“Programowanie gier logicznych”
Janusz Kraszek

materiały zamieszczone w czasopiśmie “Komputer”
(nr 1/1986, 2/1986, 4/1986, 5/1986) zebrał i przetworzył
w czerwcu 2020: Kaz / AtariOnline.pl

PROGRAMOWANIE

Strategia wygrywająca

Trudno powiedzieć, czy istnieje wyraźnie określona grupa "gier logicznych" i jakie właściwie gry można do niej zaliczyć. Tytuł ten wskazywać ma głównie na fakt, że nie będziemy zajmowali się tutaj grami zręcznościowymi, typu adventure czy losowymi. Co pozostaje? Duża grupa gier, które wymagają pewnej strategii, "logicznego myślenia" – jak się to czasami określa, nierzadko wiedzy itp. Na przykład kółko i krzyżyk, master mind, reversi, warcaby, szachy czy go. Że komputery potrafią grać w takie gry, o tym wiemy. Jeszcze przed masową produkcją mikrokomputera można było nabyć "elektronicznego partnera" do gry w szachy, a obecnie nie ma chyba właściciela komputera domowego, który nie posiadałby programu grającego w szachy czy warcaby.

Czy grając w jedną z tych gier z komputerem nie zdarzyło się nam zadziwić nad jego „inteligentnym zachowaniem"? Może poczuliśmy niepokój?

W cyklu tych kilku artykułów chciałbym powiedzieć na czym polega programowanie gier logicznych, przedstawić kilka przykładów, a także zachęcić do samodzielnych prób.

Zacznijmy od zastanowienia się na czym polega problem napisania programu grającego w taką grę. Wyobraźmy sobie, że wykonaliśmy ruch, kolej więc na akcję programu. W jaki sposób komputer może "wymyśleć" w miarę dobry ruch?

Nie ma jednej odpowiedzi na to pytanie, a to dlatego, że gra grze nierówna. Podstawowymi parametrami gier są: liczba wszystkich możliwych partii, inaczej określana liczbą kombinacji oraz to, czy znana jest czy nie strategia wygrywająca (lub nieprzegrywająca).

Co do drugiego problemu, to wiadomo, że strategia (jedna lub druga) istnieje (dla gier tu omawianych dział nauki z pogranicza matematyki i informatyki zwany teorią gier dostarcza stosownego dowodu). Natomiast parametr określający liczbę kombinacji wyraża złożoność gry. Gdy nie jest zbyt duży, tzn. taki, żeby zbadanie wszystkich możliwości stało się wykonalne w rozsądnym czasie, wówczas nie znając innej strategii można zbudować program oparty na tzw. przeszukiwaniu, czyli wybieraniu najlepszej z wszystkich możliwości. Przykład i dokładniejsze omówienie tej metody przedstawię innym razem.

Jednakże większość gier tak łatwo nie poddaje się zaprogramowaniu. Liczbę wszystkich możliwych partii w warcaby, szachy i go określają kolejne liczby: 10^{40} (10 do potęgi 40), 10^{120}, 10^{750}. Zakładając, że komputer jest w stanie zbadać milion ruchów w ciągu sekundy (co jest trochę za dużo nawet dla największych maszyn), potrzebowalibyśmy i tak około 3×10^{26} lat na jedną partię warcabów, co daje pojęcie o wielkości liczb, z jakimi mamy do czynienia. W porównaniu z tym, jakże przyjemna i prosta wydaje się perspektywa zaprogramowania gry, dla której znamy strategię wygrywającą. Wystarczy "przełożyć" ową strategię na język zrozumiały dla komputera i już. Jeżeli nawet nie zawsze jest to takie proste, to jednak prowadzi do dobrych wyników. Czasami nawet do zbyt dobrych – niektórym grom grozi zapomnienie tylko dlatego, że komputer radzi sobie z nimi zbyt łatwo.

Przykładem gry o znanej strategii jest NIM, częściej uprawiana, niż znana z nazwy. Najpowszechniej stosowanym rekwizytem w NIM są zapałki. Oto zasady. Gra dwóch graczy. Przed grą układają zapałki w trzy rzędy o dowolnej ilości w każdym. Ruch polega na zabraniu z jednego dowolnie wybranego rzędu dowolnej ilości zapałek – od jednej do wszystkich w rzędzie. Ruchy wykonują gracze na przemian. Przegrywa ten, który nie może wykonać ruchu, gdyż nie ma już zapałek w żadnym rzędzie.

