Programación 3
TUDAI - 2024

Algoritmos greedy

Docente: Federico Casanova.

Características

- Se aplica a la solución de problemas de optimización (se quiere maximizar o minimizar algo).
- Los algoritmos de optimización generalmente hacen una secuencia de pasos, con un conjunto de decisiones o alternativas posibles en cada paso.
- Un algoritmo greedy (del ingles: avaro, codicioso, voraz) siempre toma la alternativa (decisión) que parece mejor en ese momento (localmente óptima), con la esperanza de construir una solución globalmente óptima.
- La alternativa que selecciona es en función de la información que tiene disponible en ese momento.
- Una vez tomada la decisión no vuelve a replanteársela en el futuro. No plantea prueba y error.
- No siempre se llega a soluciones óptimas, pero se puede utilizar para obtener aproximaciones. Sin embargo, existen problemas para los que se demuestra que la solución greedy siempre da la mejor solución.
- La ventaja es que son algoritmos de "baja" complejidad computacional. Complejidades polinomiales (n, n^2, n^3…).

Características

- Generalmente se dispone de un conjunto de candidatos de los cuales se debe seleccionar un subconjunto que optimice alguna propiedad.

- A medida que avanza el algoritmo, se van seleccionando candidatos con un criterio de optimalidad local (criterio greedy) y se los coloca en el conjunto solución S de candidatos aceptados.

- La selección que se hace en cada paso es irrevocable, o sea no puede cambiarse luego (no se revisan las decisiones ya tomadas).

Características

- Ejemplo
  - Problema: Dado un conjunto de billetes, ¿cuál es la mínima cantidad de billetes necesarios para pagar *P* pesos?.
  - Solución *greedy*: De mi conjunto de candidatos, elijo el de mayor valor posible. O sea, dar en lo posible billetes de denominación grande.
  - Billetes: $1000, $500, $200, $100, $50, $20, $10.

Por ejemplo supongamos P= 630
Billete más grande <= 630 : agrego 500 al conjunto solución, restan 130
Billete más grande <= 130 : agrego 100 al conjunto solución, restan 30
Billete más grande <= 30 : agrego 20 al conjunto solución, restan 10
Billete más grande <= 10 : agrego 10 al conjunto solución, restan 0. FIN – Resultado 4 billetes, y es la menor cantidad posible!

Para este problema la estrategia greedy siempre da la mejor solución siempre.

Estructura general de Algoritmos greedy

```
Funcion greedy(conjunto<candidatos> C)
{ // Inicialmente el conjunto C contiene todos los candidatos
   conjunto<candidatos> S;    // Conjunto solución, inicialmente Vacío
   while (!C.vacio() && !solucion(S))  {  //mientras haya candidatos y no se llegó a una solución
       x = seleccionar(C);  // determina el mejor candidatos del conjunto a seleccionar
       C.borrar(x);
       if (factible(S, x))
            S.agregar(x);
    }
   if solucion(S)
      return S
   else
      return "No_hay_solucion";
}
```

Elementos:

- Conjunto de *candidatos a seleccionar C.*
- Conjunto de *candidatos seleccionados S.*
- *Función Solución*: determina si los candidatos seleccionados han alcanzado una solución.
- *Función Seleccionar*: determina el mejor candidato del conjunto a seleccionar en ese momento **(criterio greedy).**
- *Función de Factible*: determina si válido para nuestro problema agregar el candidato seleccionado a la solución.
- Siempre está presente una *Función Objetivo*: da el valor de la solución alcanzada que queremos sea óptima (maximice o minimice algo).

Problemas clásicos – Mochila Fraccionaria

Mochila capacidad 2 Kg.

