Pretraga u dubinu

Kako biste tražili put kroz lavirint?

Ideja koja je ovde opisana verovatno vam je već poznata: ostavljate iza sebe trag (mrvice, kamenčiće, nit konca…) i napredujete dok god možete. Kada dođete do mesta sa kojeg se ne može dalje napredovati, vraćate se unazad tragom do najbližeg mesta sa kojeg možete krenuti nekom neistraženom putanjom i nastavljate tom putanjom. Ovako postupate dok ne nađete izlaz, ili dok ne istražite sve putanje (što bi značilo da izlaz ne postoji).

Ovo je u osnovi algoritam pretrage najpre u dubinu (engl. depth-first search, skraćeno DFS). Pretraga se zove najpre u dubinu jer prvo napredujete, to jest idete dublje u lavirint dokle god možete, a vraćate se samo ako ne možete da idete dalje u dubinu. Osim pretrage u dubinu, mogli bismo na primer, prvo da označimo sva mesta na koja možemo da stignemo u jednom koraku, zatim ona na koja možemo da stignemo u dva koraka I tako dalje. Ovakva pretraga se zove najpre u širinu (engl. breadth first search, skraćeno BFS), o njoj će biti reči u drugom delu ove lekcije.

Razmortimo još jedno pitanje u vezi sa opisanim algoritmom BFS: da li pri vraćanju tragom treba taj trag uklanjati ili ostavljati? Ako bismo uklanjali trag, moglo bi se dogoditi da ista polja posećujemo veliki broj puta iz raznih pravaca. Ako bismo pak ostavljali trag i posle vraćanja, bilo bi komplikovanije prikazati putanju od polazne do tekuće pozicije i razlikovati je od delova lavirinta kojima smo išli i vratili se (slepi krakovi). Kako obe odluke imaju dobrih i loših strana, u opštem slučaju je najbolje da koristimo dve vrste traga, jedan kada napredujemo, a drugi kada se vraćamo. Tako uvek imamo jasno označenu putanju od polazne do tekuće pozicije (koja u slučaju nailaska na izlaz postaje rešenje), a takođe i garanciju da ćemo svako polje lavirinta posetiti najviše jedanput.

Neka je lavirint predstavljen matricom *a*. Funkcija *Seek* kojom tražimo put kroz lavirint ima koordinate tekućeg polja kao argumente. Pretpostavljamo da su sam lavirint i njegove dimenzije *rowCnt* i *colCnt* dostupni funkciji. Funkcija vraća vrednost koja govori da li je put pronađen. Algoritam traženja puta mogao bi se ovako zapisati:

```
Seek(i, j)
    if (i < 1) or (j < 1) or (i > rowCnt) or (j > colCnt)
        return false;

    if (a[i][j] = kExit)
        PrintMatrix();
        return true;

    if (a[i][j] != kPass)
        return false;

    a[i][j] = kSearching;
    bool found = Seek(i-1, j) or Seek(i, j-1) or Seek(i+1, j) or Seek(i, j+1);
    a[i][j] = kSearched;
    return found;
```

Trajanje ovakve pretrage je u najgorem slučaju srazmerno veličini prostora pretrage (u primeru to je veličina lavitinta). Sama pretraga zahteva prostor srazmeran najvećoj dostignutoj dubini pretrage, što u najgorem slučaju može biti isto štoi veličina prostora pretrage. U rekurzivnoj implementaciji, koja je jednostavnija, ta memorija se koristi implicitno za stek rekurzivnih poziva. U nerekurzivnoj implementaciji bilo bi neophodno da se pamti niz polja koja su posećena, a kroz koja pretraga još uvek traje (polja označena vrednošću *kSearching* u rekurzivnom algoritmu gore).

