1 Tema Nr. 9: Căutarea în lățime (BFS)

1.1 Introducere

În acest laborator se va implementa algoritmul BFS (căutarea în lățime), conform secțiunii 22.2 din [1].

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 Structura proiectului

La acest laborator veți porni de la 3 fișiere:

- `main.cpp` - sursa principală, responsabilă cu interfața de vizualizare
- `bfs.h` - definiții pentru tipuri de structuri și funcții
- `bfs.cpp` - implementarea algoritmilor
- `grid.txt` - labirintul care va fi afișat și traversat
- `Profiler.h` - biblioteca pentru numărarea operațiilor și pentru grafice

!!! Pentru rezolvarea cerințelor trebuie să faceți modificări doar în `bfs.cpp`.

Pentru a vizualiza mai ușor funcționarea algoritmului, `main.cpp` va afișa o interfață de tip text, în care se va vedea o un labirint ce trebuie traversat (celulele negre sunt libere iar cele albe sunt pereți).

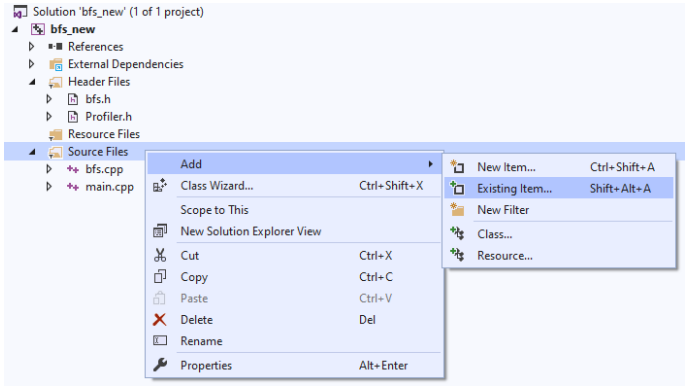

Figure 1: Fereastra *"Solution Explorer"* în Visual Studio

1.3.1 Inițializarea proiectului pe Windows, cu Visual Studio

Creați un proiect nou în Visual Studio, de tipul *"Empty Project"* și copiați fișierele de mai sus în folderul proiectului.

Adăugați cele doua fișiere `.h` la secțiunea *"Header Files"* și cele două fișiere `.cpp` la secțiunea *"Source Files"*, efectuând click dreapta → *"Add"* → *"Existing Item"*, ca în Figura 1.

1.3.2 Inițializarea proiectului pe Linux și Mac

Puteți edita fișierele proiectului cu orice editor doriți. Proiectul conține și un fișier `Makefile`, deci este suficient să rulați comanda `make` pentru compilarea acestuia. Executabilul `.exe` rezultat se va numi `main`, și se poate rula în terminal, executând `./main`.

1.4 Funcționare

La pornirea programului se va afișa labirintul, în mod similar cu Figura 2.

În partea de jos, utilizatorul poate tasta una din comenzile de mai jos:

- `exit`
  terminarea programului
- `clear`
  curățarea informațiilor anterioare din grilă
- `neighb <row> <col>`
  se vor afișa vecinii celulei de pe linia `<row>` și coloana `<col>`.
- `bfs <row> <col>`
  se va efectua o parcurgere BFS, pornind de la celula de la linia `<row>` și coloana `<col>`.

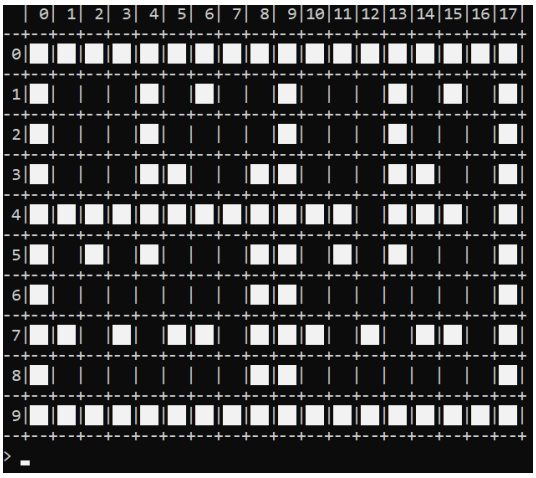

Figure 2: Interfața programului

- `bfs_step <row> <col>`
  la fel ca la `bfs`, dar rezultatul se va afișa pas cu pas, în funcție de distanța față de sursă