Pod względem ilości kombinacji NIM jest grą tego samego typu co szachy i warcaby, szczególnie że ilość rzędów może być dowolnie duża. Jednakże istnienie strategii wygrywającej czyni z niej grę łatwą do zaprogramowania. Program taki gra w NIM nie popełniając błędów (nawet dla dużej liczby rzędów) i wygrywa zawsze, kiedy tylko jest to możliwe. Czy to znaczy, że strategia wygrywająca może czasami nie prowadzić do zwycięstwa? Nie, tak nie jest. Strategie wygrywające istnieją dla gier z tzw. pełną informacją, a w NIM ilość zapałek w rzędzie jest losowa. Dopiero po rozdzieleniu mamy pełną informację i możemy mówić o istnieniu strategii. Może się wówczas okazać, że polega ona między innymi na oddaniu pierwszego ruchu przeciwnikowi, gdyż kto zacznie, ten przegra. A porządny program zostawia człowiekowi decyzję, kto ma rozpocząć grę.

Ale dość, powiedzmy jaka jest ta strategia. Niech przykładowo będzie to 15 zapałek w pierwszym rzędzie, 8 w drugim i 11 w trzecim. Należy rozpocząć od zamiany liczb określających ilość zapałek w rzędzie z systemu dziesiętnego na dwójkowy. (Systemy liczbowe – patrz podręcznik do IV klasy szkoły podstawowej.) Oto mamy:

$15_{(10)} = 1111_{(2)}$
$8_{(10)} = 1000_{(2)}$
$11_{(10)} = 1011_{(2)}$

```
  10 REM NIM
  20 REM 27 to max ilość pionów
w jednym rzędzie, 6 minimalna
  30 LET ilrz =3: LET maxil=27:
LET minil=6: DIM b(3,5): DIM d(2
,5): DIM l(3,2)
  40 LET m=maxil-minil
  50 FOR k=1 TO ilrz
  60 LET il=minil+INT (m*RND)
  65 LET f=5*k
  70 FOR z=1 TO il
  80 PRINT AT f,z;"■": PRINT AT
f+1,z;"■"
  90 NEXT z
 100 LET l(k,1)=il
 110 NEXT k
 120 GO SUB 500
 130 INPUT "Czy mam zacząć? (t/n
) ";a$
 140 IF a$="n" OR a$="N" THEN GO
 TO 300
 200 GO TO 1000
 300 IF l(1,1)=0 AND l(2,1)=0 AN
D l(3,1)=0 THEN PRINT AT 10,5;"W
ygrałem !!!": FOR k=1 TO 7: BEEP
 .4,k+2: NEXT k: STOP
 310 INPUT "Podaj ruch:
           numer rzędu (1, 2 lu
b 3) ";i
 315 IF i<>1 AND i<>2 AND i<>3 T
HEN GO TO 310
 320 INPUT " ilość zabieranych
pałeczek ";il
 325 IF il<=0 OR il>l(i,1) THEN
GO TO 310
 330 GO SUB 3000
 350 GO TO 1000
```

Liczbę binarną otrzymujemy zapisując od prawej strony reszty z dzielenia liczby dziesiętnej przez 2 (patrz też list programu).
Sprawdźmy ostatni z otrzymanych wyników:

$1011_{(2)} = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 1 + 2 + 8 = 11.$

Dla tych, którzy niewiele wiedzą o systemach liczbowych i nie mają podręcznika do IV klasy – krótkie wyjaśnienie. Dlaczego w liczbie 125 wiadomo, że 1 oznacza 100 czyli 10^2? A gdyby stała tam cyfra 3, to czy wiedzielibyśmy, że oznacza 300 czyli 3×10^2? Zachodzi wzór ogólniejszy:
cyfra razy podstawa systemu do potęgi $(n-1)$, gdzie n jest pozycją cyfry w liczbie liczoną od prawej strony. Dlatego też 125 to 100 + 20 + 5, ale 125 w systemie ósemkowym to tylko 85 = 64 + 16 + 5.