Pan 1 Kg. - $ 1500

Plata 0,5 Kg. - $ 500.000

Chocolate 1 Kg. - $ 5.600

Problemas clásicos – Mochila Fraccionaria

Problema de la Mochila. Se tienen n objetos y una mochila. Para $i = 1,2,..n$, el objeto i tiene un peso positivo p_i y un valor positivo v_i. La mochila puede llevar un peso que no sobrepase P. El objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos transportados, respetando la limitación de capacidad impuesta. Los objetos pueden ser fraccionados, si una fracción x_i $(0 \le x_i \le 1)$ del objeto i es ubicada en la mochila contribuye en $x_i p_i$ al peso total de la mochila y en $x_i v_i$ al valor de la carga. Formalmente, el problema puede ser establecido como:

$$\text{maximizar} \quad \sum_{i=1}^{n} x_i v_i \quad , \quad \text{con la restricción} \sum_{i=1}^{n} x_i p_i \le P$$

donde $v_i > 0, p_i > 0$ y $0 \le x_i \le 1$ para $1 \le i \le n$.

Por ejemplo, para la instancia n = 3 y P = 20

$(v_1, v_2, v_3) = (25, 24, 15)$

$(p_1, p_2, p_3) = (18, 15, 10)$

Algunas soluciones posibles son:

(x_1, x_2, x_3)	$x_i\ p_i$	$x_i\ v_i$
(1/2, 1/3, 1/4)	16.5	24.25
(1, 2/15, 0)	20	28.2
(0, 2/3, 1)	20	31
(0, 1, 1/2)	20	31.5

puede observarse que **(0, 1, 1/2)** produce el mayor beneficio.

Problemas clásicos – Mochila Fraccionaria

Pseudo-código: (P es W, x el array de fracciones, w el array de pesos)

```
for (int i=0; i < n; i++)
          x[i] = 0;

peso_actual = 0;

while (peso_actual < W)
{
          i = seleccion();  // No definido cómo
          if ((peso_actual + w[i]) < W) {
                    x[i] = 1;
                    peso_actual = peso_actual + w[i];
          } else {
                    x[i] = (W - peso_actual) / w[i];
                    peso_actual = W;
          }
}

return x;
```

Tomo lo más posible del elemento seleccionado

- la función `selección()` no está especificada.
- para definirla se pueden considerar tres estrategias diferentes:
  1. seleccionar el elemento de mayor valor
  2. seleccionar el elemento de menor peso
  3. seleccionar el elemento que tenga mayor valor por unidad de peso

Problemas clásicos – Mochila Fraccionaria

- la función `selección()` no está especificada.
- para definirla se pueden considerar tres estrategias diferentes:
  1. seleccionar el elemento de mayor valor
  2. seleccionar el elemento de menor peso
  3. seleccionar el elemento que tenga mayor valor por unidad de peso

- sea $n = 5, W = 100$ y objetos con los siguientes pesos y valores:

	obj. 1	obj. 2	obj. 3	obj. 4	obj. 5
w	10	20	30	40	50
v	20	30	66	40	60

- las soluciones con las tres estrategias de selección son:

	obj. 1	obj. 2	obj. 3	obj. 4	obj. 5	Valor
Max v_i	0	0	1	0,5	1	146
Min w_i	1	1	1	1	0	156
Max v_i/w_i	1	1	1	0	0,8	164

2	1,5	2,2	1	1,2

Para este problema (mochila fraccionaria) la estrategia Greedy de seleccionar el elemento que tenga mayor valor por unidad de peso da la mejor solución siempre. En cambio si la mochila no es fraccionaria sino entera (cada elementos o se pone completo o no se pone) la técnica greedy no garantiza la major solución.

Problemas clásicos – Selecc. de Actividades

- Problema Selección de Actividades
  - Tenemos la entrada de una Exposición que tiene un conjunto de actividades
    - Con la entrada podemos asistir a cualquier actividad.
    - Para cada actividad conocemos su horario de comienzo y fin.
    - Hay actividades que se solapan en el tiempo.
  - Objetivo: Asistir al mayor número de actividades => Problema de selección de actividades.

