1 Tema Nr. 10: Căutare în adâncime (DFS)

Tema Nr. 10: Căutare în adâncime (DFS)
Timp Alocat: 2 ore

1.1 Implementare

Se cere implementarea corectă și eficientă a algoritmului de căutare în adâncime (Depth-First Search - DFS) (*Capitolul 22.3 din [1]*). Pentru reprezentarea grafurilor, va trebui să folosești liste de adiacență. De asemenea, va trebui să:

- Implementați sortarea topologică (*Capitolul 22.4* din *[1]*)
- Implementați algoritmul Tarjan pentru componente tare conexe

1.2 Cerințe minimale pentru notare

Lipsa oricărei cerințe minimale (chiar și parțială) poate rezulta într-o notă mai mică prin penalizări sau refuzul de a prelua tema, rezultând în nota 0.

- *Demo*: Pregătiți un exemplu pentru exemplificarea corectitudinii fiecărui algoritm implementat. Corectitudinea fiecărui algoritm se demonstrează printr-un exemplu simplu (maxim 10 valori).
- Graficele create trebuie să fie ușor de evaluat, adică grupate și adunate prin funcțiile Profiler după cerințele temei. Tema nu va fi evaluată dacă conține o multitudine de grafice negrupate. De exemplu, analiza comparativă implică gruparea într-un singur grafic a algoritmilor comparați.
- Interpretați graficul/graficele și notați observațiile personale în antetul fișierului *main.cpp*, într-un comentariu bloc informativ.
- Nu preluăm teme care nu sunt indentate și care nu sunt organizate în funcții (de exemplu, nu prelăum teme unde tot codul este pus în main).
- *Punctajele din barem sunt corespondente unei rezolvări corecte și complete a cerinței, calitatea interpretărilor din comentariul bloc și* ***răspunsul corect dat de dumeavostră la întrebările puse de către profesor.***

1.3 Cerințe

1.3.1 DFS (5p)

Demo: Demonstrați corectitudinea algoritmului pe un graf de dimensiune mică:

- afișați graful inițial (liste de adiacență)
- afișați arborele rezultat în urma DFS

1.3.2 Sortare topologică (1p)

Demo: Demonstrați corectitudinea algoritmului pe un graf de dimensiune mică:

- afișați graful inițial (liste de adiacență)
- afișați listă de noduri sortate topologic (dacă are / dacă nu are de ce nu are?)

1.3.3 Tarjan (2p)

Demo: Demonstrați corectitudinea algoritmului pe un graf de dimensiune mică:

- afișați graful inițial (liste de adiacență)
- afișați componentele puternic conexe ale grafului

Prezentăm algoritmul lui Tarjan pentru identificarea componentelor tare conexe (SCC) într-un graf orientat. Pentru context și mai multe detalii, consultați articolul de pe Wikipedia: Algoritmul lui Tarjan pentru componentele tare conexe. Pseudocodul poate fi găsit și aici.

1.3.4 Analiza performanței pentru DFS (2p)

Cum timpul de execuție al algoritmului DFS variază în funcție de numărul de vârfuri (|V|) și de numărul de muchii (|E|) aveți de făcut următoarele analize:

1. Fixați |V|=100 și variați |E| între 1000 și 4500 cu un pas de 100. Generați pentru fiecare caz un graf aleator și asigurați-vă că nu generați aceeași muchie de 2 ori. Execută DFS pentru fiecare graf generat și numără operațiile efectuate. Apoi construiește graficul cu variația numărului de operații în funcție de |E|;
2. Fixați |E|=4500 și variați |V| între 100 și 200 cu un pas de 10. Repetă procedura de mai sus și construiește graficul cu variația numărului de operații în funcție de |V|.

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.

Algorithm 1 Tarjan's strongly connected components

```
1: procedure TARJAN(G = (V, E))
2:     Input: graph G = (V, E)
3:     Output: set of strongly connected components (sets of vertices)
4:     index ← 0
5:     S ← empty stack
6:     for all v ∈ V do
7:         if v.index is undefined then
8:             STRONGCONNECT(v)
9:         end if
10:    end for
11: end procedure

12: function STRONGCONNECT(v)
13:    // Set the depth index for v to the smallest unused index
14:    v.index ← index
15:    v.lowlink ← index
16:    index ← index + 1
17:    S.push(v)
18:    v.onStack ← true
19:    // Consider successors of v
20:    for all (v, w) ∈ E do
21:        if w.index is undefined then
22:            // Successor w has not yet been visited; recurse on it
23:            STRONGCONNECT(w)
24:            v.lowlink ← min(v.lowlink, w.lowlink)
25:        else if w.onStack then
26:            // Successor w is in stack S and hence in the current SCC
27:            // If w is not on stack, then (v, w) is an edge pointing to
   an SCC already
28:            // found and must be ignored
29:            // See below regarding the next line
30:            v.lowlink ← min(v.lowlink, w.index)
31:        end if
32:    end for
33:    // If v is a root node, pop the stack and generate an SCC
34:    if v.lowlink = v.index then
35:        start a new strongly connected component
36:        repeat
37:            w ← S.pop()
38:            w.onStack ← false
39:            add w to current strongly connected component
40:        until w = v
41:        output the current strongly connected component
42:    end if
43: end function
```