Prácticas de Algorítmica

Práctica 3: Ramificación y poda
(parte 2, TSP)

Curso 2022-2023

Objetivos de la práctica

Recordemos que la práctica 3 consta de 4 sesiones:

- Primera parte: 1 sola sesión. Vió el uso de algoritmos voraces en la resolución del **problema del ensamblaje** utilizando la técnica de ramificación y poda (RyP).
- Seguna parte: 3 sesiones. Vamos a ver el problema del **viajante de comercio** (*Traveling Salesman Problem* o TSP) usando también RyP.

En esta segunda parte:

- Veremos la versión *asimétrica* del TSP: el caso de grafos dirigidos.
- Se implementarán implementaremos varias cotas optimistas, algunas de ellas descritas en la sección 9.5 de los apuntes de teoría.

El viajante de comercio (TSP)

- Un camino se representa mediante una secuencia de vértices.
- En el caso de un ciclo hamiltoniano el último vértice es el único que se repite en el camino al aparecer también al final.
- Sin pérdida de generalidad, podemos elegir *cualquier* vértice para empezar el ciclo. Al utilizar grafos con vértices numerados consecutivamente entre `0` y `nV-1`, el vértice inicial será el `0`.
- Modelamos los estados como prefijos de un camino completo, y se implementarán mediante una lista Python `[` $v_0, v_1, \ldots, v_{k-1}$ `]`.
- Un estado será terminal si su longitud es `nV`, pero eso es un camino que no cierra el ciclo. No todo estado terminal es factible.
- Un estado terminal es factible si existe una arista que una el último vértice con el primero.
- Cuando se llega a un estado terminal y factible, el coste asociado a dicho estado es la parte conocida incluyendo el coste de la arista que une el último con el primero.

Cotas de los apuntes de teoría

En la sección 9.5.2 de los apuntes se proponen las siguientes cotas (ver dicha sección para más detalles):

1. Cota trivial: sólo la parte conocida. Es demasiado optimista.
2. La parte desconocida se obtiene sumando el nº aristas necesarias. Se toman las aristas de menor coste estén usadas o no, sean de vértices visitados o no.
3. Refina la 2 usando únicamente aristas no visitadas.
4. Sólo considera las aristas de mínimo coste que parten de cada vértice no visitado (más el último visitado).
5. Las aristas de mínimo coste que parten y llegan a vértices no visitados. Es más cara de calcular al no admitir un preproceso.

Cotas de los apuntes de teoría

6. Completar el ciclo utilizando un camino. Dicho camino no necesita pasar por todos los vértices y, de hecho, puede pasar por vértices ya visitados. Tras un preproceso se puede calcular incrementalmente.
7. El camino no debe tocar vértices de la parte visitada/conocida.
8. La parte desconocida se calcula con el árbol de expansión de coste mínimo aplicado a los vértices desconocidos más el primero y último de la parte conocida.

Cotas propuestas para la práctica

En esta práctica vamos a plantear las siguientes cotas:

1. Cota trivial: sólo la parte conocida. **Ya implementada.**
2. *(nos la saltamos)*
3. *(nos la saltamos)*
4. Sólo considera las aristas de mínimo coste que parten de cada vértice no visitado (más el último visitado..
5. Las aristas de mínimo coste que parten y llegan a vértices no visitados. Es más cara de calcular al no admitir un preproceso.
6. Completar el ciclo utilizando un camino. Dicho camino no necesita pasar por todos los vértices y, de hecho, puede pasar por vértices ya visitados. Tras un preproceso se puede calcular incrementalmente.
7. El camino no debe tocar vértices de la parte visitada/conocida.
8. *(nos la saltamos)*

Código proporcionado para la práctica

Clase `DiGraph`

Clase Python para representar un grafo dirigido mediante lista de adyacencia. Nos guardamos las aristas que salen en `self.forward` pero también las que llegan en `self.backwards` (es redundante pero facilita el uso de Dijkstra en las últimas cotas). Un fragmento:

```
class DiGraph:
    def __init__(self):
        self.forward = {} # aristas que salen de cada vértice
        self.backwards = {} # aristas que llegan a cada vértice

    def add_edge(self, u, v, weight=1):
        self.add_node(u) # just in case
        self.add_node(v) # just in case
        self.forward[u].append((v,weight))
        self.backwards[v].append((u,weight))

    def nV(self): # |V| nº vértices
        return len(self.forward)
```

Clase `DiGraph`

Hemos añadido algunos métodos útiles como:

- `nodes(self)` para iterar sobre la lista de vértices.
- `edges_from(self,u)` para iterar sobre los pares destino,peso que salen de `u`.
- `is_edge(self, u, v)` nos permite saber si hay una arista entre `u` y `v`.
- `weight_edge(self, u, v)` nos da el coste de la arista entre `u` y `v`.
- `lowest_out_weight(self, vertex, forbidden=())` devuelve el coste de la arista más barata que alga de `vertex` y no llegue a ninguno de los vèrtices de `forbidden`.
- `check_TSP(self, cycle)` para comprobar que un ciclo devuelto por el algoritmo es correcto y calcular su coste.
- `path_weight(self, path)` devuelve el coste del camino `path`.