- Entonces:
  - Dado un conjunto *C* de *n* actividades, se tiene
    si = tiempo de comienzo de la actividad *i*
    fi = tiempo de finalización de la actividad *i*
  - Encontrar el subconjunto de actividades compatibles S de tamaño máximo

Problemas clásicos – Selecc. de Actividades

- Fijemos los elementos de la técnica greedy:

• *Candidatos a seleccionar.*: Conjunto de Actividades C

• *Candidatos seleccionados*: Conjunto S, inicialmente S={Ø }

• *Función Solución*: Verificar C={Ø }. O sea ya se consideraron todas.

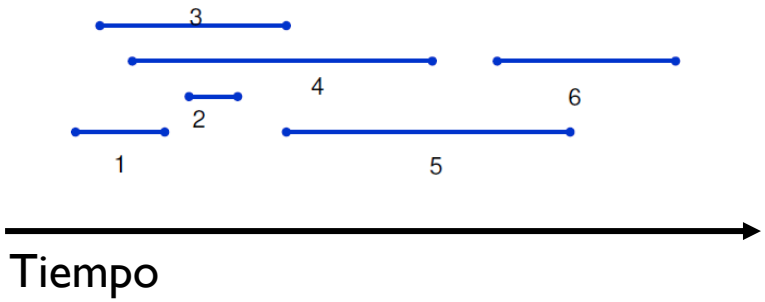
• *Función Selección*: determina el mejor candidato x. Qué estrategia ??

– Termina antes. (o sea el que deja más tiempo libre luego de la actividad para elegir más actividades del resto.)

• *Función de Factibilidad*: x es factible si es compatible con las actividades en S (no se solapan).

• *Función Objetivo*: Maximizar tamaño de S

Seleccionar tareas compatibles en orden ascendente de finalización (termina antes). Con esta selección, la idea que persigue es dejar el mayor espacio posible para que entren nuevas Actividades después de la seleccionada.



Con esta estrategia el algoritmo da siempre la mejor solución.

Problemas clásicos – Caminos más Cortos

Sea G=(V, A) un grafo dirigido, con **pesos positivos**.

Se plantea el problema de hallar los caminos más cortos desde un vértice orígen hacia cada uno de los restantes vértices.

El algoritmo de **Dijsktra** en un algoritmo greedy para resolver el **problema de los caminos más cortos desde un vértice origen hacia el resto de los vértices del grafo**.

La técnica greedy encuentra **siempre** la mejor solución (solución optima).

Condición para su aplicación:
- Los arcos deben tener asignado un valor **POSITIVO**.

Problemas clásicos – Alg. de Dijkstra

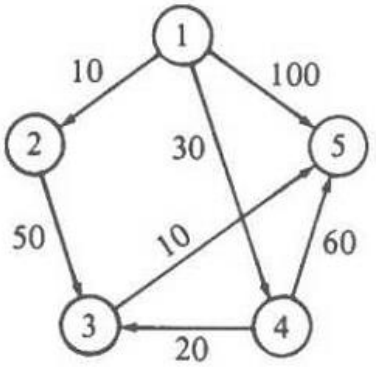
Pseudo-código

```
function Dijkstra(Grafo G,Vértice origen):
    for each Vértice v en G:          // Inicialización para cada vértice
        dist[v] = infinito            // La distancia inicial se establece en infinito
         padre[v] = indefinido        // El nodo anterior en el camino más corto desde el origen

    dist[origen] = 0                  // Distancia desde el origen hasta él mismo se establece en 0
    S = vacío                         // S: conj. vért. para los que ya se calculó el camino min desde origen
     while (G.Vértices – S) no es vacío:  // loop principal (mientras haya vért. sin considerar)
         u = vértice en (G.Vértices – S) tal que dist[u] tiene el menor valor
         S = S U {u}
         for each v en (G.Vértices – S) que sea adyacente a u:
              if (dist[u] + dist_entre(u, v)) < dist[v])
                   dist[v] = dist[u] + dist_entre(u, v)        //Proceso de actualizar loa costos de
                   padre[v] = u                                los vértices adyacentes a u, se lo
     return padre[ ]                                           llama Relajar(u) ó r(u)
```

