

VRP: Problema de rutas de vehículos

Gabriel Luque (gabriel@lcc.uma.es)

Transporte y Tráfico

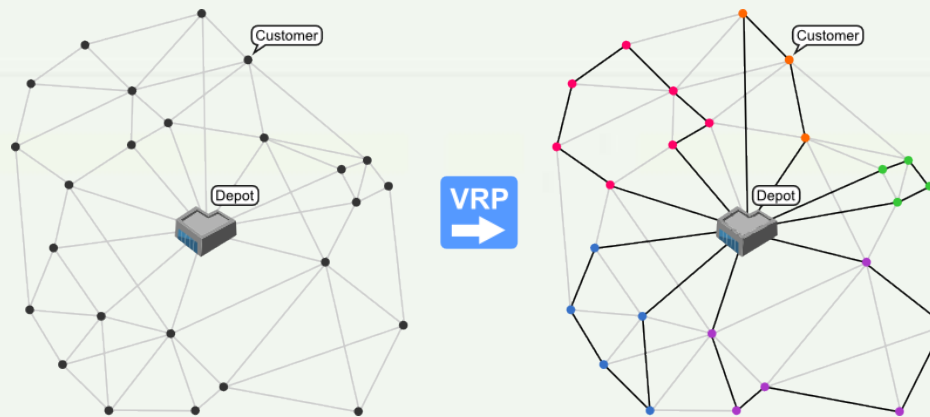
<http://neo.lcc.uma.es/vrp/>



Problema VRP

◆ Problema clásico de transporte y logística

- Hay múltiples clientes que hacen peticiones
- Hay uno o varios almacenes/depósitos
- Hay uno o varios vehículos
- Encontrar las rutas para servir a los clientes partiendo de uno de los almacenes minimizando el coste/tiempo/distancia



◆ En la actualidad sigue siendo de enorme importancia

Variantes

◆ Servicio:

- Tipo:
 - Solo recogidas o entregas
 - Mezcla de recogidas y entregas
 - Recogida de objetos y entregarlos a otro cliente
- Ventana de tiempo y flexibilidad

◆ Vehículos:

- 1 o varios
- Tipos
- Capacidades

◆ Depósitos:

