Búsqueda de todas las soluciones

```
proc buscar_todas_las_soluciones(paso : T)
{
   inicializar_alternativas()
   while (alternativas) do
   {
      obtener_siguiente_alternativa()
      if es_alternativa_válida() then
      {
         almacenar_paso()
         if es_solución() then
            procesar_solución()
         else
            buscar_todas_las_soluciones(nuevo_paso)
         borrar_paso()
      }
   }
}
```

Búsqueda de una solución

```
proc buscar_una_solución(paso : T)
{
   inicializar_alternativas()
   while (alternativas ∧ ¬solución_encontrada) do
   {
      obtener_siguiente_alternativa()
      if es_alternativa_válida() then
      {
         almacenar_paso()
         if es_solución() then
         {
            solución_encontrada ← true
            procesar_solución()
         }
         else
         {
            buscar_una_solución(nuevo_paso)
            if ¬solución_encontrada then
               borrar_paso()
         }
      }
   }
}
```

Búsqueda de la solución óptima

```
proc buscar_solución_óptima(paso : T)
{
   inicializar_alternativas()
   while (alternativas) do
   {
      obtener_siguiente_alternativa()
      if es_alternativa_válida() then
      {
         almacenar_paso()
         if es_solución() then
            if es_óptima() then
               sustituir_solución()
         else
            buscar_solución_óptima(nuevo_paso)
         borrar_paso()
      }
   }
}
```

Ejemplo: Búsqueda de todas las soluciones

Planteamiento:

Dado un grafo dirigido, encontrar todos los caminos posibles entre dos vértices dados (vértice inicial y vértice final) de manera tal que los vértices contenidos en cada camino sean visitados una sola vez.

Solución:

- Identificar los elementos principales en el problema.

Esquema a utilizar: Búsqueda de todas las soluciones (se pide encontrar todos los caminos posibles entre dos vértices dados del grafo).
Paso: Partiendo de un vértice cualquiera *v*, visitar un vértice adyacente *w*.
Solución: Cuando el vértice visitado (*w*) coincide con el vértice final (*vf*).

- Especificar las constantes, tipos y variables requeridas.

Constantes
n = ?
n es una constante entera positiva que representa el número de vértices del grafo.

Tipos

Vértice = 0..n
Se especifica este tipo suponiendo que los vértices del grafo pueden ser etiquetados con un número entero entre 1 y n.

Adyacencias = **array** [1 .. n, 1 .. n] **of logical**
Matriz de adyacencias que representa el grafo. Si A es una instancia del tipo Adyacencias entonces cada A[i, j] ($1 \leq i \leq n$, $1 \leq j \leq n$) toma alguno de los siguientes valores:

- *true* si existe un arco entre los vértices i y j.
- *false* si no existe un arco entre los vértices i y j.

Visitado = **array** [1 .. n] **of logical**
Arreglo que indica si los vértices han sido visitados o no. Si V es una instancia del tipo Visitado entonces cada V[i] ($1 \leq i \leq n$) toma alguno de los siguientes valores:

- *true* si el vértice i ya ha sido visitado
- *false* si el vértice i no ha sido visitado aún.

Camino = **array** [1 .. n] **of** Vértice
Arreglo en el que se almacena el camino que se construye.

Variables

vi, vf : Vértice;
ady : Adyacencias;
vis : Visitado;
c : Camino;
t : 1..n;

- Adaptar el esquema general a utilizar al problema particular.

```
proc buscar_todos_los_caminos(v : Vértice)
{
   var
      k : Vértice;

   k ← 0;
   while (k ≠ n) do
   {
      k ← k + 1;
      if visitable(v,k) then
      {
         agregar_vértice(k);
         if (k = vf) then
            escribir_camino(c)
         else
            buscar_todos_los_caminos(k);
         eliminar_vértice(k)
      }
   }
}
```

- Definir las funciones y procedimientos necesarios.

```
func visitable(v, w : Vértice) : logical
{
   return(ady[v,w] ∧ ¬vis[w])
}


proc agregar_vértice(v : Vértice)
{
   c[t] ← v;
   vis[v] ← true;
   t ← t + 1
}


proc eliminar_vértice(v : Vértice)
{
   t ← t - 1;
   c[t] ← 0;
   vis[v] ← false
}
```

- Elaborar el algoritmo principal.

```
proc todos_los_caminos_principal()
{
   const
      n = ?;
   type
      Vértice = 0..n;
      Adyacencias = array [1..n,1..n] of logical;
      Visitado = array [1..n] of logical;
      Camino = array [1..n] of Vértice;
   var
      vi, vf : Vértice;
      ady : Adyacencias;
      vis : Visitado;
      c : Camino;
      t : 1..n;
      i, j : integer;
      encontrado : logical;

   read(vi,vf);
   for i in [1..n] do
      for j in [1..n] do
         read(ady[i,j]);

   for i in [1..n] do
```

```
      vis[i] ← false;
   c[1] ← vi;
   vis[vi] ← true;
   t ← 2;
   encontrado ← false;
   buscar_todos_los_caminos(vi)
}
```

Ejemplo: Búsqueda de una solución

Planteamiento:

Suponga que el problema es reformulado en los siguientes términos: Dado un grafo dirigido, encontrar algún camino entre dos vértices dados (vértice inicial y vértice final) de manera tal que los vértices contenidos en cada camino sean visitados una sola vez.

Solución

- Identificar los elementos principales en el problema.