Problemas clásicos – Alg. de Dijkstra

```
u = vértice en (G.Vértices – S) tal que dist[u] tiene el menor valor
S = S U {u}
for each v en (G.Vértices – S) que sea adyacente a u:    //Relajar(u)  r(u)
    if (dist[u] + dist_entre(u, v)) < dist[v])
        dist[v] = dist[u] + dist_entre(u, v)
        padre[v] = u
```



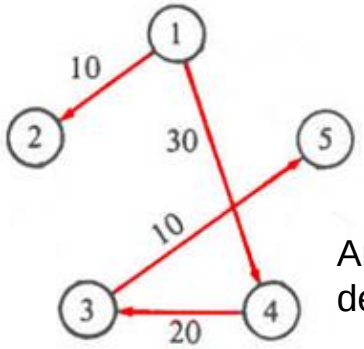
Supongamos el grafo contiene los vértices {1,2,3,4,5}
Y que el origen es el vértice **1**. O sea se quiere encontrar la distancia mínima entre el vértice 1 y cada uno de los demás vértices.

Paso	u	S	dist[1]	dist[2]	dist[3]	dist[4]	dist[5]	padre[1]	padre[2]	padre[3]	padre[4]	padre[5]
Inicialización	-	{ }	0	∞	∞	∞	∞	indef	indef	indef	indef	indef
Loop1- r(u)	1	{1}	**0**	10	∞	30	100	indef	1	indef	1	1
Loop2- r(u)	2	{1,2}	**0**	**10**	60	30	100	indef	1	2	1	1
Loop3- r(u)	4	{1,2,4}	**0**	**10**	50	**30**	90	indef	1	4	1	4
Loop4- r(u)	3	{1,2,4,3}	**0**	**10**	**50**	**30**	60	indef	1	4	1	3
Loop5- r(u)	5	{1,2,4,3,5}	**0**	**10**	**50**	**30**	**60**	indef	1	4	1	3

Solución:

dist[1]	dist[2]	dist[3]	dist[4]	dist[5]
0	10	50	30	60

padre[1]	padre[2]	padre[3]	padre[4]	padre[5]
indef	1	4	1	3

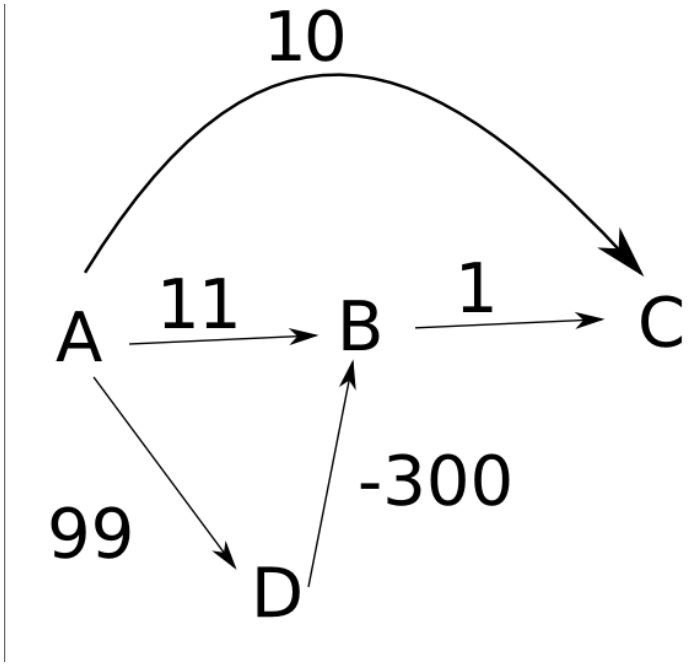


Arbol de caminos mínimos desde 1

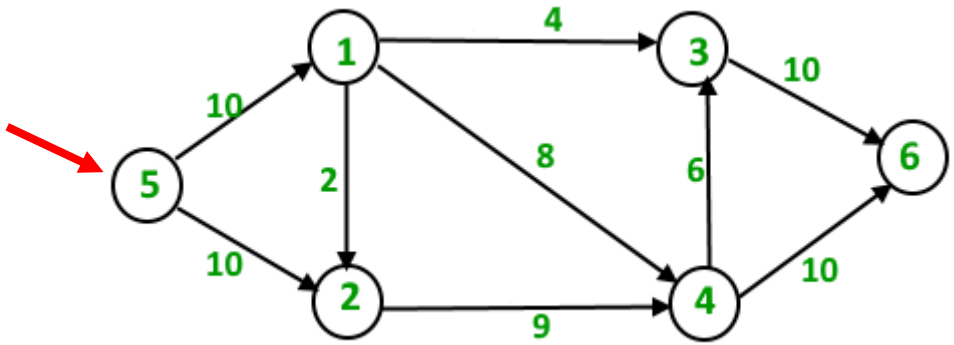
El algoritmo de Dijkstra SIEMPRE da la mejor solución (los caminos más cortos desde un orígen)

Problemas clásicos – Alg. de Dijkstra

Por qué para que funcione Dijkstra las aristas tienen que ser todas positivas ?
Supongamos origen A.



Problemas clásicos – Alg. de Dijkstra



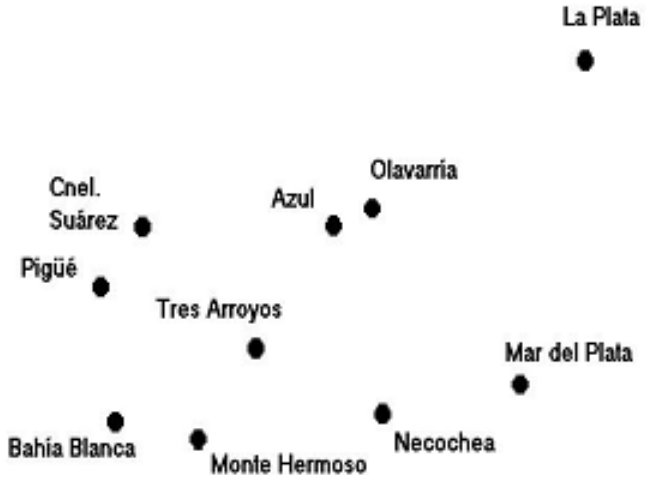
Paso	u	S	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]

Tarea: realizar el seguimiento para este grafo y completar la tabla.

Puede aplicarse a grafos no dirigidos ?

Problemas clásicos – Problema del Viajante

problema del agente viajero (TSP)



- se tienen *n* ciudades y las distancias entre cada par de ellas, representadas en una matriz.
- la ciudad de origen es Azul

- Problema: VIAJANTE se quiere partir de una de ellas y visitar exactamente una vez cada ciudad, regresando al punto de partida al final, y recorriendo la menor distancia posible.

Estrategia greedy, empezando del punto de partida elegir en cada paso la ciudad más próxima no visitada, la estrategia greedy es que si en cada paso voy a la ciudad más cercana desde donde me encuentro, al finalizar la suma total de distancias será la menor posible.

En este caso, como veremos, la estrategia greedy no asegura la mejor solución.

Problemas clásicos – Problema del Viajante

Si arrancamos de la ciudad 1 y tenemos las siguientes distancias (grafo no dirigido):

Ciudades	1	2	3	4	5	6
1	–	3	10	11	7	25
2	3	–	8	12	9	26
3	10	8	–	9	4	20
4	11	12	9	–	5	15
5	7	9	4	5	–	18
6	25	26	20	15	18	–

- solución *greedy*: 1,2,3,5,4,6,1, con distancia 60.
- solución optimal: 1,2,3,6,4,5,1, con distancia 58.

Para el problema del viajante la estrategia greedy no asegura una solución óptima.

Podría ser una buena aproximación para tener un primer valor para poda si queremos la solución óptima y resolviéramos el problema por backtracking.