- 1 o varios
- Elementos disponibles

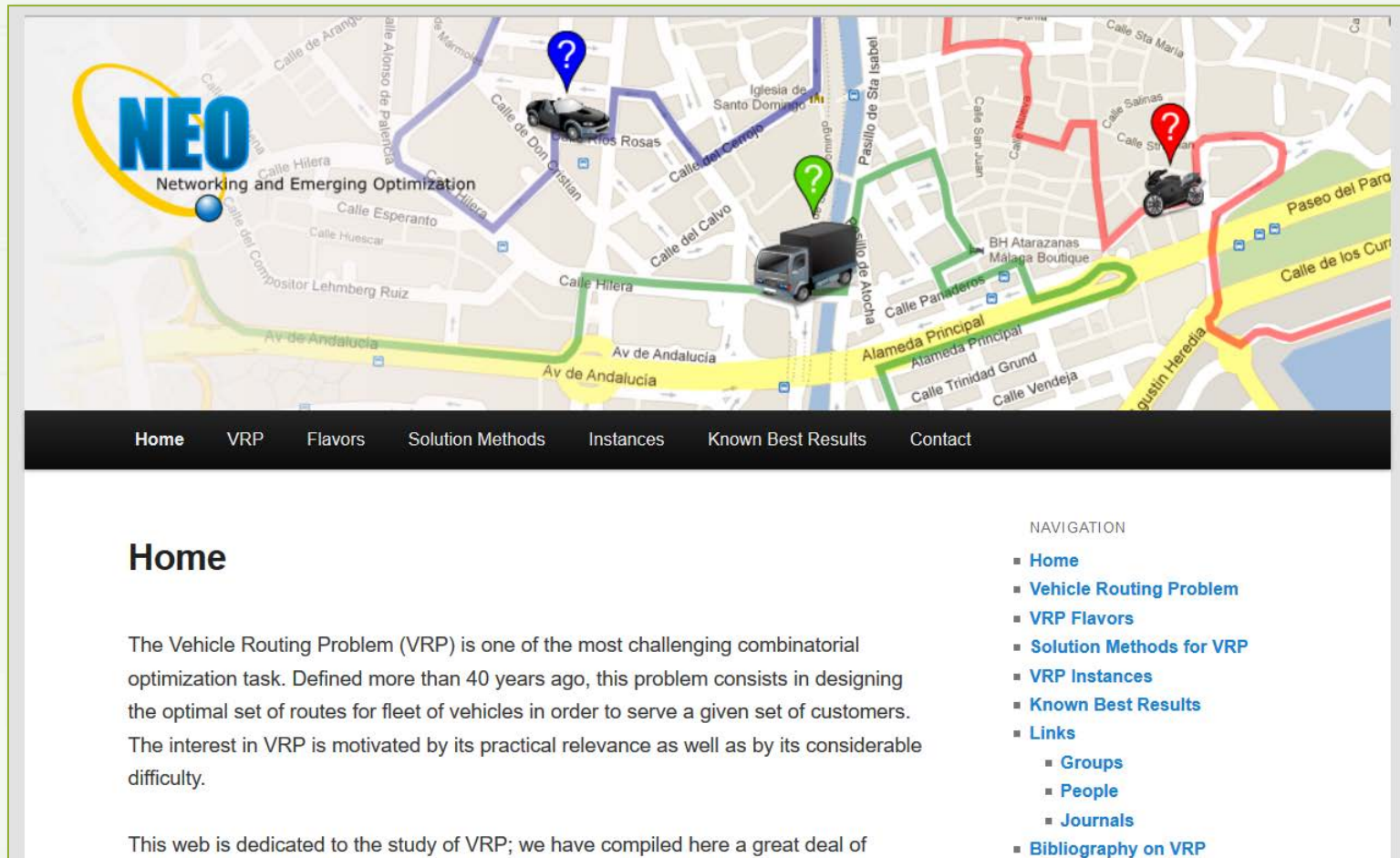
◆ Prioridad clientes

◆ Otros:

- Compatibilidad Vehículos y servicios
- Tiempo de descanso conductores
- Coste ruta variable

Variantes

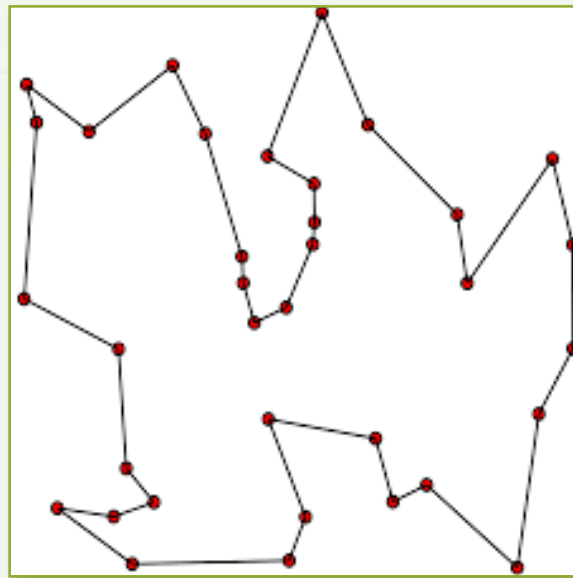
<http://neo.lcc.uma.es/vrp/>



Variantes

◆ El TSP (problema del viajante) puede considerarse una variante de este problema:

- Múltiples clientes
- Tipo de servicio simple
- Un único vehículo
- Sin restricciones de capacidad, ventana de tiempo, ...



Problema abordado

- ◆ Minimizar tiempo de atención a todos los clientes (suma de tiempo de rutas)
- ◆ Un único depósito
- ◆ Múltiples vehículos (no acotados) homogéneos (misma capacidad)
- ◆ Tipo de servicio simple (por ejemplo, solo reparto), sin ventana de tiempo y el coste de la entrega fija
- ◆ Tiempo de viaje entre clientes fijo
- ◆ Limitación temporal en la ruta

Problema abordado

◆ Objetivo:

- Minimizar la suma de los tiempos que tarda cada ruta

◆ Restricciones:

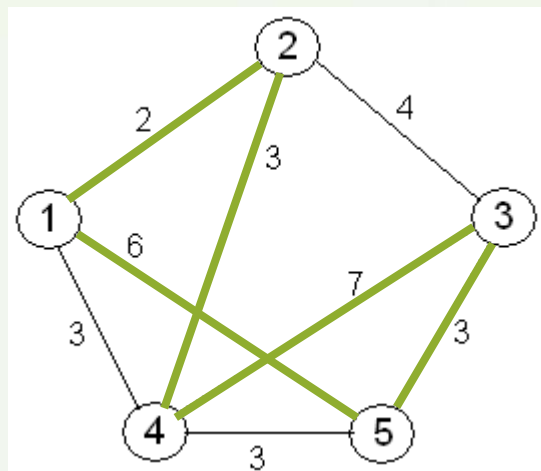
- Cada cliente se atiende una sola vez
- Todos los clientes deben ser atendidos
- Cada ruta debe empezar y acabar en el almacén
- La suma de las peticiones de los clientes de una ruta no puede superar la capacidad
- La suma de los tiempos de desplazamiento entre cliente y la atención no debe superar un máximo

◆ CVRP (Capacitated Vehicle Routing Problem)

Cómo representar una solución

◆ Una ruta: Permutación

- Representa el orden en el que se visitan los clientes
- No es necesario añadir el almacén al inicio y fin (aunque debe considerarse en la evaluación)