Ustawiając binarne liczby jedna pod drugą otrzymujemy cztery kolumny zer i jedynek:

```
1 1 1 1
1 0 0 0
1 0 1 1
```

Strategia gry jest następująca:
Wykonuj takie ruchy, by liczba jedynek w każdej kolumnie była parzysta lub równa zeru. Jeżeli nie jest to możliwe przed wykonaniem pierwszego ruchu, zaproponuj rozpoczęcie gry przeciwnikowi.

W naszym przykładzie widzimy, że dwie pierwsze kolumny od lewej zawierają nieparzystą ilość jedynek. Należy ustalić, w którym rzędzie wykonać ruch oraz ile zabrać zapałek. Dlatego trzeba wybrać ten rząd, w którym występuje jedynka w najbardziej na lewo stojącej kolumnie o nieparzystej ilości jedynek – określmy ją jako kolumnę nieparzystą. W naszym przykładzie jest to kolumna pierwsza z lewej, a jedynka występuje w każdym z trzech rzędów. Można wybrać dowolny z nich. My weźmy pierwszy. Natomiast liczba zapałek, jaka ma pozostać w wybranym rzędzie jest liczbą, jaka powstaje po zamianie bitu (cyfra 0 lub 1) na przeciwny w kolumnach nieparzystych i pozostawieniu bez zmian w kolumnach parzystych. Stąd przez odjęcie obliczamy liczbę zapałek, które należy zabrać z rzędu.

W naszym przykładzie zamieniamy jedynki na zera w dwóch pierwszych kolumnach z lewej strony w rzędzie pierwszym. Daje to liczbę 0011 czyli 11, a w systemie dziesiętnym 3. Stąd poszukiwaną liczbą jest 15−3 =12. Zauważmy, że odejmując 12 zapałek z pierwszego rzędu otrzymujemy parzyste liczby jedynek we wszystkich kolumnach. A o to przecież chodziło.

Pozostałe możliwe ruchy dla tego przykładu to zabranie z drugiego albo trzeciego rzędu czterech zapałek – zamiana 1 na 0 w kolumnach pierwszej i drugiej od lewej. Oba powyższe ruchy również dadzą same parzyste kolumny. Po każdym z nich przeciwnik musi przegrać, jeżeli tylko w dalszym ciągu kontynuować będziemy powyższą procedurę wyboru ruchu. Jak łatwo zgadnąć, nietrudno jest to zaprogramować. Tak przedstawia się program grający w NIM.

Warto może dodać jeszcze do niego kilka szczegółów, takich jak efekty dźwiękowe czy lepsza grafika. To jednak zostawiam dla wszystkich tych, którzy będą mieli na to ochotę. A tym, którzy zechcą zaprogramować własną grę radzę, by nie żałowali wysiłku przed rozpoczęciem programowania i zastanowili się nad strategią i algorytmem podczas szukania rozwiązań najlepszych i najprostszych. Może ktoś znajdzie strategię wygrywającą?

Janusz Kraszek

GIER LOGICZNYCH

```
500 REM zerowanie
505 FOR k=1 TO 5: LET d(1,k)=0:
NEXT k
510 FOR i=1 TO 3
520 GO SUB 2000
525 PRINT AT 5*i,30;l(i,1)
530 NEXT i
540 RETURN
1000 REM ruch
1010 LET rz=1: LET x=l(1,2): IF
x<l(2,2) THEN LET x=l(2,2): LET
rz=2
1020 IF x<l(3,2) THEN LET rz=3:
LET x=l(3,2)
1030 IF x=0 THEN PRINT AT 10,5;"
Przegrałem !?": BEEP .5,-7: BEEP
.5,-9: STOP
1040 FOR k=x TO 1 STEP -1
1050 IF d(1,k)=1 THEN GO TO 1100
1060 NEXT k
1070 REM układ z parzystymi
kolumnami
1080 LET il=1: LET i=rz: GO SUB
3000: GO TO 300
1100 FOR z=1 TO ilrz
1110 IF b(z,k)=1 THEN GO TO 1200
1120 NEXT z
1130 PRINT "błąd - tu nigdy nie
powinniśmy się znaleźć": STOP
1200 LET l=0: LET p=1
1210 FOR w=1 TO l(z,2)
1220 LET c=b(z,w)
1230 IF d(1,w)=1 THEN LET c=1 AN
D b(z,w)=0
1240 LET l=l+p*c: LET p=2*p
1250 NEXT w
1260 LET il=l(z,1)-l: LET i=z: G
O SUB 3000: GO TO 300
```

```
2000 REM zapis dwójkowy
2005 FOR k=1 TO 5: LET d(2,k)=0:
NEXT k
2010 LET x=l(i,1): LET j=0
2020 IF x=0 THEN GO TO 2060
2025 LET j=j+1: LET y=INT (x/2)
2030 LET c=x-2*y: LET d(2,j)=c
2050 LET x=y: GO TO 2020
2060 LET l(i,2)=j
2065 FOR k=1 TO 5
2070 IF d(2,k)<>b(i,k) THEN LET
d(1,k)=1 AND d(1,k)=0
2080 LET b(i,k)=d(2,k)
2090 NEXT k
2100 RETURN
3000 REM wykonaj ruch: rząd=i,
ilość to il
3300 LET l(i,1)=l(i,1)-il
3310 LET f=5*i
3400 FOR k=l(i,1)+1 TO 31
3500 PRINT AT f,k;" ": PRINT AT
f+1,k;" "
3600 NEXT k
3610 PRINT AT f,30;l(i,1)
3700 GO SUB 2000
3800 RETURN
```

