Implementarea algoritmului de baleiere pentru aflarea intersecțiilor unor segmente

Ciuperceanu Vlad-Mihai
Grupa 151

1 Aspecte ale proiectării codului

Pentru memorarea datelor citite, am construit clasa *Segment*, ce reține, pentru fiecare segment, coordonatele capetelor sale, precum și indicele său. Compararea a două segmente se va face, în ordine, după coordonatele x, respectiv y a capătului din stânga, apoi coordonatele x, respectiv y a capătului din dreapta. Alături de clasă, am implementat și câteva funcții ce permit verificarea intersecției a două segmente, precum și găsirea punctului de intersecție.

Înainte de a începe algoritmul de *sweeping*, vom asocia fiecărui capăt de segment câte un eveniment, care se va declanșa atunci când linia imaginară cu care parcurgem planul ajunge la punctul respectiv. Pentru un eveniment reținem coordonatele punctului asociat, indicele său și tipul de capăt (0 - stânga, 1 - dreapta). Două evenimente vor fi comparate după coordonata x, apoi după tipul de punct și, în final, după coordonata y (pentru a procesa evenimentele în ordinea corectă). Vom pune toate evenimentele într-un vector încă de la citire, urmând să le sortăm, folosind metoda de comparare de mai sus.

Algoritmul folosește o structură de date de tip arbore binar de căutare echilibrat, în care reținem evenimentele active de la un anumit moment. Mai exact, un considerăm că un segment este activ dacă am procesat capătul din stânga, dar încă nu am ajuns la capătul din dreapta. De aceea, vom folosi structura de date în care ținem evenimentele active pentru a insera un eveniment atunci când dăm de un capăt stânga și pentru a scoate un eveniment asociat unui capăt stânga atunci când dăm de capătul său drept. Astfel, parcurgem evenimentele stocate, aflându-ne în unul din două cazuri. Dacă punctul asociat este un capăt din stânga al unui segment, atunci căutăm punctele sale de intersecție cu segmentele fix de deasupra și de dedesubt (vecinii) care au evenimentul asociat unui capăt activ, urmând să inserăm evenimentul în structura de date. Altfel, verificăm din nou cu vecinii, urmând să scoatem din structură evenimentul corespunzător capătului stâng. Fiecare punct de intersecție va fi stocat într-un map în care, pentru doi indici ale unor segmente, reținem coordonatele punctului lor de intersecție.

Per total, operațiile de care avem nevoie sunt: inserare, ștergere, căutare,

găsirea celui mai mic element care este mai mare sau egal decât un element dat, precum și diverse funcții care să verifice unele cazuri speciale care trebuie tratate.

Prima implementare folosește structura de date *map* din STL, în care nu ne interesează neapărat valorile, ci operațiile pe care le putem face rapid cu cheile. Vom reține un $map < Event, int > activeEventsMAP$, în care păstrăm, pentru fiecare eveniment, indicele segmentului său. Operațiile de mai sus există deja, prin *insert*, *erase*, $find$, $lower\_bound$, *begin*, *end*, ce au complexitățile $O(log\, n)$, ultimele două fiind efectuate în $O(1)$.

A doua implementare folosește structura de date AVL, implementată prin clasa template cu același nume. În cadrul său am definit clasa $Node$ pe care o folosim ulterior, clasă ce este folosită doar în cadrul clasei AVL. Pentru fiecare nod reținem cheia (al cărei tip de date este dat de template), pointeri către nodurile din stânga și din dreapta, precum și înălțimea. Pe lângă funcționalitățile necesare structurii de date, am definit și alte metode pentru a simula funcțiile predefinite din STL care ne trebuie, având aceeași complexitate pentru ele. De asemenea, am păstrat publice doar funcțiile folosite în mod direct în main, restul rămânând private.

2 Exemple

Din punctul de vedere al complexității, cele două implementări ar trebui să fie similare, întrucât și structura de date *map* din STL are în spate tot un arbore binar echilibrat, fie că este vorba de AVL Trees sau de Red-Black Trees.

Pentru a testa complexitatățile celor două structuri de date, am păstrat ambele implementări în main, contorizând timpul trecut.

Mai întâi, am rulat pe 2 teste mici, apoi pe unele puțin mai mari (primele 3 regăsindu-se sub formă de comentariu la sfârșitul codului din main, iar ultimul având aceleași coordonate ale segmentelor ca la cel cu 20 de segmente, însă având 70 de segmente; doar primul test a conținut puncte de intersecție, celelalte neavând niciunul).

Rezultatele, măsurate în microsecunde, au fost:

map	AVL
20	10
10	15
13	15
39	19

Astfel, putem spune că *map* și AVL sunt relative ca viteză, implementarea cu AVL fiind uneori mai rapidă datorită avantajului pe care îl are față de Red-Black Trees atunci când avem nevoie de operații de căutare.