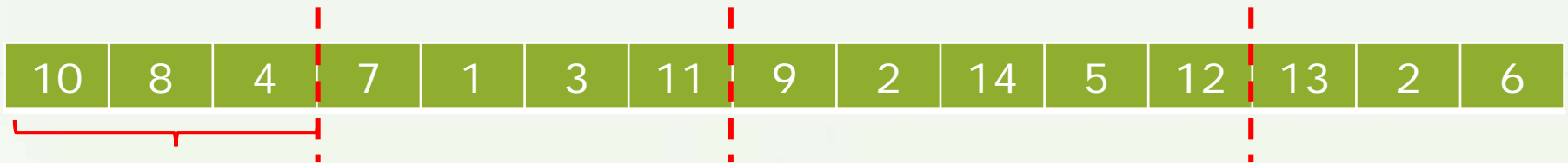
El 1 es el almacén

Cómo representar una solución

◆ Múltiples rutas

◆ Rutas implícitas: Permutaciones

- La solución no tiene ningún indicador de donde empieza y acaba las rutas
- Se usa un criterio de división



Cumple restric.

No cumple restric.

◆ Ventajas:

- Simple
- Muchos operadores ya existentes
- Cumple restricciones automáticamente
- Minimiza el número de vehículos

◆ Inconvenientes:

- No permite todas las posibles soluciones

Cómo representar una solución

◆ Múltiples rutas

◆ Rutas explícitas: Permutaciones con separadores

- La solución no tiene ningún indicador de donde empieza y acaba las rutas
- Se usa un criterio de división

10	8	4	0	7	3	11	9	2	0	5	1	0	2	6
----	---	---	---	---	---	----	---	---	---	---	---	---	---	---

◆ Ventajas:

- Permite todas las soluciones
- Operadores especiales (inter- e intra- ruta)

◆ Inconvenientes:

- Operadores especiales (costosos)
- Hay que verificar restricciones
- Complejo: representación, configuración (op.) e implementación

Implementando una propuesta

- ◆ Se utilizará la representación basada en permutaciones con rutas implícitas
- ◆ Clase `cInstance`: manejo de los datos:
 - Constructor: lee los ficheros
 - `nCustomers()`: número de clientes
 - `demand(i)`: demanda del cliente i
 - `distance(i,j)`: tiempo de ir i a j
 - `capacity()`: capacidad de los vehículos
 - `maxRouteTime()`: Tiempo máximo de ruta

Implementando una propuesta

◆ Solución: `vector<unsigned>`

- Las operaciones deben asegurar generar/mantener la permutación.

◆ Poniéndolo junto: solución aleatoria:

```
void generateSolution(Solution &solution, const cInstance &c){
    unsigned aux, ind1, ind2;
    unsigned max = c.nCustomers();

    solution.clear();

    // Create Ordered Array from 0 to nCustomers
    for(int i = 0; i < max; i++){
        solution.push_back(i+1);

        // move values to create random array
        for(int i = 0; i < (max*5) ; i++){
            ind1 = rand()%max;
            ind2 = rand()%max;

            aux = solution[ind1];
            solution[ind1] = solution[ind2];
            solution[ind2] = aux;
        }
    }
}
```

Implementando una propuesta

◆ Evaluando una solución: evaluate

```
double evaluate(const Solution &solution, const cInstance &c){
    double fitness = 0.0;
    int ultimo = 0;
    double dist = 0.0;
    int cap = c.capacity();

    register int c_actual;

    for(int j = 0; j < c.nCustomers(); j++) {
        c_actual = solution[j];

        if( ((dist + c.distance(ultimo,c_actual) + c.distance(c_actual,0)) > c.maxRouteTime())
            || (( cap - c.demand(c_actual)) < 0)
            // No solo mete una nueva ruta cuando se pasa sino cuando crear una nueva es mejor
            || ((c.distance(ultimo,0) + c.distance(0,c_actual)) < c.distance(ultimo,c_actual))
        ) {
            fitness += dist + c.distance(ultimo,0);
            ultimo = 0;
            dist = 0.0;
            cap = c.capacity();
        }
        else {
            dist += c.distance(ultimo,c_actual);
            cap -= c.demand(c_actual);
            ultimo = c_actual;
        }
    }

    fitness += dist + c.distance(ultimo,0);
}
```

Implementando una propuesta

◆ Poniéndolo junto: Random Search:

```
void RS(const cInstance &c, const unsigned steps, const bool verbose){
    Solution current_sol, best_sol;
    double fitness, best_fit;

    generateSolution(current_sol, c);
    fitness = evaluate(current_sol, c);
    best_sol = current_sol; best_fit = fitness;
    if(verbose) cout << fitness << endl;

    for(int i = 1; i < steps; i++)
    {
        generateSolution(current_sol, c);
        fitness = evaluate(current_sol, c);
        if(verbose) cout << fitness << endl;
        if(fitness < best_fit)
        {
            best_sol = current_sol;
            best_fit = fitness;
        }
    }

    if(verbose){
        cout << "Best solution: " << endl;
        for(int i = 0; i < best_sol.size(); i++)
            cout << best_sol[i] << " ";
        cout << endl << "Fitness: " << best_fit << endl;
    } else {
        cout << best_fit << endl;
    }
}
```