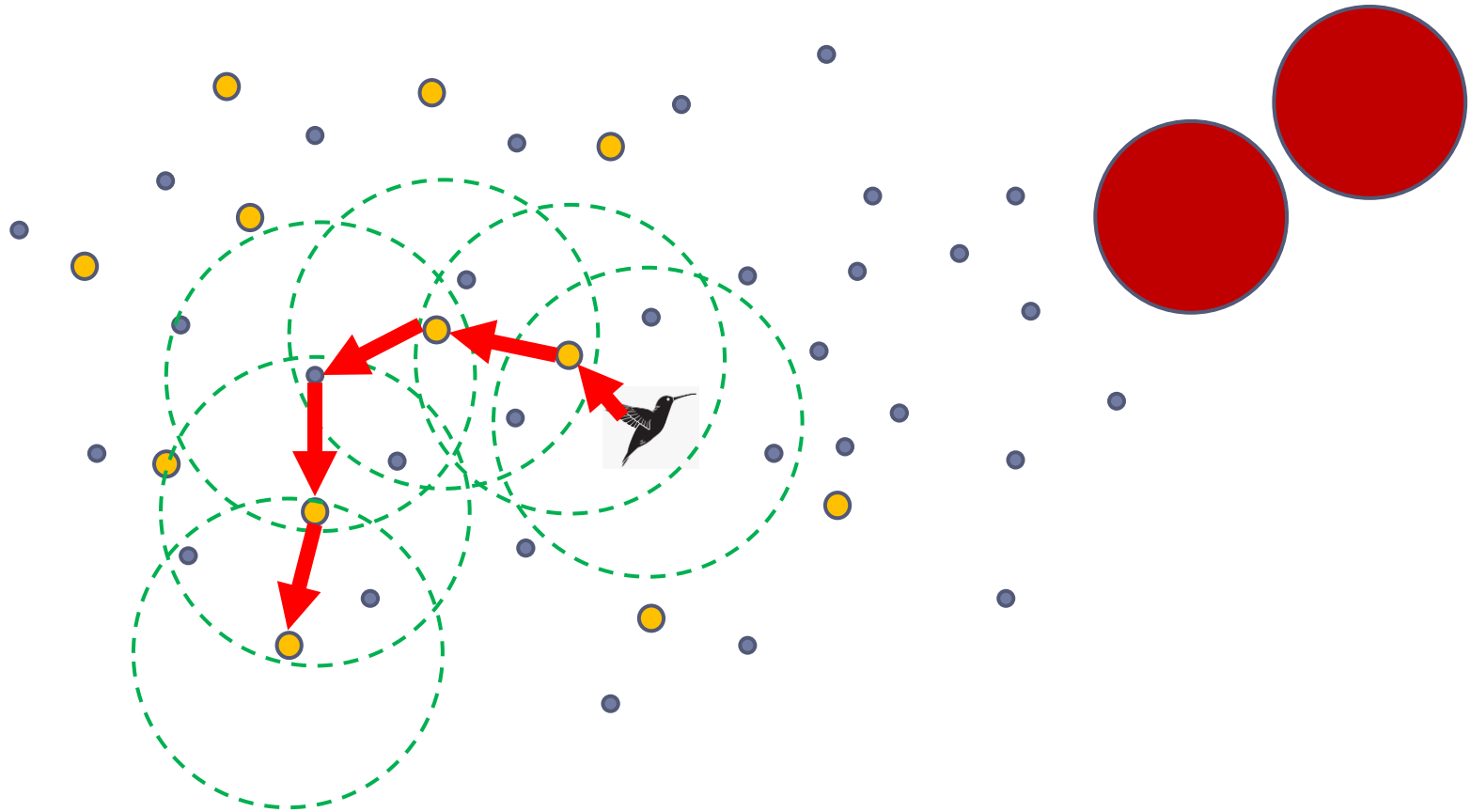
Problema del colibrí

El colibrí parte desde un lugar origen, y puede visitar como máximo 5 flores, su objetivo es recolectar la mayor cantidad de néctar total.

El colibrí puede moverse una cierta distancia sin tener que parar a descansar (radio de acción).

El colibrí puede ver dentro de su radio de acción cuánto néctar tiene cada flor antes de visitarla.

