 3, 4, 5, 6)	39	38	5 8	6

Problema de la mochila

Table 4. Iterations of the STM TS with *TabuTenure* = 3.

Iteration	Current solution	Profit	Weight	Tabu active	Move
1	(1, 4, 5, 6, 8)	33	35		8
2	(1, 4, 5, 6)	30	30	8	3
3	(1, 3, 4, 5, 6)	39	38	3 8	6
4	(1, 3, 4, 5)	34	32	6 3 8	5
5	(1, 3, 4)	27	24	5 6 3	2
6	(1, 2, 3, 4)	**41**	36	2 5 6	4
7	(1, 2, 3)	33	27	4 2 5	6
8	(1, 2, 3, 6)	38	33	6 4 2	8
9	(1, 2, 3, 6, 8)	**41**	38	8 6 4	3
10	(1, 2, 6, 8)	32	30	3 8 6	5
11	(1, 2, 5, 6, 8)	39	38	5 3 8	6
12	(1, 2, 5, 8)	34	32	6 5 3	8
13	(1, 2, 5)	31	27	8 6 5	3
14	(1, 2, 3, 5)	40	35	3 8 6	5
15	(1, 2, 3)	33	27	5 3 8	4

Problema de la mochila



Introducción

La palabra **GRASP** es una abreviación de

- **G**reedy
- **R**andomized
- **A**daptive
- **S**earch
- **P**rocedure

se podría traducir al español como: Procedimientos de búsqueda voraces, aleatorizados y adaptativos. GRASP fue inventado por Thomas Feo y Mauricio Resende en 1989. En 1995 adquiere el nombre de GRASP.

Feo & Resende 1989

Operations Research Letters 8 (1989) 67–71
North-Holland

April 1989

A PROBABILISTIC HEURISTIC FOR A COMPUTATIONALLY DIFFICULT SET COVERING PROBLEM *

Thomas A. FEO
The University of Texas, Austin, TX 78712, USA

Mauricio G.C. RESENDE
University of California, Berkeley, CA 94720, USA

Received May 1988
Revised October 1988

An efficient probabilistic set covering heuristic is presented. The heuristic is evaluated on empirically difficult to solve set covering problems that arise from Steiner triple systems. The optimal solution to only a few of these instances is known. The heuristic provides these solutions as well as the best known solutions to all other instances attempted.

algorithms • heuristic • tests • integer programming • set covering

Feo & Resende 1995

Journal of Global Optimization **6:** 109–133, 1995.

Greedy Randomized Adaptive Search Procedures

THOMAS A. FEO
Operations Research Group, Department of Mechanical Engineering, The University of Texas, Austin, TX 78712 U.S.A. (email: feo@emx.utexas.edu)

and

MAURICIO G.C. RESENDE
Mathematical Sciences Research Center, AT&T Bell Laboratories, Murray Hill, NJ 07974 U.S.A. (email: mgcr@research.att.com)

(Received: 29 July 1994; accepted: 14 October 1994)

Abstract. Today, a variety of heuristic approaches are available to the operations research practitioner. One methodology that has a strong intuitive appeal, a prominent empirical track record, and is trivial to efficiently implement on parallel processors is GRASP (Greedy Randomized Adaptive Search Procedures). GRASP is an iterative randomized sampling technique in which each iteration provides a solution to the problem at hand. The incumbent solution over all GRASP iterations is kept as the final result. There are two phases within each GRASP iteration: the first intelligently constructs an initial solution via an adaptive randomized greedy function; the second applies a local search procedure to the constructed solution in hope of finding an improvement. In this paper, we define the various components comprising a GRASP and demonstrate, step by step, how to develop such heuristics for combinatorial optimization problems. Intuitive justifications for the observed empirical behavior of the methodology are discussed. The paper concludes with a brief literature review of GRASP implementations and mentions two industrial applications.

Key words: Combinatorial optimization, search heuristic, GRASP, computer implementation.

El algoritmo

GRASP consiste principalmente de dos fases:

1. La **fase constructiva** (*Greedy Randomized Adaptive*): que consiste en generar una buena solución factible.
2. La **fase de mejora local** (*Local Search*): que consiste en mejorar localmente la solución.

Una vez que se han ejecutado las dos fases, la solución obtenida se almacena y se procede a efectuar una nueva iteración, guardando cada vez la mejor solución que se haya encontrado hasta el momento.

Greedy Randomized Adaptive

La primer fase consiste de tres partes:

1. Algoritmo voraz
   - Siempre toma la mejor elección disponible en cada paso.
2. Función adaptativa
   - Se tiene una función que guía al algoritmo voraz para determinar qué elemento candidato seleccionar para incluir en la solución parcial. Se conoce como **función greedy** y mide la contribución local de cada elemento a la solución parcial.
3. Selección probabilista
   - Seleccionar a los candidatos (con los mejores valores de la función greedy) entre los cuales elegimos el siguiente elemento que se agregará a la solución parcial. Esta se llama **lista restringida de candidatos** (*Restricted Candidate List (RCL)*).
   - El siguiente candidato a ser agregado a la solución se selecciona en forma aleatoria de la RCL.