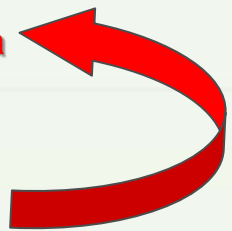
Genera solución aleatoria

Evalúa la solución

Genera solución aleatoria

Evalúa la solución

Actualiza la mejor



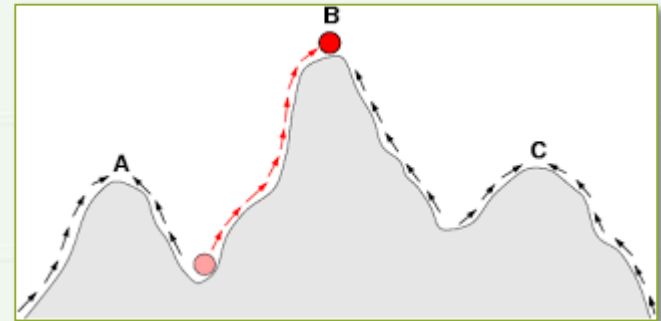
Ejercicios iniciales

1. Descargue el código de <https://github.com/GabJL/VRP>
2. Descomprima el código en la carpeta alumno
3. Compile el código:
 - Abre un terminal
 - `cd VRP-master/` # Muévase al directorio code
 - `make` # Compilación
4. En el código se facilita una búsqueda aleatoria (RS). Ejecútelo:
 - `./RS instances/vrpnc1.txt 1000 1` # Ejecución con 1000 iteraciones y 1 ejecución
5. Ejecute 30 veces el algoritmo y guarde los resultados en `resRS.txt`:
 - `./RS instances/vrpnc1.txt 1000 30 > resRS.txt`
- ◆ Todos los algoritmos tienen 3 parámetros:
 - instancia,
 - número de iteraciones y
 - número de ejecuciones independientes. (si es 1 muestra la evolución si es > 1 solo el resultado final)
- ◆ Se ejecutan igual que el ejemplo.

Técnicas de Optimización

◆ Métodos de escalada (*hill climbing*)

1. $s \leftarrow \text{generarSolucion}()$
2. $f \leftarrow \text{evaluar}(s)$
3. Repetir N veces
 1. $s' \leftarrow \text{generarVecino}(s)$
 2. $f' \leftarrow \text{evaluar}(s')$
 3. if ($f' < f$)
 1. $s \leftarrow s'$
 2. $f \leftarrow f'$
4. Solución final en s con fitness f



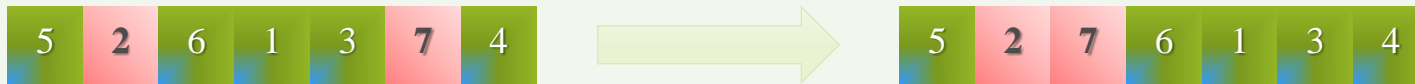
◆ ¿Vecindario en VRP?

- General + reparación
- Específico

Ejercicios: métodos de escalada 1

◆ Inserción:

- Se selección 2 posiciones aleatorias
- Se mueve uno junto al otro



6. El HC con el vecindario de Inserción: HC1. Pruebe a ejecutar 1 ejecución este algoritmo y observe como el fitness decrece
 - `./HC1 instances/vrpnc1.txt 1000 1`
7. Ahora ejecute 30 veces el algoritmo y guarde los resultados en resHC1.txt
 - `./HC1 instances/vrpnc1.txt 1000 30 > resHC1.txt`
8. ¿Cuál es mejor RS o HC1?

Comparación experimental

- ◆ Diferentes ejecuciones dan diferentes resultados:
 - Se necesitan múltiples ejecuciones (30 mínimo)
 - Aplicar estadística para el análisis

- ◆ ¿Qué valor representa mejor a los valores?
 - Media si siguen una distribución normal (+ desviación típica)
 - Mediana en otro caso

- ◆ ¿Si la media/mediana son diferentes podemos decidir que los resultados son diferentes?
 - Normal (paramétrico): test t
 - No normal (no paramétrico): test de Wilcoxon

Comparación experimental (en R)

◆ Entrar:

- R

◆ Salir

- `quit()`

◆ Cargando datos:

- `x <- read.csv("file.txt", header = TRUE)`
- `x <- read.table("file.txt", sep= "\t")`

◆ Acceso a una columna (muestra):

- `x$Nombre`
- `x$V1` (por defecto las nombra como V1 V2 ...)