Opisano traženje puta u lavirintu je samo jedan, možda najpoznatiji primer primene algoritma pretrage u dubinu. Međutim, pretraga u dubinu se može koristiti kad god je potrebno nizom poteza stići od date polazne situacije (pozicije, stanja) do završne ili jedne od završnih. Primeri ovakvih zadataka su Lojdova slagalica od 15 brojeva i problem osam dama.

U slučaju Lojdove slagalice početna pozicija se bira na slučajan način, završna pozicija je ona na kojoj su brojevi poređani po veličini (kao na slici), a potez se izvršava tako što prazno polje zameni mesto sa jednim njemu susednim poljem. Na žalost, broj pozicija u ovom problemu (igri) je toliko veliki da kompletna pretraga prostora svih pozicija može trajati vrlo dugo. Da ne bismo morali da prolazimo ceo prostor pretrage, poželjno je na neki inteligentan način birati sledeći potez. Postoji niz poboljšanja pretrage u dubinu i upravo ova slagalica je odličan primer kako se pretraga može ubrzati usmeravanjem i ograničavanjem, ali o tome će biti reči u jednoj od lekcija naprednijeg nivoa.

Lojdova slagalica (15 puzzle)

Problem osam dama se sastoji u sledećem: potrebno je na šahovsku tablu postaviti osam dama tako da se one međusobno ne napadaju, to jest da se nikoje dve dame ne nalaze u istom redu, istoj koloni ili na istoj dijagonali table. Početna pozicija je ovde prazna tabla, potez je dodavanje jedne dame na tablu poštujući ograničenja o nenapadanju, a završna pozicija je bilo koja pozicija sa osam dama na tabli, tako da se dame ne napadaju. Da bismo pojednostavili pretragu, primetimo da tih osam dama moraju da se nalaze u različitim kolonama table. Zbog toga možemo i-tu damu uvek dodavati u i-tu kolonu. Dodavanje vršimo tako što probamo svako polje kolone redom i proveravamo da li dama sa tog polja napada neku od ranije postavljenih dama. Ako napada, vraćamo se i pokušavamo sledeće polje, a ako ne napada, nastavljamo pretragu prelaskom na sledeću damu i sledeću kolonu. Kada uspemo da načinmo osam koraka u dubinu, tada je na tabli osam dama i došli smo do jednog rešenja koje prikazujemo. Pretraga se zaustavlja posle prvog pronađenog rešenja, ali se algoritam veoma lako može izmeniti da posle jednog pronađenog rešenja nastavlja pretragu za preostalim rešenjima.

U algoritmu koji sledi, niz ***pos*** sadrži pozicije dama po kolonoma, petlja po i isprobava sve pozicije dama na tekućoj dubini (u tekućoj koloni), a petlja po j proverava da li se novopostavljena dama napada sa nekom od prethodno postavljenih dama.

```
Search(depth)
    if (depth == 8)
        OutputSolution;
        return true;

    ++depth;
```

```
for i = 1 to 8
    pos[depth] = i;
    valid = true;
    for j = 1 to depth-1
        if (pos[j] == pos[depth] or
            j + pos[j] == depth + pos[depth] or
            j - pos[j] == depth - pos[depth])
                valid = false;
    if (valid && Search(depth))
        return true;

return false;
```

Zanimljivo je da je pre pojave računara bio potreban genije i upornost veličine Gausa (jedan od najvećih matematičara svih vremena) za potpuno rešavanje ovog problema (nalaženje svih 92 rešenja). Danas je to, kao što vidimo, običan školski zadatak do čijeg rešenja može da dođe svako ko ume da programira.

Pretraga u širinu

Pretraga u širinu se od pretrage u dubinu razlikuje po redosledu obliaženja (posećivanja) polja, odnosno pozicija u postoru pretrage. Kao što je već napomenuto, ovom pretragom se prvo pronalaze sve pozicije koje nastaju posle jednog poteza, zatim sve koje nastaju posle dva poteza, itd. Kada je poznat skup S(k) pozicija dostižnih k poteza, tada se skup S(k+1) pozicija dostižnih u k+1 poteza dobija napredovanjem za jedan korak na svaki mogući način is svake od pozicija iz S(k). Ranije posećene pozicije se ne uzimaju ponovo u obzir. Postupak se nastavlja dok se ne dođe do tražene (završne) pozicije, ili dok se ne pretraže sve pozicije u zadatim okvirima.

