EI1022, Algoritmia

Problemas 1
Recorridos de grafos

Contenido

Introducción

Vamos a utilizar recorridos de grafos con dos objetivos:

- Encontrar el camino más corto entre dos celdas de un laberinto
- Estudiar qué celdas puede alcanzar un caballo en un tablero de ajedrez

El camino más corto en un laberinto

- Al interpretar un laberinto como un grafo no dirigido podemos encontrar el camino más corto entre dos vértices mediante un recorrido primero en anchura
- Vamos a escribir un programa, `shortest_path`, para resolver laberintos mediante un recorrido en anchura

Implementación

- Crea el fichero `shortest_path.py` y declara los tipos:

```
type Vertex = tuple[int, int]
type Edge = tuple[Vertex, Vertex]

type Data = tuple[int, int]
type Result = tuple[UndirectedGraph[Vertex],
                    list[Vertex]]
```

Implementación (2)

- A continuación, escribe la función

```
def bf_search(g: UndirectedGraph[Vertex],
              source: Vertex, target: Vertex) -> list[Edge]:
```

que hace un recorrido en anchura desde `source` y se detiene al encontrar `target`

Implementación (3)

- Añade una segunda función

```
def path_recover(edges: list[Edge],
                 target: Vertex) -> list[Vertex]:
```

que devuelve el camino hasta `target` usando el resultado de `bf_search` (puedes usar la de las traspas de teoría)

Implementación (4)

- Ahora vas a escribir las funciones `read_data`, `process` y `show_result`
- La función `read_data` devolverá dos enteros en la primera línea de la entrada: el número de filas (`rows`) y el número de columnas (`cols`) del laberinto
- Puedes usar `split()` para separar los dos números:

```
def read_data(f: TextIO) -> Data:
    return tuple(int(s) for s in f.readline().split())
```

Implementación (5)

- La función `process` generará un laberinto con el número de filas y columnas dado y encontrará el camino más corto desde `(0,0)` a `(rows-1, cols-1)`. Devolverá una tupla con el laberinto y el camino
- Para generar el laberinto puedes usar la función que escribiste en la última sesión de prácticas o puedes emplear la llamada `create_labyrinth(rows, cols)` del módulo `labyrinth.py` disponible en el aula virtual
- Para encontrar el camino más corto deberás usar las funciones `bf_search` y `path_recover`

Implementación (6)

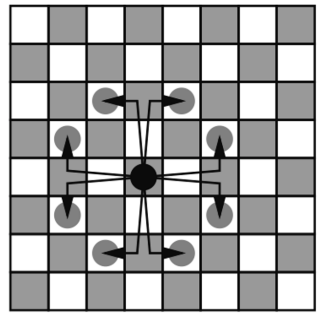
- La función `show_result` mostrará por la salida estándar el camino encontrado escribiendo una línea por cada celda
- Una vez escritas las funciones, escribe el programa principal que las llame

Prueba

- Para ver si el camino realmente es una solución, escribe el programa `shortest_path_viewer` que pregunte al usuario el número de filas y columnas, genere un laberinto y lo muestre por pantalla junto con la solución
- Genera el laberinto y la solución con la función `process` de `shortest_path`
- Para mostrar el laberinto utiliza el módulo `labyrinthviewer` como en la última práctica pero antes de llamar a la función `run` usa `add_path` para mostrar el camino
- Por ejemplo: si `lv` es un `LabyrinthViewer` y el camino más corto está en la variable `p`, añádelo con `lv.add_path(p)`

Recorridos del caballo de ajedrez

- El caballo de ajedrez puede efectuar los siguientes movimientos:



- En un tablero de 8x8, un caballo puede ir de cualquier casilla a cualquiera otra, sin embargo, en uno de 3x3, no es posible
- Vas a implementar un programa que permita ver qué casillas puede alcanzar un caballo en un tablero de tamaño arbitrario

El tablero como grafo

- Podemos interpretar un tablero como un grafo no dirigido en el que:
  - Los vértices son las casillas del tablero representas mediante tuplas `(fila, columna)`
  - Una arista entre dos vértices significa que el caballo puede ir de uno a otro en un movimiento

El programa

- Crea el fichero `knight.py` y declara los tipos:

```
type Vertex = tuple[int, int]
type Data = tuple[int, int, int, int]
type Result = tuple[UndirectedGraph[Vertex], int]
```

- Implementa la función auxiliar

```
def knight_graph(rows: int, cols: int) -> UndirectedGraph[Vertex]:
```

que devuelva el grafo correspondiente a un tablero con `rows` filas y `cols` columnas

El programa (2)

- El tipo `Data` es:

```
type Data = tuple[int, int, int, int]
```

- Así, la función `read_data` leerá cuatro números enteros en dos líneas:
  - En la primera línea: `rows` y `cols`, las dimensiones del tablero
  - En la segunda línea: `first_row` y `first_col`, la posición inicial del caballo

El programa (3)

- El tipo `Result` es:

```
type Result = tuple[UndirectedGraph[Vertex], int]
```