◆ Descripción de los datos:

- `summary(x)` `mean(x$V1)` `median(x$V1)` `var(x$V1)`

Comparación experimental (en R)

◆ Normalidad:

- test de Shapiro-Wilk: `shapiro.test(x$V1)`
- $p\text{-value} > 0.05$ (95% confianza) \Rightarrow son similares (sigue distribución normal)

◆ Test paramétricos: ($p\text{-value} < 0.05 \Rightarrow$ Diferentes)

- t-test: `t.test(x,y)`

◆ Test no paramétricos: ($p\text{-value} < 0.05 \Rightarrow$ Diferentes)

- test de Wilcoxon: `wilcox.test(x$V1,y$V1)`

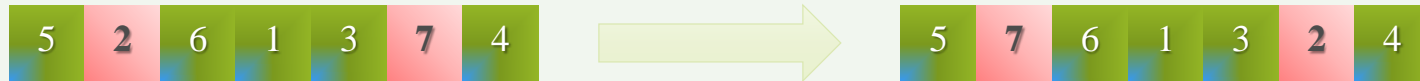
◆ Visual:

- `boxplot(c(x,y))`

Ejercicios: métodos de escalada 2

◆ Intercambio:

- Se selección 2 posiciones aleatorias
- Se intercambian las posiciones



9. Edite el código HC2.cpp (actualmente es igual que HC1):

- Localice la función generateNeighbor
- Borre las líneas que borran (erase) y añaden (insert)
- Intercambie en neigh el contenido de pos1 y pos2
- Compílelo con make y ejecútelo con `./HC2 instances/vrpnc1.txt 1000 1`

10. Ahora ejecute 30 veces el algoritmo y guarde los resultados en resHC2.txt

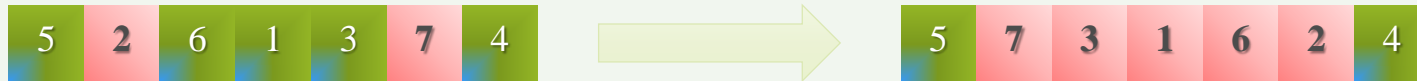
- `./HC2 instances/vrpnc1.txt 1000 30 > resHC2.txt`

11. ¿Cuál es mejor HC1 o HC2? Aplique los test oportunos

Ejercicios: métodos de escalada 3

◆ Inversión (2-opt):

- Se selección 2 posiciones aleatorias
- Se invierten la sub-permutación entre ambas posiciones



12. Ejecute 30 veces el algoritmo HC3 (que implementa inversión) y guarde los resultados en resHC3.txt

- `./HC3 instances/vrpnc1.txt 1000 30 > resHC3.txt`

13. ¿Cuál es mejor HC1, HC2 o HC3?

Comparación experimental

- ◆ Diferentes ejecuciones dan diferentes resultados:
 - Se necesitan múltiples ejecuciones (30 mínimo)
 - Aplicar estadística para el análisis

- ◆ ¿Qué valor representa mejor a los valores?
 - Media si siguen una distribución normal (+ desviación típica)
 - Mediana en otro caso

- ◆ ¿Si la media/mediana son diferentes podemos decidir que los resultados son diferentes?
 - Normal (paramétrico):
 - 2 casos: test t
 - > 2 casos: Análisis de la varianza (ANOVA)
 - No normal (no paramétrico):
 - 2 casos: test de Wilcoxon
 - > 2 casos: test de Kruskal-Wallis

Comparación experimental (en R)

◆ Test paramétricos: ($p\text{-value} < 0.05 \Rightarrow$ Diferentes)

- 2 muestras: t-test:

```
t.test(x,y)
```

- > 2 muestras: ANOVA:

```
# Preparación de datos
```

```
xdf <- data.frame(cbind(x1, x2, x3, x4))
```

```
xs <- stack(xdf)
```

```
# Test y resultado
```

```
anova1 <- aov(values ~ ind, data = xs)
```

```
summary(anova1)
```

```
# Post análisis si son diferentes
```

```
TukeyHSD(anova1)
```


Comparación experimental (en R)

◆ Test no paramétricos: ($p\text{-value} < 0.05 \Rightarrow$ Diferentes)

- 2 muestras: test de Wilcoxon

```
wilcox.test(x$V1,x$V2)
```

- > 2 muestras: test de Kruskal-Wallis:

```
# Preparación de datos
```

```
xdf <- data.frame(cbind(x1, x2, x3, x4))
```

```
xs <- stack(xdf)
```

```
# Test y resultado
```

```
kruskal.test(values ~ ind, data = xs)
```

```
# Post análisis si son diferentes (install.packages("dunn.test"))
```