Programowanie gier logicznych [2]

Na siłę, czyli przeszukiwanie

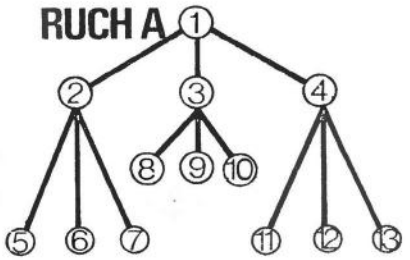
W poprzednim artykule mówiliśmy o strategii wygrywającej, która pozwala konstruować algorytmy grające bezbłędnie. Jednakże dla większości gier nie znamy takiej strategii i ideał gry optymalnej i bezbłędnej jest nieosiągalny.

Podstawą algorytmu grającego w grę, dla której nie jesteśmy w stanie sformułować strategii wygrywającej jest tzw. brute force, co w tłumaczeniu dosłownym znaczy "ślepa (tępa) siła". W języku polskim metoda ta nazywana jest przeszukiwaniem.

Czy pamiętacie historie o Dżinie czy o lampie Alladyna? Wielki i potężny Dżin, który na nasz rozkaz wykona dowolnie trudne i uciążliwe zadanie. Szybkość, wytrwałość, rzetelność to atrybuty, dzięki którym dokonać można rzeczy wielkich.

Każda maszyna ma coś z Dżina, ale wyspecjalizowanego w konkretnym zadaniu. Dopiero komputer to narzędzie wyższego typu, "cały Dżin". Na nasze życzenie będzie wykonywał dowolną operację 1000 razy, 10 000 razy czy jeszcze więcej. Zaraz potem możemy mu rozkazać, aby zrobił co innego, na przykład milion razy. Niespodziewanie, dzięki temu niewielkiemu urządzeniu, wydajemy dyspozycje związane z liczbami, których wielkości wykraczają poza nasze codzienne doświadczenie. Tylko na to, by policzyć od jednego do miliona, potrzebowalibyśmy około 23 dni i nocy.

Wyobraźmy sobie, że musimy znaleźć szyfr sześciocyfrowy spełniający określony warunek. Ilość wszystkich sześciocyfrowych ciągów wynosi dokładnie milion. Jeżeli rozwiązanie mamy znaleźć w niezbyt długim czasie, na przykład w ciągu dnia lub dwóch, nie możemy próbować wszystkich takich ciągów. Co pozostaje? Wytężyć inteligencję i "wziąć się na sposób", by całą działalność istotnie skrócić. Otóż to. A komputer? Ten nie potrzebuje "sposobu". Wystarczy "brute force", siła i szybkość, czyli sprawdzenie



każdego sześciocyfrowego ciągu. Skutek będzie taki sam: rozsądny czas zakończenia obliczeń oraz poprawne rozwiązanie.

Widzimy więc, że inteligencja, której musieliśmy użyć daje się zastąpić przez „brute force".

Z metodą przeszukiwania dla gier związane jest pojęcie tak zwanego drzewa gry, którego przykład widzimy niżej.