Clase `DiGraph`

- `Dijkstra(self, src, reverse=False)` aplica el algoritmo de Dijkstra para calcular el camino más corto desde `src`, pero si ponemos `reverse=True` será el camino más corto desde cada vértice hast `src`.
- `Dijkstra1dst(self, src, dst, avoid=())` versión que sólo devuelve la distancia entre `src` y `dst` evitando pasar por vértices del conjunto `avoid`.
- `StronglyCC(self)` calcula las componentes fuertemente conexas del grafo en tiempo lineal (algoritmo de Tarjan). Si el grafo tiene más de una componente fuertemente conexa no tendrá ciclo Hamiltoniano y no valdrá la pena intentar resolver el TSP con ramificación y poda. Es fácil ver que es condición necesaria pero no suficiente (un grafo puede tener una sola componente fuertemente conexa pero tener que pasar más de una vez por un mismo vértice para pasar por todos).

Clase `DiGraph`

Tenemos algunas funciones auxiliares:

- `generate_random_digraph(nV, dimacsList=None, seed=None)` genera un grafo dirigido aleatorio con pesos enteros. Este grafo calcula los pesos asignando una coordenada a cada vértice y multiplicando la distancia euclídea por un factor corrector (entre 1 y 1.5) para simular que las carreteras normalmente no van en líneas rectas (rodeos por accidentes geográficos, etc.). Permite guardar el grafo en un formato DIMACS (ver abajo). Admite un parámetro `seed` que permite controlar la generación de valores pseudo-aleatorios para facilitar la reproducibilidad de los experimentos pasándole un número entero.
- `generate_random_digraph_1scc` llama al generador anterior hasta asegurarse de generar un grafo con una sola componente fuertemente conexa (condición necesaria pero no suficietne para la existencia de solución en TSP).
- `load_dimacs_graph` carga un grafo en formato DIMACS (ver enlace).

Clase `BranchBoundImplicit`

Implementa el método `solve` de RyP con poda implícita:

```
def solve(self):
  A = [ self.initial_solution() ] # cola de prioridad
  while len(A)>0 and A[0][0] < self.fx:
    s_score, s = heapq.heappop(A)
    for child_score, child in self.branch(s_score, s):
      if self.is_complete(child): # si es terminal
        if self.is_factible(child):
          child_score, child = self.get_factible_state(child_score,
                                                        child)
          if child_score < self.fx:
            self.fx, self.x = child_score, child


      else: # no es terminal
        if child_score < self.fx:
          heapq.heappush(A, (child_score, child) )
  return self.fx, self.x, stats
```

Clase `BranchBoundExplicit`

Implementa el método `solve` de RyP con poda explícita:

```
def solve(self):
  A = [ self.initial_solution() ] # cola de prioridad
  while len(A)>0:
    s_score, s = heapq.heappop(A)
    for child_score, child in self.branch(s_score, s):
      if self.is_complete(child): # si es terminal
        if self.is_factible(child):
          child_score, child = self.get_factible_state(child_score,
                                                       child)
          if child_score < self.fx:
            self.fx, self.x = child_score, child
            A = [(scr,st) for (scr,st) in A  if scr < child_score]
            heapq.heapify(A) # transform into a heap,
      else: # no es terminal
        if child_score < self.fx:
          heapq.heappush(A, (child_score, child) )
  return self.fx, self.x, stats
```

Clase `TSP`

Las clases `BranchBoundImplicit` y `BranchBoundExplicit` están pensadas para utilizar la herencia y combinarlas con otras clases que aporten los métodos faltantes. Python permite utilizar herencia múltiple y combinar varias clases además de derivar de ellas para añadir los métodos necesarios. La clase `TSP` proporciona la parte básica que necesitamos para implementar el viajante de comercio:

```python
class TSP:
    def __init__(self, graph, first_vertex=0):
        self.G = graph # DiGraph
        self.first_vertex = first_vertex
        self.fx = float('inf')
        self.x = None

    def is_complete(self, s):
        '''
        s es una solución parcial
        '''
        return len(s) == self.G.nV()
```

Clase TSP

```python
def is_factible(self, s):
    '''
    asumimos s is complete, falta ver si hay una arista desde
    el último vértice s[-1] hasta el primero s[0]
    '''
    return self.G.is_edge(s[-1],s[0])

def get_factible_state(self, s_score, s):
    '''
    En el caso del TSP se encarga de añadir la arista que
    vuelve al origen Dependiendo del tipo de cota optimista,
    hay que corregir el valor de la parte conocida y
    desconocida, en el caso general no nos la jugamos (mejor
    optimizar para casos particulares):
    '''
    s_new = s+[s[0]]
    return self.G.path_weight(s_new), s_new
```

Clase `TSP_Cota1`

La idea es utilizar `TSP` para, vía herencia, crear otra clase que añada métodos como `branch`. Veamos el caso mś sencillo de la cota trivial:

```
class TSP_Cota1(TSP):
    'Versión con cota trivial. La parte desconocida vale 0'
    def initial_solution(self):
        initial = [ self.first_vertex ]
        initial_score = 0
        return (initial_score, initial)
    def branch(self, s_score, s):
        's_score es el score de s'
        lastvertex = s[-1]
        for v,w in self.G.edges_from(lastvertex):
            if v not in s:
                yield (s_score + w, s+[v])
```

Observación

Hemos implementado la cota optimista dentro de `branch` tras inicializar el estado inicial en `initial_solution`. La alternativa es un método cota, pero eso hace menos eficiente la implementación de ciertas cotas.