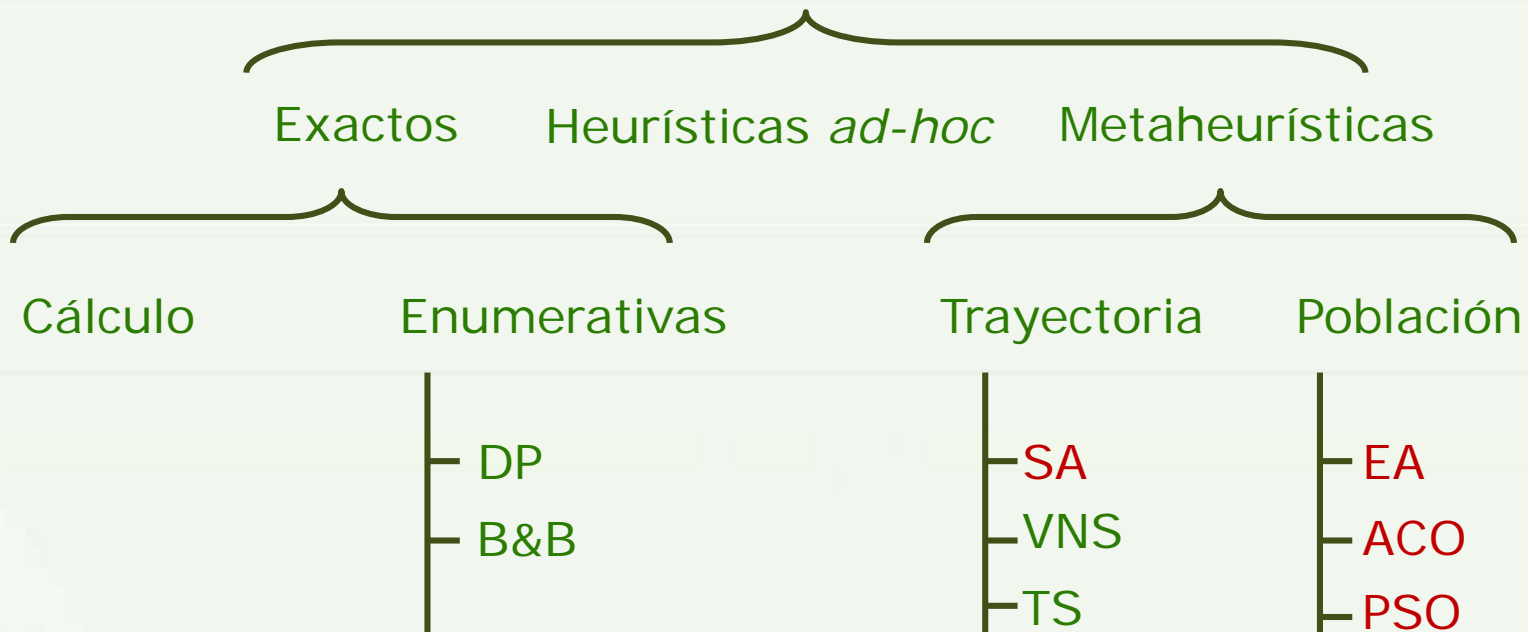
```
library("dunn.test")
```

```
dunn.test(xs$values,xs$ind, kw = TRUE, "holm")
```

```
# Los * indican que son diferentes
```

Técnicas de Optimización

Algoritmos de Optimización

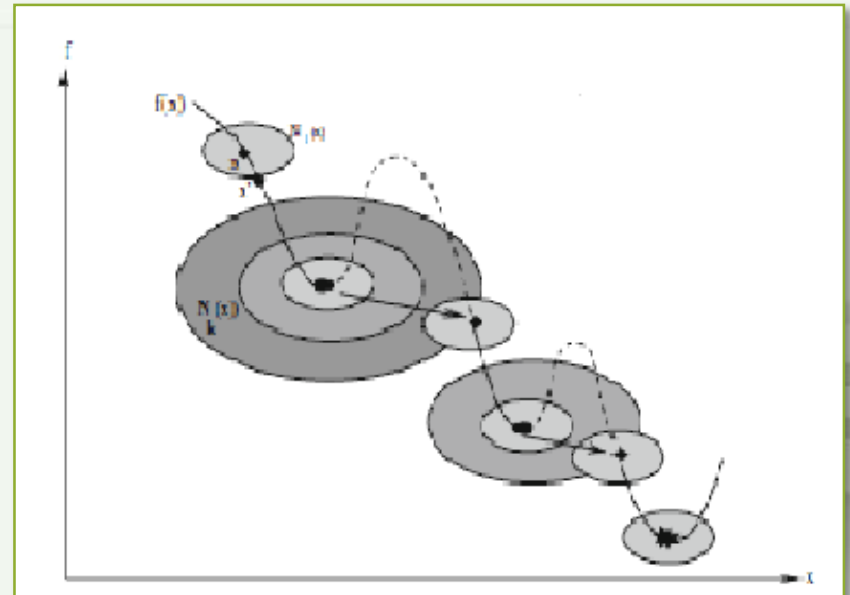


Inspiradas en la naturaleza

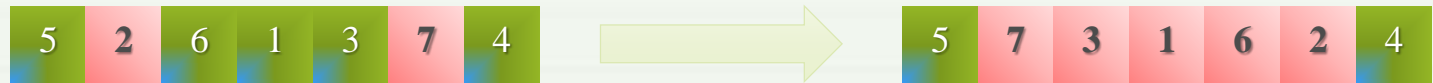
Técnicas de Optimización

◆ Variable Neighborhood Search (VNS)