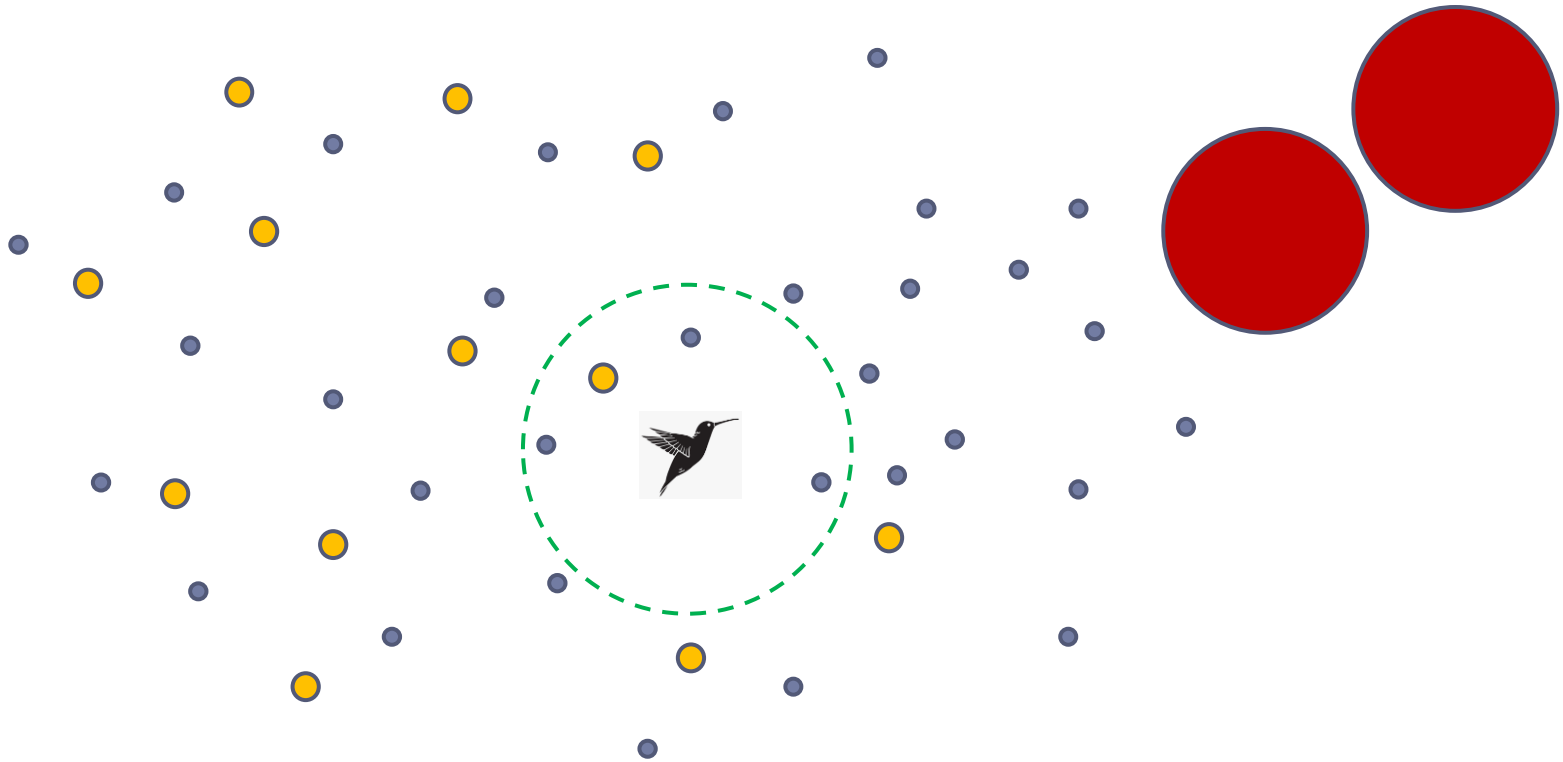
Estrategia Greedy



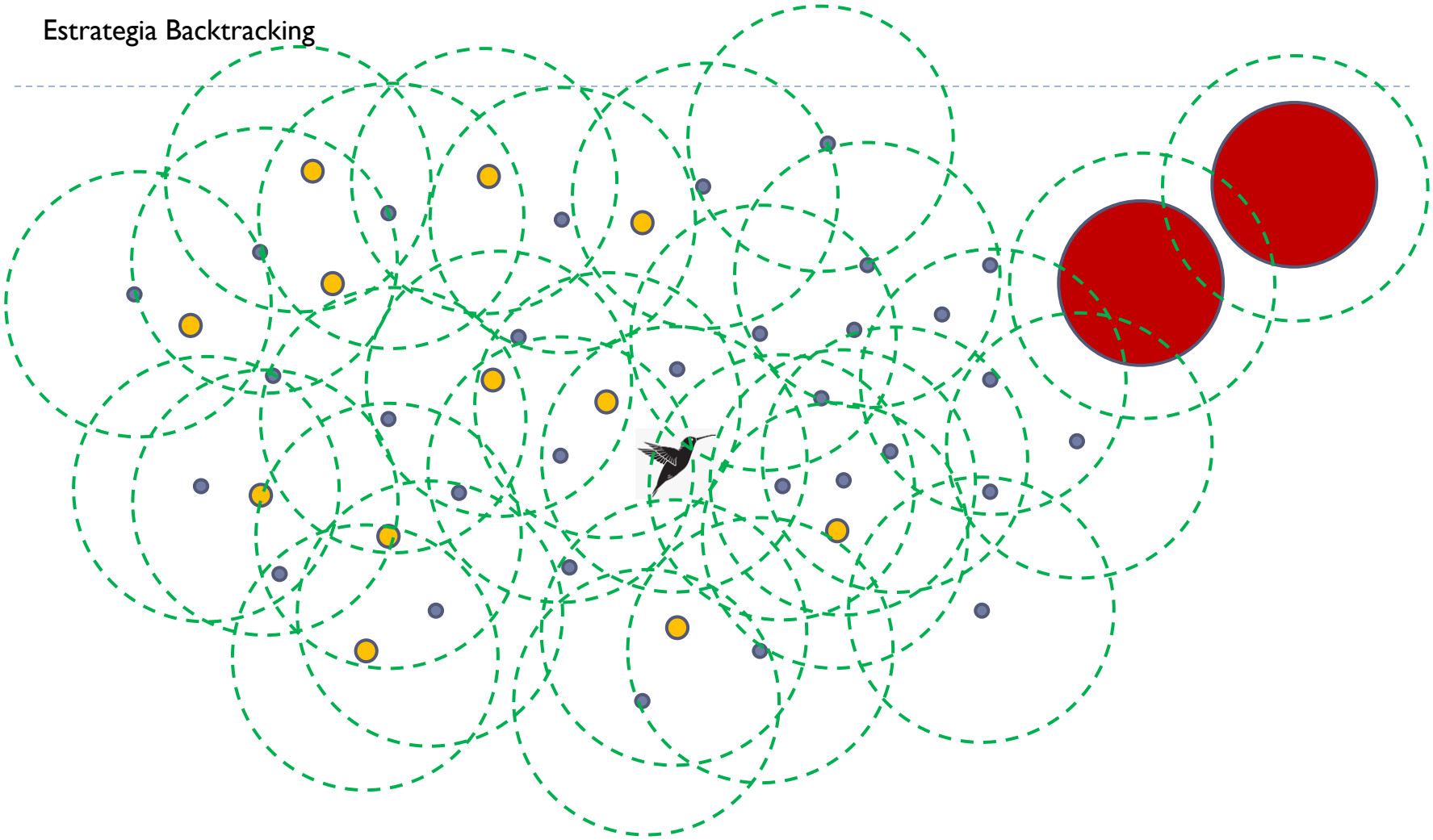
Resultado:

El tamaño de los circulos es proporcional al contenido de nectar de la flor.

Estrategia Backtracking



Estrategia Backtracking



Estrategia Backtracking