Clases derivadas

```python
class TSP_Cota1I(TSP_Cota1, BranchBoundImplicit):
    pass

class TSP_Cota1E(TSP_Cota1, BranchBoundExplicit):
    pass

class TSP_Cota4I(TSP_Cota4, BranchBoundImplicit):
    pass

class TSP_Cota4E(TSP_Cota4, BranchBoundExplicit):
    pass

class TSP_Cota5I(TSP_Cota5, BranchBoundImplicit):
    pass

class TSP_Cota5E(TSP_Cota5, BranchBoundExplicit):
    pass
```

$\vdots$

Clases derivadas

Esta variable es útil para los experimentos:

```
repertorio_cotas = [('Cota1I',TSP_Cota1I),
                    ('Cota1E',TSP_Cota1E),
                    ('Cota4I',TSP_Cota4I),
                    ('Cota4E',TSP_Cota4E),
                    ('Cota5I',TSP_Cota5I),
                    ('Cota5E',TSP_Cota5E),
                    ('Cota6I',TSP_Cota6I),
                    ('Cota6E',TSP_Cota6E),
                    ('Cota7I',TSP_Cota7I),
                    ('Cota7E',TSP_Cota7E)]
```

Probando el código

Prueba mínima con el grafo de ejemplo de los apuntes de teoría:

```
def prueba_mini():
    g = load_dimacs_graph(ejemplo_teoria,True)
    for nombre,clase in repertorio_cotas:
        print(f'----- checking {nombre} -----')
        tspi = clase(g)
        fx,x,stats = tspi.solve()
        print(fx,x,stats)
```

Probando el código

Prueba más compleja con el generador de grafos aleatorios:

```python
def prueba_generador():
    for nV in range(5,1+max(tallaMax.values())):
        print("-"*80)
        print(f"Probando grafos de talla {nV}")
        g = generate_random_digraph_1scc(nV)
        for nombre,clase in repertorio_cotas:
            if nV <= tallaMax[nombre]:
                print(f'\n  checking {nombre}')
                tspi = clase(g)
                fx,x,stats = tspi.solve()
                print(' ',fx,x,stats)
                if x is not None:
                    print(' ',g.check_TSP(x))
```

Actividades

Completar las cotas

Las clases para completar las cotas heredan todas de TSP y deben implementar estos métodos:

- El constructor **solamente** en caso de necesitar añadir nuevos atributos. En ese caso se debe llamar primero al constructor de la clase padre (`TSP`). Ejemplo:

```python
def __init__(self, graph, first_vertex=0):
    super().__init__(graph, first_vertex)
    # self.atributo = ...
```

- `initial_solution` devuelve la solución inicial **pero** en cada caso el cálculo de su cota optimista puede variar. Especialmente cierto en las cotas que se calculan de manera incremental donde la cota del estado inicial se calcula de forma distinta al resto de casos.
- `branch` devuelve los hijos acopañados por su *score*. Es aquí donde se implementa la lógica del cálculo de las cotas. En los casos de cota incremental tenemos a nuestra disposición la cota optimista del estado padre.

Experimentación para analizar el comportamiento

Se trata de completar la siguiente función, ejecutarla y analizar los resultados obtenidos:

```
def experimento():
    '''
    Probar con varias tallas entre 10 y 20
    Para cada talla generar entre 5 y 20 instancias (según lo lento
    que funcione)
    IMPORTANTE probar diversos algoritmos con LAS MISMAS instancias
    Descartar las que no den solución (las que sacan None,float('
     inf'))
    Mostrar valores medios de todas las instancias de cada talla.
    Sacar los datos de manera que sea fácil realizar una
     interpretacion
    '''
```

Experimentación para analizar el comportamiento

Recordamos que las funciones `solve` calculan y devuelven estadísticas:

```
def solve(self):
    ...
    return self.fx, self.x, stats
```

Donde `stats` es un diccionario con estos campos:

- `time`: el tiempo de ejecución.
- `iterations`: nº iteraciones.
- `gen_states`: nº estados generados (en `branch`).
- `podas_opt`: nº estados podados por cota optimista.
- `maxA`: tamaño máximo alcanzado por el cjt estados activos a lo largo del algoritmo.
- `cost_expl`: tamaño acumulado de los conjuntos de estados activos todas las veces que se ha realizado poda explícita (este campo no se calcula para la poda implícita).