Załóżmy, że gra, której drzewo dotyczy, jest dwuosobowa (gracze A i B) oraz że w każdej pozycji gracze mają do wyboru trzy ruchy (lub żadnego, gdy nastąpił koniec gry). Zacznijmy od pozycji, w której ruch należy do gracza A. Odpowiada jej punkt 1, nazywany korzeniem drzewa. Trzy możliwe ruchy gracza A w pozycji 1 wyrażają węzły 2, 3 i 4. Z każdego z nich odchodzą gałęzie (takie są fachowe określenia elementów drzewa również w informatyce) do następnych trzech węzłów, które stanowią trzy możliwe odpowiedzi gracza B na każdy z trzech ruchów A. Węzły przedstawiają konkretne pozycje z gry, gałęzie ruchy, które do danej pozycji doprowadziły. Na przykład do pozycji 7 dochodzi się z pozycji 1, gdy gracz A zagra ruch 1-2 (tak oznaczmy ruch, który „przeprowadza" pozycję 1 w 2), a gracz B ruch 2-7. Po ruchu B, który doprowadził do pozycji od 5 do 13, kolej na gracza A – gdyby uwzględnić jeszcze jego możliwe ruchy, trzeba byłoby dodać 27 węzłów i gałęzi. Jak widać ilość pozycji rośnie potęgowo, a każdy poziom drzewa odpowiada kolejnym ruchom graczy A i B.

Wysokością drzewa nazywamy ilość takich poziomów, stopniem – ilość gałęzi odchodzących z każdego węzła. Ilość wszystkich węzłów drzewa o stopniu s i wysokości h określa liczba:

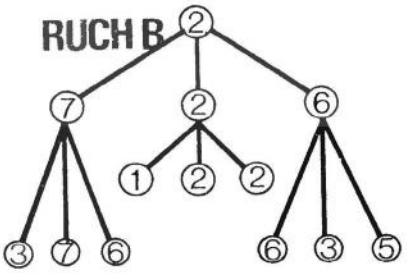
$$s*(s^n-1)/(s-1)+1 = s^0+s^1+s^2+...+s^n.$$

Drzewo gry przedstawia graficznie wszystkie możliwe sytuacje i ruchy od zadanej pozycji do określonej wysokości. Nic poza nim nie może się zdarzyć. Jest to model przewidywania sytuacji, które będą konsekwencją pozycji wyjściowej. W ten sposób programy próbują znaleźć najlepszy ruch. A nie jest to praca łatwa.

Poprzednio mówiliśmy o złożoności gry i przytaczaliśmy niewiarygodnie duże liczby mówiąc o warcabach, szachach i go. Jak się one mają do drzewa gry?

Jeżeli mamy kompletne drzewo od korzenia reprezentującego pierwszy ruch do węzłów kończących grę, w których wiadomo kto wygrał, wówczas złożonością gry będzie ilość wszystkich dróg prowadzących od korzenia do węzłów końcowych (zwanych liściami drzewa). Liczba ta równa jest ilości węzłów końcowych, gdyż do każdego z nich istnieje jedna i tylko jedna droga prowadząca od korzenia. Wiemy, że liczby te są zbyt duże nawet dla najszybszych komputerów.

Wiele dróg w drzewie gry odpowiada ruchom złym, samobójczym, fatalnym itd., jednym słowem takim, które człowiek w swojej analizie odrzuca natychmiast, czy wręcz w ogóle nie bierze ich pod uwagę. Techniki eliminujące takie



ruchy z analiz komputera przyczyniają się do zredukowania drzewa. Pozwala to rozważyć więcej ruchów "rozsądnych" nie wydłużając czasu obliczeń, dzięki czemu program gra lepiej.

Jak przebiega wybór ruchu na podstawie drzewa gry? Oczywiście samo "chodzenie" po drzewie nie wystarcza. Potrzebna jest tu funkcja oceniająca, która by każdej pozycji przypisywała określoną liczbę. Liczba ta wyrażałaby "dobroć" danej pozycji z punktu widzenia jednego bądź drugiego gracza. Najprostszą funkcją oceniającą może być funkcja, która pozycji wygranej przypisuje pewną stałą P, pozycji przegranej -P (wygrana przeciwnika), a innym 0. Wówczas algorytm wybierający ruch będzie unikał zagrań prowadzących do przegranej, wybierał takie, po których wygrywa, a poza tym będzie grał losowo.