Restricted Candidate List (RCL)

Dicha lista puede tener

1. Un número fijo de elementos (restricción por cardinalidad)
2. Elementos con los valores de la función greedy dentro de un rango dado.
   - $C_{\text{min}} \leq C(\text{candidatos}) \leq C_{\text{min}} + \alpha(C_{\text{max}} - C_{\text{min}})$
   - Parámetro de calidad: $0 \leq \alpha \leq 1$, donde se pueden aceptar no sólo al mejor, sino también a los valores más cercanos a dicho valor mínimo.
   - $\alpha = 0$ construcción puramente miope
   - $\alpha = 1$ construcción puramente aleatoria
   - El voraz aleatorizado es un equilibrio entre calidad y diversidad.

El parámetro α

- Diseño básico
  - Valor de α constante.
  - RCL de tamaño constante.
- Mejoras con α variable
  - Reducir el valor de α si tras un cierto número de iteraciones no se ha mejorado la solución.
  - Seleccionar α al azar en el intervalo $[0, 1]$.
- GRASP reactivo
  - En cada iteración GRASP, se elige un valor de α, de entre un conjunto discreto de valores $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$
  - La probabilidad de que α_j sea elegida es $p(\alpha_j)$
  - Modificar las probabilidades para favorecer valores que produzcan una buena solución.

Pseudocódigo GRASP

```
procedure grasp()
1    InputInstance();
2    for GRASP stopping criterion not satisfied  →
3        ConstructGreedyRandomizedSolution(Solution);
4        LocalSearch(Solution);
5        UpdateSolution(Solution,BestSolutionFound);
6    rof;
7    return(BestSolutionFound)
end grasp;
```

Fig. 1. A generic GRASP pseudo-code

Pseudocódigo fase constructiva

```
procedure ConstructGreedyRandomizedSolution(Solution)
1    Solution = {};
2    for Solution construction not done  →
3        MakeRCL(RCL);
4        s = SelectElementAtRandom(RCL);
5        Solution = Solution ∪ {s};
6        AdaptGreedyFunction(s);
7    rof;
end ConstructGreedyRandomizedSolution;
```

Fig. 2. GRASP construction phase pseudo-code.

Pseudocódigo fase mejora local

```
procedure local(P,N(P),s)
1    for s not locally optimal →
2        Find a better solution t ∈ N(s);
3        Let s = t;
4    rof;
5    return(s as local optimal for P)
end local;
```

Fig. 3. GRASP local search phase.

Búsqueda local

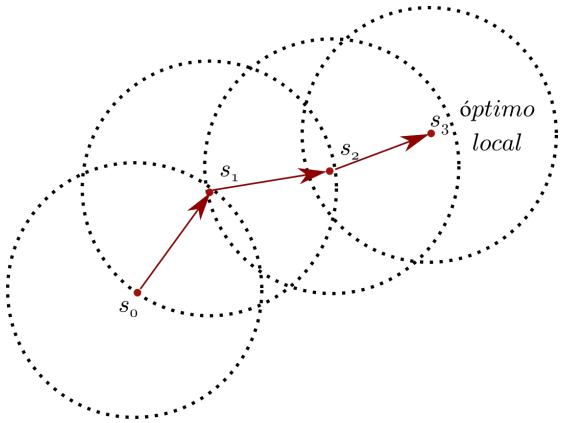
Input:
$s :=$ an initial feasible solution.
Output: $s :=$ a local optimum solution with respect to a neighborhood function $\mathcal{N}$ and an objective function f.

1: **while** $\mathcal{N}(s)$ contains a better solution than s **do**
2: Choose a solution $s' \in \mathcal{N}(s)$ with better value $f(s')$ than $f(s)$.
3: Set $s \leftarrow s'$
4: **end while**
5: **return** s

Algorithm 1: Local search(s)

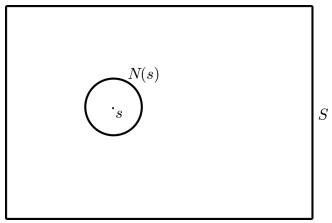
Búsqueda local

La búsqueda local comienza desde una solución inicial s_0 y, de manera iterativa, se mueve a una solución vecina hasta alcanzar un óptimo local.



Función de vecindad

Suponga un espacio de solución S. Una **estructura de vecindad** es un mapeo $\mathcal{N} : S \rightarrow 2^S$ que define para cada $s \in S$ un conjunto $\mathcal{N}(s) \subseteq S$ de soluciones que se encuentran en la vecindad de s. El conjunto $\mathcal{N}(s)$ se llama **vecindario** de s y cada $s^{'} \in \mathcal{N}(s)$ se llama **vecino** de s.