1. Se definen K vecindarios (de más reducido a más amplio): V_1, V_2, \dots, V_k
2. Parte de una solución inicial (generalmente aleatoria)
3. $N = 1$
4. Se busca el mejor vecino usando el vecindario V_N
 - Si es mejor que la actual, se reemplaza la actual por la nueva y $N = 1$
 - Si es peor, $N++$
5. Se vuelve al paso 4



Ejercicios: VNS



◆ Vecindarios:

- Usaremos inversión como base
- La variedad de vecindarios la conseguimos ajustando como elegimos los números aleatorios:
 - Vecindario 1: posiciones consecutivas
 - Vecindario 2: posiciones separadas por un elemento
 - Vecindario 3: posiciones separadas por 2 elementos
 - Vecindario i: posiciones separadas por i elementos

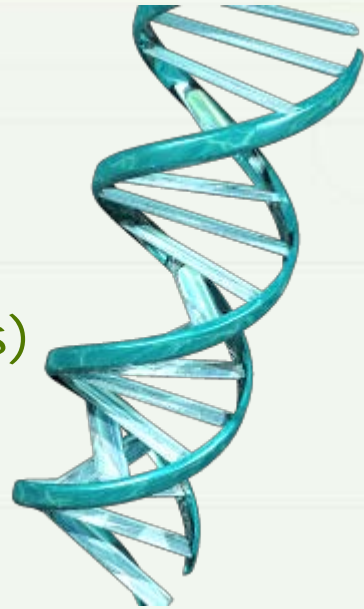
14. Ejecute 30 veces el algoritmo VNS y guarde los resultados en resVNS.txt

- `./VNS instances/vrpnc1.txt 1000 30 > resVNS.txt`

15. ¿Mejora a HC3 (se basan en el mismo operador)? ¿y al resto de los HC?

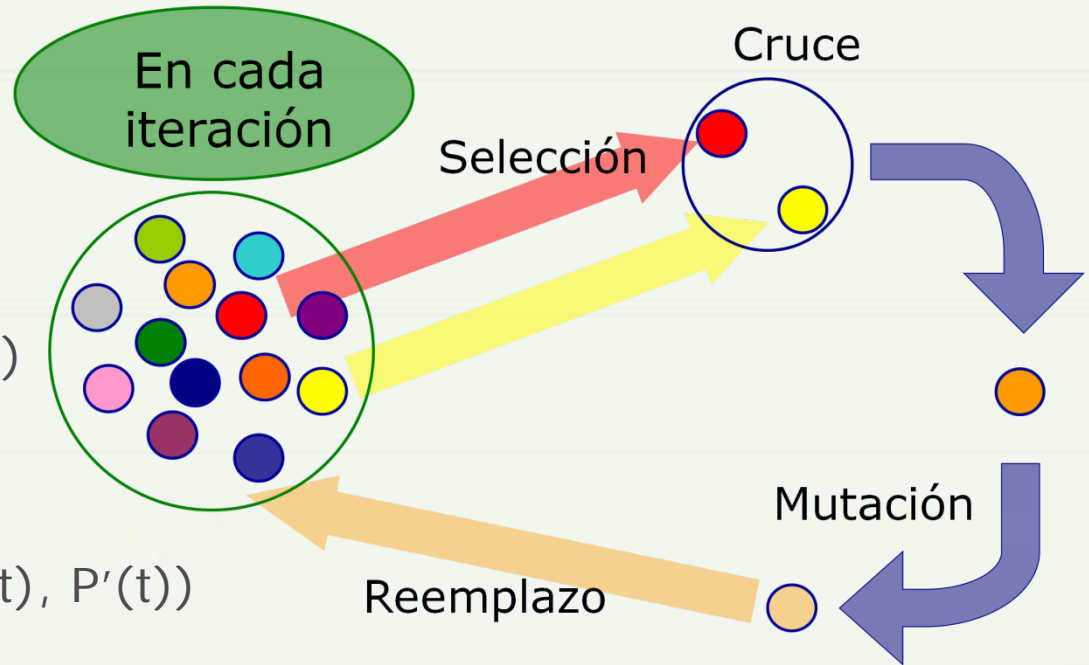
Algoritmos Evolutivos

- ◆ Basados en la evolución natural de Darwin
- ◆ Algoritmo poblacional (maneja múltiple soluciones)
- ◆ Tres pasos principales:
 - Selección
 - Reproducción (cruce y mutación)
 - Remplazo
- ◆ Múltiples familias de acuerdo a cómo realizan esos pasos:
 - Algoritmos Genéticos (GA),
 - Programación Genética (PG),
 - Estrategias Evolutivas (ES),
 - ...