Prednost pretrage u širinu je što se prvo posećivanje svake nove (pa i završne) pozicije uvek postiže u najmanjem mogućem broju poteza. Zahvaljujući tome, rešenja otkrivena pretragom u širinu su uvek optimalna po broju poteza. Sa druge strane, radi pretrage u širinu potrebno je organizovati pamćenje posećenih pozicija, da bi se pretraga mogla nastaviti iz svih tih pozicija.

Algoritam pretrage u širinu koristi jednu naprednu strukturu podataka, koja se naziva red (ili red za čekanje, engl. queue). Ona zaista funkcioniše kao red ljudi koji čekaju na neku uslugu: objekti mogu da se dodaju na kraj reda, a objekat koji je na početku reda može da se uzme iz reda i da se upotrebi (odnosno da dobije uslugu na koju je čekao). U slučaju lavirinta, objekti predstavljaju pozicije iz prostora pretrage, a to su ovde dostignuta polja. Red se lako može implementirati pomoću običnog niza i dve celobrojne promenljive, koje označavaju indeks početka i kraja reda. U nekim programskim jezicima u okviru standardnih biblioteka postoji red kao već implementirana struktura podataka, koja se odmah može koristiti. O redovima će biti više reči u posebnoj lekciji, posvećenoj često korišćenim strukturama podataka.

Da bismo bolje ilustrovali mogućnosti algoritma pretrage u širinu, postavićemo problem lavirinta nešto drugačije. Neka je matricom *a* dat lavirint i polazno polje u njemu. Zadatak je da se za svako polje lavirinta odredi najmanji broj poteza koji je potreban da se do tog polja stigne kroz lavirint kretanjem od polaznog polja.

Stavljaćemo u red za čekanje koordinate onih polja koja su susedna posećenim, tako da ta susedna polja čekaju da budu posećena. Za svako polje ćemo u pomoćnoj matrici *m* (često se radi uštede prostora upotrebljava i ulazna matrica *a*) pamtiti broj poteza koji je potreban za stizanje do tog polja.

```
MinMoves(xstart, ystart)
    dirx[4] = {1, -1, 0, 0}; // Ovaj niz, zajedno sa sledecim zadaje 4 moguca smera kretanja
    diry[4] = {0, 0, 1, -1};

    for r = 1 to rowCnt
        for c = 1 to colCnt
            m[r][c] = infinity;

    queue<Square> q;
    q.push(Square(xstart, ystart));
    m[ystart][xstart]=0;
    while (!q.empty())
        x = q.front().x;
        y = q.front().y;
        q.pop();
        for dir = 1 to 4
            xx = x + dirx[dir];
            yy = y + diry[dir];
            if (xx > 0 && yy > 0 && xx <= rowCnt && yy <= colCnt &&
               m[yy][xx] == infinity && a[yy][xx] != kWall)
                m[yy][xx] = m[y][x] + 1;
                q.push(Square(xx, yy));
    // Ispisivanje izracunate matrice
    for r = 1 to rowCnt
        for c = 1 to colCnt
            if (m[r][c] == infinity)
                Write('.');
            else
                Write(m[r][c]);
        WriteLine();
```

Prilikom prtetrage u širinu želimo da dodamo u red samo one susede tekućeg polja, koji još nisu posećeni. Ovde se posećena polja prepoznaju po tome što imaju upisan potreban broj poteza (različit od inicijalne vrednosti).

Vreme rada i zauzeće memorije su u najgorem slučaju srazmerni veličini prostora pretrage.