- `bfs_tree <row> <col>`
  la fel ca la `bfs`, dar se va afișa și arborele BFS sub grilă

- `path <row1> <col1> <row2> <col2>`
  se va afișa cel mai scurt drum între celulele `(<row1> <col1>)` și `(<row2> <col2>)`

- `perf`
  se vor genera graficele pentru evaluarea performanței algoritmului

1.4.1 Exemplu: comanda `neighb`

Dacă se rulează:
`neighb 2 3` ar trebui să apară imaginea din Figura 3.

Celula de start se va colora cu verde, iar vecinii acesteia cu albastru.

În momentul de față funcția `get_neighbors()` nu este implementată, deci nu se va afișa rezultatul dorit. Puteți verifica dacă ați implementat corect această funcție rulând comanda pentru diverse celule. Fiecare celulă va avea maxim 4 vecini (sus, jos, stânga, dreapta), și nu trebuie afișate celule din afara grid-ului sau celule care conțin pereți.

1.4.2 Exemplu: comenzile `bfs` și `bfs_step`

Dacă se rulează:
`bfs 6 3` ar trebui să apară imaginea din Figura 4.

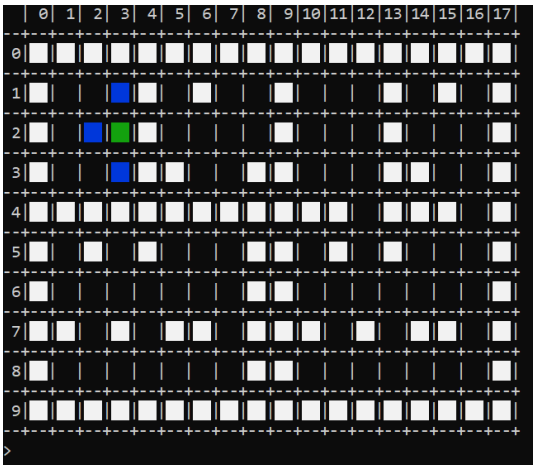

Figure 3: Rezultatul comenzii `neighb 2 3`

Celula de start se va colora cu verde, iar celulele parcurse se vor colora cu albastru. Pe fiecare celulă albastră va apărea o săgeată care va indica în ce direcție se află părintele din arborele BFS.

În momentul de față funcția `bfs()` nu este implementată, deci nu se va afișa rezultatul dorit. Puteți verifica dacă ați implementat corect această funcție rulând comanda pentru diverse celule.

1.4.3 Exemplu: comanda `bfs_tree`

Dacă se rulează:
`bfs 2 6` ar trebui să apară imaginea din Figura 5.

Rădăcina arborelui este nodul de start, respectiv `(2, 6)`. Copii acestui nod, sunt nodurile în care se poate ajunge direct din rădăcină: `(2, 5)`, `(2, 7)` și `(3, 6)` (ordinea acestora poate să difere în altă implementare).

1.4.4 Exemplu: comanda `path`

Dacă se rulează:
`path 5 10 3 15` ar trebui să apară imaginea din Figura 6.

Celula de start se va colora cu verde, cea de final cu roșu, iar celulele care fac parte din drumul cel mai scurt se vor colora cu albastru. Pe fiecare celulă albastră va apărea o săgeată care va indica direcția de mers.

În momentul de față funcțiile `shortest_path()` și `bfs()` nu sunt implementate, deci nu se va afișa rezultatul dorit. Puteți verifica dacă ați implementat corect aceste funcții rulând comanda pentru diverse perechi de celule.