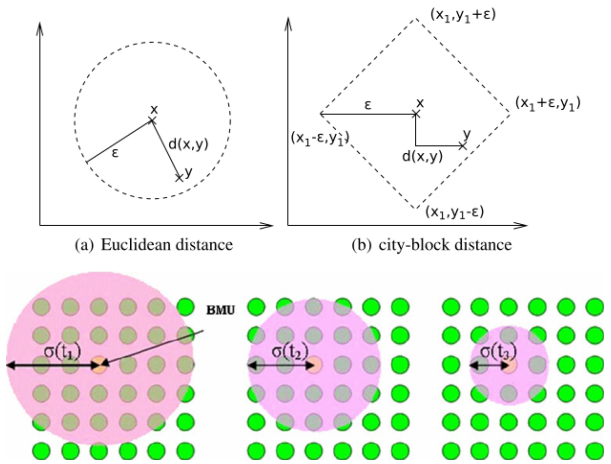


Función de vecindad

Una función de vecindad es un mecanismo que cambian un atributo, o una combinación de atributos, de una solución s a otra solución $s^{'} \in \mathcal{N}(s)$ en un paso (o iteración). $\mathcal{N}(s) := \{s^{'} \in S \mid \mathcal{N}(s, s^{'})\} \subseteq S$

Función de vecindad

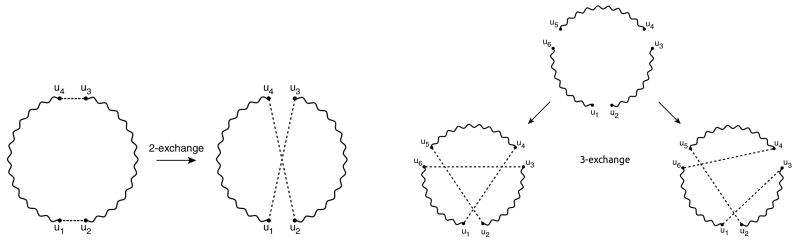
$$\mathcal{N}(x) = \{y \in S \mid dist(x, y) \leq \varepsilon\}$$



Función de vecindad

TSP

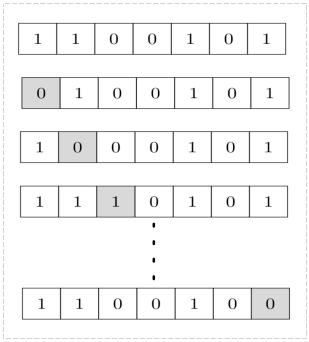
Para el TSP simétrico con n vertices, el tamaño del espacio de búsqueda es $(n-1)!/2$.



Cuando se realiza el 2-intercambio, el tamaño de la vecindad es $n(n-1)/2$. El 3-intercambio tiene una vecindad de $n(n-1)(n-2)/6$.

Función de vecindad

1-flip



Cambiar un bit de un vector binario. El tamaño de la vecindad es n, donde n es el número de posiciones del vector.

Función de vecindad

Las operaciones que definen una vecindad, en la mayoría de los casos, son bastante simples, por ejemplo:

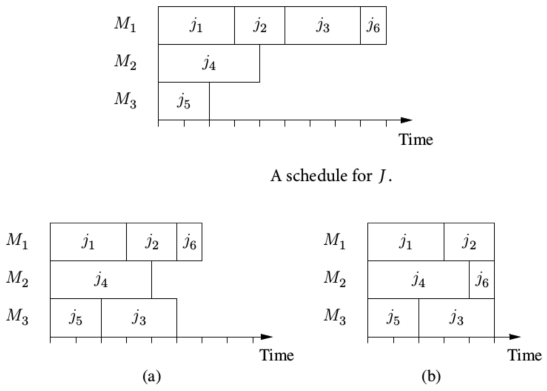
- remover un elemento
- insertar un elemento
- intercambiar un elemento
- mover un elemento de un conjunto a otro

Para un problema de programación de tareas en máquinas se pueden realizar los siguientes movimientos:

- Intercambiar un trabajo de la posición i a la posición j, donde $i \neq j$.
- Remover un trabajo de la posición i e insertarlo en la posición j.

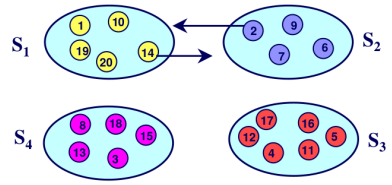
Función de vecindad

Mover un trabajo de una máquina con máxima carga a otra máquina con mínima carga.

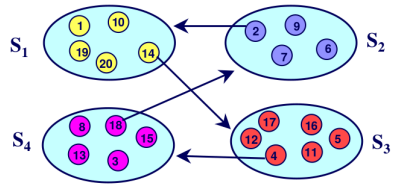


Función de vecindad

2-exchanges

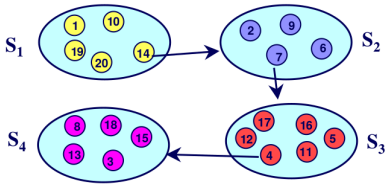


Multi-Swap



Función de vecindad

A Path Exchange



Two swaps