Podstawą algorytmów, które na podstawie drzewa gry wybierają ruch, jest metoda zwana minimaksem. Stosuje się ją dla dowolnych funkcji oceniających. Jako przykład niech posłuży rys. 2.

Spójrzmy na ostatnie końcowe węzły tego drzewa. Każdemu z nich przypisana jest pewna liczba będąca wartością funkcji oceniającej dla pozycji odpowiadającej danemu węzłowi. Pamiętajmy, że są to pozycje bezpośrednio po ruchu gracza B, a całe drzewo rozpatrujemy pod kątem gracza A (jego ruchy wyrażają gałęzie wychodzące z korzenia).

Pierwsza trójka liczb to 3, 7 i 6. Skoro opisują one "dobroć" pozycji po ruchu gracza B, to który z ruchów wybrałby B? Oczywiście ten z największą wartością, gdyż daje mu on najlepszą pozycję. Nie należy zakładać, że przeciwnik popełni błąd. Dlatego też wartość 7 przypisujemy węzłowi odpowiadającemu pozycji po ruchu gracza A, a znajdującemu się bezpośrednio przed trzema rozpatrywanymi węzłami. Czy słusznie? Ano tak, gdyż tyle jest dla nas wart ruch do tej pozycji doprowadzający.

Podobnie postępujemy dla pozostałych trójek liczb przenosząc wyżej (zwracając – jak mówią informatycy) wartość największą. Otrzymujemy teraz trzy liczby na poziomie gracza A: 7, 2, 6. Którą powinniśmy wybrać? Największą czy najmniejszą? Jeżeli wybierzemy ruch 1-2 z rys. 1, wówczas B będzie miał szansę wykonać ruch 2-6 i osiągnąć pozycję o wartości 7. Jeżeli natomiast wybierzemy ruch 1-3 z rys. 1, wówczas B nie może zrobić nic ponad osiągnięcie pozycji o wartości 2. Wynika z tego jasno, że powinniśmy wybrać wartość minimalną 2, czyli ruch 1-3 z rys. 1.

Nasze drzewo jest raczej małe. Jednak metoda pozostaje ta sama: zwracamy naprzemiennie wartość najmniejszą i największą do węzła położonego bezpośrednio wyżej. W ten sposób otrzymujemy pewną liczbę w korzeniu drzewa, która jest wartością danej pozycji, ale obliczoną w stosunku do tego co się może wydarzyć, na podstawie przewidywania ruchów. Oczywiście im poziom większy, tym obliczenia rzetelniejsze. Ale pamiętajmy o czasie, o potęgowym wzroście liczby węzłów przy zwiększaniu wysokości. Z powyższego wynika również, jak wielką rolę przy wyborze ruchu odgrywa funkcja oceniająca. Tu też mieści się wiele możliwości ulepszania programów grających.

W następnym odcinku omówiona będzie technika "obcinania zbędnych gałęzi" pozwalająca zaoszczędzić wiele czasu. Może niektórzy z Czytelników, gdy przyjrzą się dokładniej drzewu z rys. 2, zauważą pewne możliwości skrócenia obliczeń.