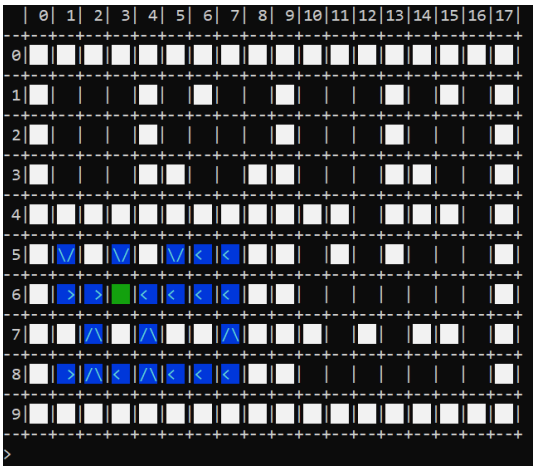

Figure 4: Rezultatul comenzii `bfs 6 3`

1.4.5 Structuri de date folosite

În fișierul `bfs.h` sunt definite câteva structuri de date utile în cadrul framework-ului.

Structura `Grid` modelează o grilă, formată din `rows` linii și `cols` coloane, elementele acesteia fiind în matricea `mat`. O celulă liberă va avea valoarea 0, iar una ce conține un perete va avea valoarea 1.

Structura `Point` modelează un punct sau o celulă din grilă, câmpurile `row` și `col` reprezentând linia și coloana la care se află.

Structura `Node` modelează un nod din graf și conține următoarele câmpuri:

- `position` de tip `Point` reprezintă celula din grilă corespunzătoare nodului.
- `adjSize` - numărul de vecini ai nodului respectiv
- `adj` - vectorul de vecini, de dimensiune `adjSize`
- `color` - culoarea nodului; la început toate nodurile au culoarea `COLOR_WHITE`, adică valoarea 0
- `dist` - distanța față de nodul de start, în parcurgerea BFS
- `parent` - pointer la nodul părinte, în arborele BFS

Structura `Graph` modelează un graf și conține numărul de noduri `nrNodes` și vectorul `v` cu pointeri spre acestea.

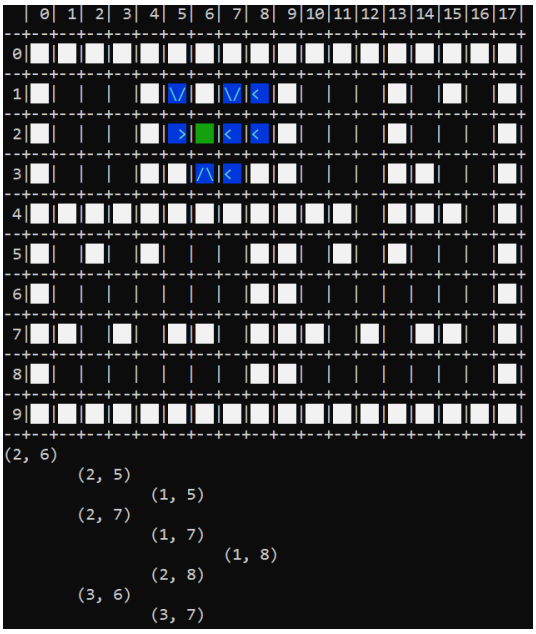

Figure 5: Rezultatul comenzii `bfs_tree 2 6`

1.5 Cerințe

1.5.1 Determinarea vecinilor unei celule (2p)

În `bfs.cpp`, trebuie completată funcția `get_neighbors()` care primește ca parametri un pointer la structura de tip `Grid`, un punct `p` de tip `Point` și un vector de puncte `neighb` care se va completa cu vecinii punctului `p`. Funcția va returna numărul de vecini completați în vectorul `neighb`.

Un punct din grilă va avea maxim 4 vecini (sus, jos, stânga, dreapta). Nu toți vecinii sunt valizi: unii vecini pot ajunge în afara grilei (coordonate negative, sau peste dimensiuni) sau pot fi în interiorul unui zid. Din acest motiv, după ce calculați poziția unui vecin, ar trebui să verificați că aceasta cade în interiorul grilei, apoi că aceasta este liberă (valoarea din matrice la poziția respectivă e 0).

Vecinii valizi se vor completa în vectorul `neighb`. Se garantează că la apelul funcției din framework, acesta va avea cel puțin 4 elemente, deci nu se poate depăși capacitatea acestuia. Deoarece numărul de vecini completați poate fi mai mic de 4, trebuie să returnăm numărul acestora.

1.5.2 Implementarea algoritmului BFS (3p)

În `bfs.cpp`, trebuie completată funcția `bfs()` care primește ca parametri un pointer la structura de tip `Graph` și nodul de start `s` de tip `Node*`. Funcția va aplica algoritmul BFS conform secțiunii 22.2 din [1].