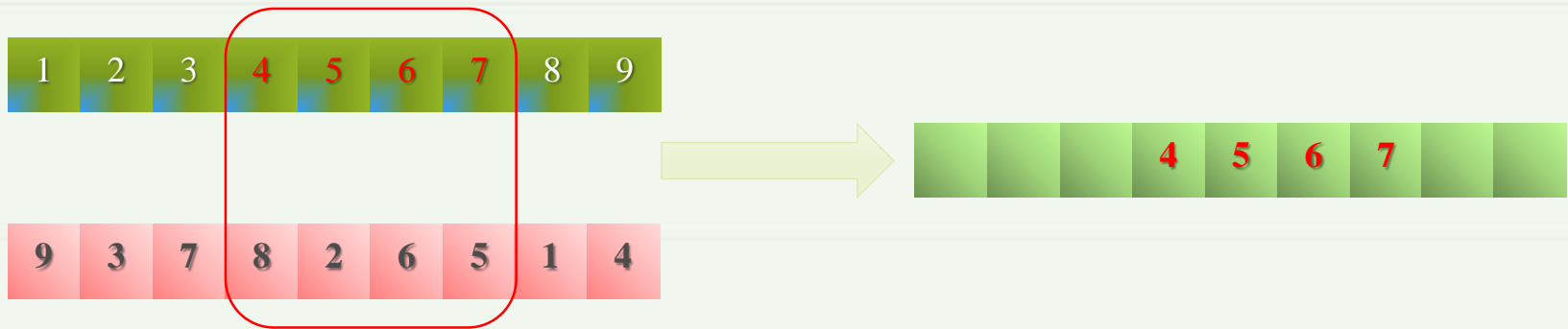
Algoritmos Evolutivos

```
t := 0
Generar(P(t))
Evaluar(P(t))
while not end condition do
  P'(t) := Seleccionar(P(t))
  P'(t) := Cruzar(P'(t))
  P'(t) := Mutar(P'(t))
  Evaluar(P'(t))
  P(t+1) := Reemplazar(P(t), P'(t))
  t := t+1
end while
```

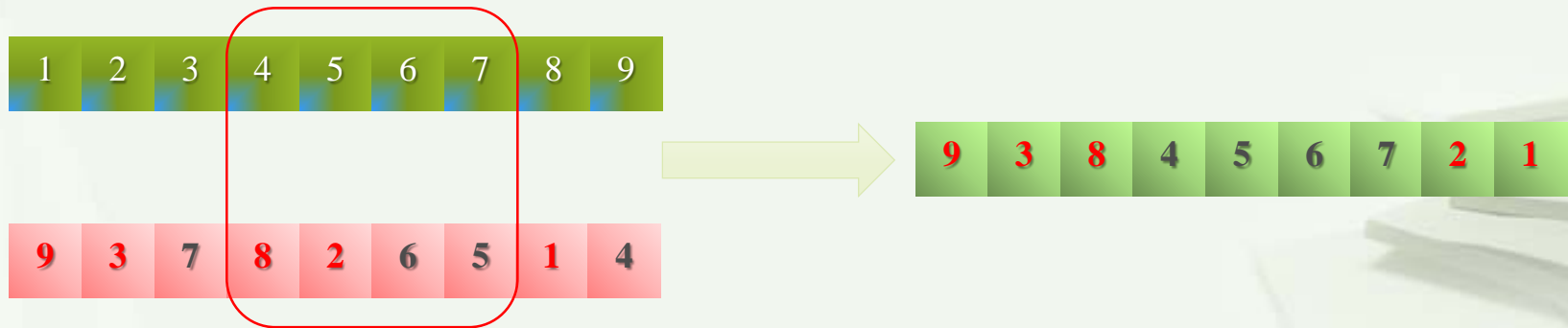


OX: Order Crossover

- ◆ Copiamos la sub-permutación elegida aleatoriamente



- ◆ Copiamos el resto siguiendo el orden del segundo individuo:
9, 3, 8, 2, 1



Ejercicios: GA

En el código se facilita el código de un GA con las siguientes características:

- Población: 10
- Estado estacionario: genera un único descendiente por iteración
- Mutación: Inversión
- Recombinación: OX
- Selección: Aleatoria
- Reemplazo: Aleatoria (pero solo añade el nuevo si es mejor)

16. Ejecute el código:

- `./ssGA instances/vrpnc1.txt 1000 30 > resGA.txt`

17. Realice los test estadísticos correspondientes para ver si es mejor o no que los otros probados.

18. Ejecute una sola vez el GA para observar la evolución, ¿converge? ¿es posible que con más generaciones mejore?

VRP: Problema de rutas de vehículos

Gabriel Luque (gabriel@lcc.uma.es)

Transporte y Tráfico

<http://neo.lcc.uma.es/vrp/>