JANUSZ KRASZEK

PROGRAMOWANIE GIER LOGICZNYCH

Alpha-beta pruning

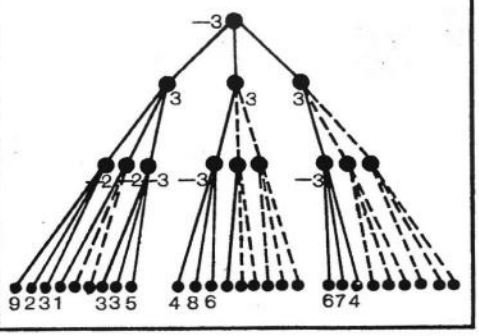
```
main()
{int i, j, val;
for(i=P3+1;i<P10;i++) tab[P12][i]=P3+P6;
for(i=0;i<3;i++){pi[i][0]=-(i+1);pi[i][1]=0; pi[i][2]=i+1;
                 pj[i][0]=pj[i][1]=pj[i][2]=1+i;}
for(i=P3+1;i<=P3+P6;i++)
    for(j=P3+1;j<P10;j++) MASKA[i][j]=1;
rep : printf("Podaj wysokość drzewa \n");
      scanf("%d",&poziom);
      if(!(poziom>=1 && poziom<20)) goto rep;
poziom++; kol=1;
printf("Czy chcesz zagrać pierwszy? (1 lub 0) \n");
scanf("%d", &i);
if(i) {j=czytajruch(); WSTAW(j+P3,kol); kol=-kol;}
WSTAW(4+P3,kol);kol=-kol;drukuj(4+P3);
loop: j= czytajruch(); j+=P3;
      WSTAW(j,kol);  drukuj(j); win=ocenapoz(j);
      if(win){printf("PRZEGRAŁEM \n"); exit();}
      kol=-kol;
      val=alphabeta();
      if(val == 0) {printf("PODDAJĘ SIĘ \n"); exit();}
      WSTAW(val,kol);  drukuj(val);
      win=ocenapoz(val);
      if(win){printf("WYGRAŁEM \n"); exit();}
      kol=-kol; goto loop;
 }
 czytajruch()
 {int j;
 for(j=P3+1;j<P10;j++) if(tab[P12][j]> P3) goto podaj; remis();
podaj:  printf("Podaj ruch (1 do 7) \n");
 scanf("%d", &j);
 if(j<1 || j>7 || tab[P12][j+P3]<=P3) goto podaj; return(j);}
 drukuj(j)
 int j;
 {int p,q;
  for(p=P3+1;p<=P3+P6;p++)
    { for(q=P3+1;q<P10;q++)
      if(tab[p][q]==0) printf("| |");
      else (tab[p][q]==1? printf("|X|") :printf("|O|"));
      printf("\n");}
 printf("\n Ostatni ruch w kolumnie %d \n",j-P2);}
remis()
{printf("REMIS  \n");exit();}
focena(i,j)
int i,j;
{int p, k, s, ti, tj;
int v[3];
s=(tab[i+1][j]==-kol)? 2:0;
for(p=0;p<3;p++)
    {v[p]=0; for(k=0;k<3;k++)
               {ti=i+pi[k][p]; tj=j+pj[k][p];
                if(MASKA[ti][tj])
                   {if(tab[ti][tj]==kol) break;
                    else v[p]+=(tab[ti][tj]? P5 : P2);}
               else break;}}
for(p=0;p<3;p++)
    { for(k=0;k<3;k++)
               {ti=i-pi[k][p]; tj=j-pj[k][p];
                if(MASKA[ti][tj])
                   {if(tab[ti][tj]==kol) break;
                    else v[p]+=(tab[ti][tj]? P5 : P2);}
               else break;}}
for(p=0;p<3;p++) s+=v[p];
return(s);}
```

```
#include <stdio.h>
#define P3 2
#define P12 12
#define P10 10
#define P6 6
#define P5 5
#define P2 2
#define P100 100
int kol, win, first, poziom;
int tab[13][13], MASKA[13][13];
int pi[3][3],pj[3][3];
#define WSTAW(k,c) (tab[tab[P12][k]][k]=c, tab[P12][k]--)
#define USUN(k)    (tab[P12][k]++, tab[tab[P12][k]][k]=0)
next(k1)
int k1;
{int k;
for(k=k1+1;k<P10;k++)
    if(tab[P12][k]>P3) return(k);
return(NULL);}
generuj(l)
int l;
 { first=(l>poziom || win)? NULL : next(P3);}
ocenapoz(j)
int j;
{int k, i;
i=tab[P12][j]+1;
if(tab[i+3][j]==kol && tab[i+2][j]==kol && tab[i+1][j]==kol)
                              return(P100);
for(k=0;k<3;k++)
 {  if(tab[i+pi[0][k]][j+pj[0][k]] != kol) goto E3;
    if(tab[i+pi[1][k]][j+pj[1][k]] != kol) goto E2;
    if(tab[i+pi[2][k]][j+pj[2][k]] != kol) goto E1;
    return(P100);
E3: if(tab[i-pi[2][k]][j-pj[2][k]] != kol) continue;
E2: if(tab[i-pi[1][k]][j-pj[1][k]] != kol) continue;
E1: if(tab[i-pi[0][k]][j-pj[0][k]] != kol) continue;
    return(P100);}
return(0);}
alphabeta()
{int ply,a[20],r[20], mem=0;
ply=2; a[0]=a[1]=-P100;
first=next(P3); if(first==NULL) remis(); else goto ini;
nowe_ruchy:  generuj(ply);
ini:         r[ply+1]=first;
if(r[ply+1]==NULL){
a[ply]=ply-(win?win:focena(tab[P12][r[ply]]+1,r[ply]));}
   else
   {   a[ply]=a[ply-2];
nizej:  ply++; WSTAW(r[ply],kol); win=ocenapoz(r[ply]); kol=-kol;
         goto nowe_ruchy;
zbadaj: if(-a[ply+1]>a[ply])
          {a[ply]=-a[ply+1];
           if(a[ply+1]<=a[ply-1]) {USUN(r[ply+1]);
                                   goto koniec_poziomu;}}
        USUN(r[ply+1]);
        r[ply+1]= next(r[ply+1]);
        if(r[ply+1]) goto nizej; }
koniec_poziomu:  ply--; if(ply>=2){ kol=-kol;
                 if(ply==2 && -a[3]>a[2]) mem=r[3];
                 goto zbadaj;}
return(mem);}
```