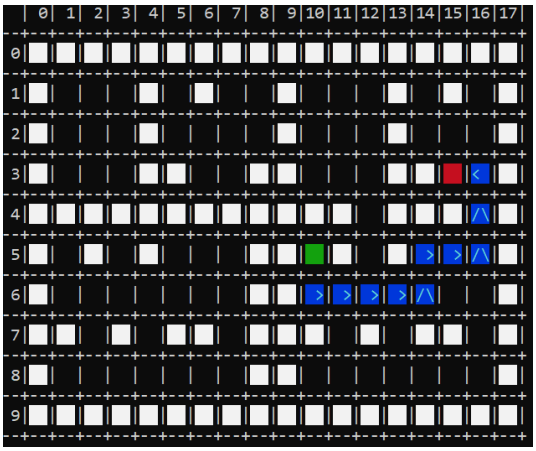

Figure 6: Rezultatul comenzii `path 5 10 3 15`

Nodurile din graf au la început culoarea `COLOR_WHITE`, iar câmpurile `dist` și `parent` sunt inițializate cu 0, respectiv `NULL`. După parcurgere, toate nodurile la care se poate ajunge din nodul de start trebuie să aibă culoarea `COLOR_BLACK`, distanța `dist` setată pe numărul de pași de la nodul de start până la nodul respectiv, iar pointerul `parent` trebuie să indice părintele în arborele BFS.

1.5.3 Afișarea arborelui BFS (2p)

În `bfs.cpp`, trebuie completată funcția `print_bfs_tree()` care primește ca parametru un pointer la structura de tip `Graph` pe care s-a rulat deja algoritmul BFS, deci culorile nodurilor și părinții sunt deja setate.

În funcția data, este deja implementată construcția unui vector de părinți `p`, în care nodurile colorate cu negru în parcurgerea BFS vor fi numerotate de la 0 la n. Deasemenea se construiește vectorul `repr` care conține coordonatele fiecărui nod.

Pentru afișarea acestui arbore se poate adapta codul din laboratorul de arbori multi-căi.

1.5.4 Evaluarea performanței algoritmului BFS (3p)

Funcția `performance()` realizează numărarea operațiilor, variind pe rând numărul de muchii, respectiv numărul de vârfuri al grafului. Pentru fiecare valoare, trebuie să implementați construcția unui graf aleator, conex, care să aibă numărul respectiv de vârfuri și de muchii.

În interiorul funcției `bfs()` va trebui să implementați numărarea propriu-zisă a operațiilor, folosind parametrul opțional `op`. Deoarece acest parametru este opțional, uneori funcția `bfs()` va fi apelată din framework cu valoarea acestuia setată pe `NULL`. Din acest motiv, atunci când numărați o operație, verificați tot timpul că `op` e un pointer valid, ca în exemplul de mai jos:

```
if(op != NULL) op->count();
```

1.5.5 Bonus: Determinarea celui mai scurt drum (0.5p)

În `bfs.cpp`, trebuie completată funcția `shortest_path()` care primește ca parametri un pointer la structura de tip `Graph`, nodurile de început și sfârșit `start` și `end` de tip `Node*`, respectiv vectorul `path`, ca parametru de ieșire în care se vor completa nodurile de pe traseu, în ordine. Funcția va returna numărul de noduri completate în vectorul `path`.

Pentru determinarea celui mai scurt drum între două noduri, se recomandă folosirea algoritmului BFS, implementat anterior, apoi reconstrucția drumului mergând din părinte în părinte în arborele BFS.

Vectorul `path`, în care se completează traseul, va avea o lungime de minim 1000 de elemente, la apelarea funcției. Returnați numărul de elemente care au fost completate în el, sau -1 în cazul în care nu se poate ajunge la nodul `end` pornind de la nodul `start`.

1.5.6 Bonus: Unde poate ajunge un cal pe tablă? (0.5p)

Folosind framework-ul dat, arătați că un cal poate ajunge pe orice poziție a unei table de șah goale, pornind din colțul din stânga-sus. Dați exemple de tablă care să conțină poziții libere la care nu se poate ajunge.

References

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