Algorytmiczne Aspekty Teorii Gier

Ćwiczenia 3

2 marca 2009

1. Wartość drzewa gry, algorytm zrandomizowany. Znaleźć algorytm zrandomizowany, który dla pełnego drzewa binarnego o wysokości $h = 2k$ (wysokość definiujemy jako ilość krawędzi na najdłuższej ścieżce od korzenia) z wartościami zero i jeden w liściach oblicza wartość drzewa gry (definicja wartości drzewa gry pod koniec poprzednich ćwiczeń) odwiedzając średnio nie więcej niż 3^k liści.

Wskazówki

- Najpierw zaproponujmy jakiś sensowny algorytm, a później będziemy się martwić obliczeniem czasu, w którym działa. Najprawdopodobniej to będzie ten algorytm o który chodzi.
- Zastanówmy się, czy jeśli wiemy coś o pewnych poddrzewach, to być może nie musimy obliczać wartości innych.
- Tak. Przykładowo jeśli w węźle max jedno z poddrzew wiemy, że ma wartość 1, to nie musimy obliczać wartości drugiego, bo wiadomo, że w ojcu też będzie wartość 1. Analogicznie dla min i wartości 0.
- Warto przypomnieć, omówić główną ideę alfa-beta obcięć w algorytmie minimax. To ma podobną ideę jak u nas, tylko, że w naszym przypadku, gdy są wartości tylko 0 i 1, to możemy wykonywać nawet prostsze obcięcia.
- Algorytm jest zatem następujący:
  Będąc w danym węźle losujemy które z jego poddrzew obliczamy najpierw. Jak obliczymy wartość tego poddrzewa i wówczas wartość ojca będzie już zdeterminowana, to nie obliczamy wartości drugiego poddrzewa. W ten sposób postępujemy w każdym węźle.
- Zastanówmy się teraz średnią ilością liści, które musimy odwiedzić. Jak to obliczyć?
- Niech *drzewo dobre* to takie drzewo, które jeśli korzeń to max, to ma wartość 1, a jeśli korzeń do min, to ma wartość 0. Niech *drzewo złe* to takie drzewo, które nie jest dobre. Niech D_k i Z_k to średnia ilość odwiedzeń liści odpowiednio w drzewie (pełnym, binarnym) dobrym i złym o głębokości k. Chcemy zatem pokazać, że $D_{2k} \leqslant 3^k$ i $Z_{2k} \leqslant 3^k$.
- Zastosujmy rekurencję. Mamy oczywiście $D_0 = 1$, $Z_0 = 1$. Poza tym $Z_{k+1} = D_k + D_k = 2D_k$, gdyż jeśli drzewo głębokości $k+1$ jest złe, to jego poddrzewa muszą być dobre, czyli nie ominiemy żadnego obliczenia. Mamy też $D_{k+1} \leqslant \frac{1}{2}(Z_k + (D_k + Z_k)) = Z_k + \frac{1}{2}D_k$, gdyż jeśli drzewo głębokości $k+1$ jest dobre, to któreś jego poddrzewo jest złe i mamy co najmniej $\frac{1}{2}$ prawdopodobieństwa, że najpierw policzymy jego wartość i nie będzie trzeba liczyć wartości tego drugiego.
- Łatwo teraz indukcyjnie wykazać, że $D_{2k} \leqslant 3^k$ i $Z_{2k} \leqslant 3^k$ (przez indukcję dowodzimy oba fakty razem).
- Gdy przyjrzymy się dokładniej sytuacji, to widać, że jest tu jeszcze trochę luzu i asymptotycznie ilość odwiedzonych liści powinna być mniejsza niż 3^k. Można to rozwiązać rekurencyjnie. Korzystając z napisanych wyżej rekurencji można otrzymać, że $D_{k+2} \leqslant$

$Z_{k+1} + \frac{1}{2}D_{k+1} = \frac{1}{2}D_{k+1} + 2D_k$. Rozwiązując rekurencję otrzymujemy asymptotyczne oszacowanie górne na D_{2k} wynoszące $\left(\frac{\sqrt{33}+1}{4}\right)^2 k = \left(\frac{34+2\sqrt{33}}{16}\right)^k \approx 2,84^k$.

2. Pebbling game. Gra jest jednoosobowa. Gramy na DAG-u (Directed Acyclic Graph), czyli grafie skierowanym acyklicznym o stopniu wejściowym 2 (czyli do każdego wierzchołka wchodzą co najwyżej 2 strzałki). Celem gry jest położenie kamyka na pewnym wyróżnionym wierzchołku v. Dostępne ruchy to:

- zdjęcie kamyka, który leży na pewnym wierzchołku
- położenie kamyka na wierzchołku w, jest to jednak dozwolone o ile na wszystkich wierzchołkach, z których wchodzą strzałki do w leżą już kamyki (czyli w szczególności, jeśli do w nie wchodzi żadna strzałka, to od razu można kłaść na w kamyk).