Esquema a utilizar: Búsqueda de una solución (se pide encontrar un camino cualquiera entre dos vértices dados del grafo).
Paso: Partiendo de un vértice cualquiera *v*, visitar un vértice adyacente *w*.
Solución: Cuando el vértice visitado (*w*) coincide con el vértice final (*vf*).

- Especificar las constantes, tipos y variables requeridas.

Se utilizan las mismas constantes, tipos y variables que en el caso anterior. Adicionalmente se requieren las siguientes:

- encontrado : **logical**

Variable de tipo lógico que indica si ya se encontró un camino

- Adaptar el esquema general a utilizar al problema particular.

```
proc buscar_un_camino(v : Vértice)
{
   var
      k : Vértice;

   k ← 0;
   while (k ≠ n) ∧ (¬encontrado) do
   {
      k ← k + 1;
      if visitable(v,k) then
      {
         agregar_vértice(k);
         if (k = vf) then
            encontrado ← true;
            escribir_camino(c)
```

```
         else
         {
            buscar_un_camino(k);
            if (¬encontrado) then
               eliminar_vértice(k)
         }
      }
   }
}
```

- Definir las funciones y procedimientos necesarios

Se mantienen las definiciones de las funciones y procedimientos utilizados en el caso anterior.

- Elaborar el algoritmo principal.

```
proc un_camino_principal()
{
   const
      n = ?;
   type
      Vértice = 0..n;
      Adyacencias = array [1..n,1..n] of logical;
      Visitado = array [1..n] of logical;
      Camino = array [1..n] of Vértice;
   var
      vi, vf : Vértice;
      ady : Adyacencias;
      vis : Visitado;
      c : Camino;
      t : 1..n;
      i, j : integer;
      encontrado : logical;

   read(vi,vf);
   for i in [1..n] do
      for j in [1..n] do
         read(ady[i,j]);

   for i in [1..n] do
      vis[i] ← false;
   c[1] ← vi;
   vis[vi] ← true;
   t ← 2;
   encontrado ← false;
   buscar_un_camino(vi)
}
```

Ejemplo: Búsqueda de la solución óptima

Planteamiento:

Suponga que el problema es reformulado en los siguientes términos: Dado un grafo dirigido y el costo de recorrer un arco (igual para cualquier arco del grafo), encontrar el camino de menor costo entre dos vértices dados (vértice inicial y vértice final) de manera tal que los vértices contenidos en cada camino sean visitados una sola vez.

Solución:

- Identificar los elementos principales en el problema.

Esquema a utilizar: Búsqueda de la solución óptima (en este caso se tiene la función de costo asociada a un camino, la cual debe ser minimizada y el objetivo es encontrar el camino para el cual el valor de la función es el mínimo).
Paso: Partiendo de un vértice cualquiera *v*, visitar un vértice adyacente *w*.
Solución: Cuando el vértice visitado (*w*) coincide con el vértice final (*vf*).

- Especificar las constantes, tipos y variables requeridas.

Se utilizan las mismas constantes, tipos y variables que en el primer caso. Adicionalmente se requieren las siguientes:

- costo : **float**

Variable que representa el costo de recorrer un arco. Del enunciado se tiene que el costo de recorrer un arco es el mismo para cualquier arco del grafo.

- costo_actual : **float**

Variable que representa el costo de recorrer el camino actual.

- costo_óptimo : **float**

Variable que representa el costo asociado al camino de menor costo conseguido hasta el momento.

- c_óptimo : **float**

Arreglo de vértices en el cual se almacena el camino cuyo costo es costo_óptimo.

- Adaptar el esquema general a utilizar al problema particular.

```
proc buscar_camino_óptimo(v : Vértice)
{
     var
          k : Vértice;

     k ← 0;
     while (k  ≠ n) do
     {
          k ← k + 1;
          if visitable(v,k) then
               agregar_vértice(k);
               if (k = vf) then
```

```
                    if (costo_actual < costo_óptimo) then
                        sustituir_camino;
                else
                    buscar_camino_óptimo(k);
                eliminar_vértice(k);
            endif
        }
}
```

- Definir las funciones y procedimientos necesarios.

Las definiciones de las funciones y procedimientos utilizados en el primer caso se modifican de la siguiente manera:

```
proc agregar_vértice(v : Vértice)
{
    c[t] ← v;
    costo_actual ← costo_actual + costo;
    vis[v] ← true;
    t ← t + 1;
}

proc eliminar_vértice(v : Vértice)
{
   t ← t - 1;
   c[t] ← 0;
   costo_actual ← costo_actual - costo;
   vis[v] ← false
}

proc sustituir_camino()
{
    costo_óptimo ← costo_actual;
    for i in [1..(t-1)] do
        c_óptimo[i] ← c[i]
}
```

- Elaborar el algoritmo principal.

```
proc camino_óptimo_principal()
{
    const
        n = ?
    type
        Vértice = [0..n];
        Adyacencias = array [1..n,1..n] of logical;
        Visitado = array [1..n] of logical;
        Camino = array [1..n] of Vértice;
    var
        vi, vf : Vértice;
        ady : Adyacencias;
```

```
        vis : Visitado;
        c, c_óptimo : Camino;
        t : 1..n;
        i, j : integer;
        costo, costo_actual, costo_óptimo : float;

    read(vi, vf, costo);

    for i in [1 .. n] do
        for j in 1 .. n do
            read(ady[i, j]);
    for i in [1 .. n] do
        vis[i] ← false;
    c[1] ← vi;
    vis[vi] ← true;
    t ← 2;
    costo_actual ← 0;
    costo_óptimo ← n * costo;
    buscar_camino_óptimo(vi)
}
```