- La función `process` generará el grafo correspondiente al tablero, lo recorrerá desde `(first_row, first_col)` y devolverá una tupla con el grafo y el número de casillas alcanzadas en el recorrido
- Usa `bf_search` para recorrer el grafo
- La función `show_result` mostrará el número devuelto por `process`

Visualización del grafo

- Para entender los resultados, puede ser útil visualizar el grafo que se construye
- Haz un programa, `knight_viewer.py`, para ver el grafo correspondiente a un tablero.
- Los tipos serán:

```
type Vertex = tuple[int, int]
type Data = tuple[int, int]
type Result = UndirectedGraph[Vertex]
```

- Y las funciones:
  - `read_data`: lee `rows` y `cols` en una línea
  - `process`: genera el grafo del tablero
  - `show_result`: visualiza el grafo usando `Graph2dViewer`

Visualización del grafo (2)

- Para escribir `show_result`, usaremos la clase `Graph2dViewer` del módulo `algoritmia.viewers.graph2d_viewer` que permite visualizar grafos cuyos vértices son puntos en el plano
- Con `vertexmode` se especifica qué son los puntos:
  - `Graph2dViewer.X_Y`: por defecto, los puntos son pares (x, y)
  - `Graph2dViewer.ROW_COL`: los puntos son pares (row, col)
- Para los tableros usamos el segundo modo:

```
def show_result(result: Result):
    gv = Graph2dViewer(result,
                       vertexmode=Graph2dViewer.ROW_COL)
    gv.run()
```

Pruebas

Usa tu programa para responder a estas preguntas:

- ¿Cuántas casillas podemos alcanzar en cero o más movimientos en un tablero de 8x8 si el caballo parte de la posición `(0,3)`?
- ¿Puede un caballo alcanzar cualquier casilla del tablero de 8x8 desde cualquier otra?
- Partiendo de la posición `(0,0)`:
  - ¿Cuántas casillas podemos alcanzar en cero o más movimientos en un tablero de 2x10?
  - ¿Y en uno de 3x3?

Adicional 1: Camino de la búsqueda en profundidad

- Si sabemos que en nuestro laberinto hay solo un camino entre dos vértices, también podemos utilizar un recorrido primero en profundidad, ¿por qué?
- Copia el fichero `shortest_path.py` en `df_path.py`
- Añade la función

```
def df_search(g: UndirectedGraph[Vertex],
              source: Vertex,
              target: Vertex) -> list[Edge]:
```

que hace un recorrido primero en profundidad desde `source` y que se detiene en el momento en que encuentra `target`

Prueba del nuevo recorrido

- Para entender las diferencias entre el recorrido en anchura y en profundidad, vamos a ver cómo se comportan ante un laberinto que tiene más de un camino entre dos celdas
- Para permitir más de un camino entre dos celdas, podemos modificar el algoritmo de generación de laberintos de modo que se añadan a los pasillos generados normalmente un número adicional de pasillos aleatorios
- Modifica `Data` para que tenga tres componentes:

```
type Data = tuple[int, int, int]
```

- Y modifica la función `read_data` para que lea y devuelva los enteros `rows` y `cols` de la primera línea y `additional` (el número de pasillos adicionales) de la segunda

Prueba del nuevo recorrido (2)

- Modifica la función `process` para que encuentre dos caminos: uno usando `df_search` y otro con `bf_search`
- Para generar el laberinto puedes modificar tu generador o aprovechar que el generador de `labyrinth.py` tiene un tercer parámetro opcional con el número de pasillos adicionales
- La tupla devuelta por `process` ahora tendrá tres componentes:

```
type Result = tuple[UndirectedGraph[Vertex],
                    list[Vertex],
                    list[Vertex]]
```

Prueba del nuevo recorrido (3)

- Modifica la función `show_result` para que muestre ambos caminos en el laberinto separados por una línea en blanco
- Copia `shortest_path_viewer.py` en `df_path_viewer.py` y modifícalo para que muestre los dos recorridos
- Para que se vean separados, puedes usar los parámetros adicionales de `add_path`
- Por ejemplo, si el segundo camino está en la variable `path_df`, con

  ```
  lv.add_path(path_df, offset=2, color="blue")
  ```

  se mostrará el segundo camino desplazado dos pixeles y en color azul

Adicional 2: Precálculo de caminos

- Escribe un programa, `knight_matrix`, cuya entrada sean dos números (filas y columnas) y muestre por pantalla una matriz `m` tal que `m[r][c]` sea el número de saltos que debe dar el caballo para llegar hasta la celda `(r, c)` desde la `(0, 0)`. Si no se puede llegar a la celda, `m[r][c]` será `-1`

- Los tipos serán:

```
type Vertex = tuple[int, int]
type Matrix = list[list[int]]
type Data = tuple[int, int]
type Result = Matrix
```

- La función `show_result` mostrará la matriz como una serie de filas de números separados por espacios

Adicional 2: Precálculo de caminos (2)

- Esta es la matriz correspondiente a un tablero de 3x4:

```
0  3  2  5
3  4  1  2
2  1  4  3
```

- Y esta la de un tablero de 3x3 (observa la casilla central):

```
0  3  2
3 -1  1
2  1  4
```