Naszym celem jest zminimalizować ilość użytych w grze kamyków. Ile kamyków potrzeba do położenia kamyka w korzeniu drzewa binarnego o wysokości h (niekoniecznie pełnego)? A ile potrzeba do położenia kamyka w korzeniu drzewa binarnego o n wierzchołkach?

Uwaga: Dalszą częścią tego zadania jest zadanie domowe nr 1.

Wskazówki

- Sprawdzić ile wychodzi dla małych przypadków, na przykład małego pełnego drzewa binarnego.
- Udowodnić, że $h + 2$ działa.
- Indukcyjnie po wysokości (łatwo, dla jednego syna mam co najwyżej $h + 1$, zostawiam kamyk w synie i mam jeszcze $h + 1$ na drugiego syna (o wysokości być może mniejszej nawet niż $h - 1$). Zatem faktycznie $h + 2$ wystarcza.
- Dla drzewa o n wierzchołkach podobnie.
- Można pokazać indukcyjnie, że $\lceil \lg n \rceil$ wystarczy, najpierw do drzewa o syna o większej (ściśle rzecz biorąc nie mniejszej) ilości wierzchołków dajemy kamyk, a potem mając ten jeden kamyk w większym synu robimy mniejszego syna (który ma przynajmniej 2 razy mniej wierzchołków niż korzeń). W ten sposób łatwo wychodzi.

3. Piony w rzędzie. Gra dwóch graczy, ruszają się naprzemiennie. Na początku w rzędzie jest ustawiona pewna liczba pionów, pomiędzy niektórymi są przerwy. W pojedynczym ruchu można zdjąć z planszy bądź 1 pion, bądź 2 sąsiadujące piony. Przegrywa ten, kto nie może zrobić ruchu. Napisać program, który odpowie na pytanie kto z danej pozycji początkowej (czyli na przykład ciągu spójnych grup pionów wielkości odpowiednio $2, 5, 3, 1, 3$) posiada strategię wygrywającą (gra ta jest w oczywisty sposób zdeterminowana, gdyż jest skończona, czyli dla każdej pozycji któryś z graczy posiada strategię wygrywającą z tej pozycji).

Wskazówki

- Skorzystać z czegoś, co już było.

- Spojrzeć jak wyglądają możliwe ruchy. Tak naprawdę albo pojedynczy słupek zmniejszamy, albo rozdzielamy na dwa mniejsze słupki, na których gry już toczą się rozdzielnie.
- Zauważmy, że możemy tu zastosować Grundy numbers, o których była mowa na pierwszych ćwiczeniach. Zauważmy, że oprócz obserwacji powyżej - nasza gra ma taką własność, że gracze są tu symetryczni, czyli każdy z nich ruszając się z konkretnej planszy może zrobić te same ruchy. Ta własność również jest konieczna do skorzystania z Grundy numbers.
- Dlaczego tak naprawdę Grundy numbers. Zobaczmy, że jeśli ze słupkiem długości n związżemy liczbę $G(n)$, to jest ona równa najmniejszej liczbie całkowitej większej lub równej zero, która nie występuje wśród $G(i) \otimes G(n-1-i)$ oraz wśród $G(i) \otimes G(n-2-i)$ dla i od 0 do $n-1$ lub $n-2$ odpowiednio.
- Schemat działania jest następujący. Zwracamy xor wszystkich Grundy numberów dla odpowiednich spójnych grup pionów. Wcześniej liczymy Grundy numbery dla wszystkich liczb od 0 do $M-1$ (załóżmy że jest to maksymalna wielkość słupka), metodą liczenia najmniejszej liczby nie należącej do zbioru określonego wyżej.
- Przejdźmy do napisanie kodu. Przykładowy może wyglądać następująco:

```
// Uwaga, nie kompilowałem tego programu, więc może zawierać błędy
// Tu oczywiście jakieś includy
#define M 1000
#define pb push_back
#define REP(i,n) for(i=0;i<n;i++) // moje ulubione makro, bardzo skraca napisy

int gnumbers[M];

int mex(vector<int> v) {
// minimum excludant
// napisałem go chamsko kwadratowo, choć pewnie da się lepiej
  int i, j;
  REP(i,v.size()+1) {
    REP(j,v.size()) {
      if (v[j]==i) break;
      return i;
    }
  }
}

int gnumber(int n) {
  int i;
  vector<int> v;
  REP(i,n/2) {
  // To n/2 jest tu z górką i tak zwykle jedna, czy dwie pary
  // pośrodku zostaną wrzucone dwa razy
    v.pb(gnumbers[i]^gnumbers[n-1-i]);
    v.pb(gnumbers[i]^gnumbers[n-2-i]);
```

```
  }
  return mex(v);
}

int mainResult(vector<int> v) {
  int i, ret;
  REP(i,M) gnumbers[i] = 0;
  REP(i,M) gnumbers[i] = gnumber(i);
  ret = 0;
  REP(i,v.size()) ret ^= gnumbers[v[i]];
  return ret;
}
```