W poprzednim odcinku mówiliśmy o metodzie generowania ruchu na podstawie drzewa gry, zwanej minimaks. Poważną niedoskonałością tej metody jest sprawdzanie zawsze całego drzewa. Bywają sytuacje, kiedy bez wpływu na ostateczny wynik, można szybciej zakończyć przeszukiwanie pewnych fragmentów drzewa. Na przykład na rys. 2 z poprzedniego odcinka można było nie badać ostatnich dwóch liści (węzłów końcowych) o wartościach 3 i 5. Dlaczego? Mam nadzieję wyjaśnić to w dalszym ciągu artykułu.

Obecnie najlepszą metodą "obcinania" zbędnych gałęzi jest algorytm zwany alpha-beta pruning. Pruning oznacza obcinanie, a alpha i beta (dalej będę pisał alfa zamiast alpha) to dwie liczby (zmienne), przy pomocy których dokonuje się obcięcia. Spójrzmy na rys. 1 prezentujący wynik działania tego algorytmu. Linie przerywane wskazują na węzły, które zostały pominięte (obcięte). Łatwo zauważyć, jak duże oszczędności czasowe może dać ta metoda.

Powiedzmy w końcu na czym ona polega. Istotne jest, żeby pamiętać zasadę działania algorytmu minimaks, która omówiona została w poprzednim odcinku.



PROGRAMOWANIE GIER LOGICZNYCH

37

Spójrzmy na pierwsze trzy liście drzewa: 9, 2 i 3. Pamiętamy, że każdemu węzłowi odpowiada pewna pozycja w grze. Zalety pozycji wyrażają liczby przypisane liściom przez funkcję oceniającą. Ustalmy jeszcze rzecz następującą. Z każdego węzła odchodzą trzy gałęzie, które wyrażają trzy i tylko trzy możliwe ruchy gracza. Nazwijmy je odpowiednio L, S, P od gałęzi lewej, środkowej i prawej. W ten sposób każdy węzeł daje się opisać jako ciąg liter L, S i P, które odpowiadają ruchom, jakie trzeba wykonać, aby do danej pozycji (węzła) dojść. Na przykład węzeł LPP, to liść o wartości 5.

Korzeń reprezentuje konkretny ruch (L, S lub P), jaki ma zostać oceniony za pomocą tego drzewa. Stąd poziomy B i D odpowiadają pozycjom po ruchu przeciwnika. Funkcja oceniająca rozpatruje pozycje z "naszego" punktu widzenia, dlatego też pozycja o minimalnej wartości jest dla naszego przeciwnika najkorzystniejsza. Dlatego z liczb 9, 2 i 3 zapisujemy w poziomie C wartość 2, ale ze zmienionym znakiem. Czyni to z metody minimaks tzw. megamaks i znacznie ułatwia zapis algorytmu: zamiast na przemian szukać liczby najmniejszej i największej, zawsze szukamy najmniejszej, po czym zapisujemy ją wyżej ze zmienionym znakiem.

Węzeł LL otrzymuje wartość -2. Zostaje ona także zapamiętana pod zmienną beta. Następnie badana jest pozycja LSL, która oceniona została na 1 punkt. Dlaczego możemy pominąć dalsze węzły tego poddrzewa – LSS i LSP?

Proszę pamiętać, że zapisujemy wyżej wartość najmniejszą, która dla tego poddrzewa będzie mniejsza bądź równa 1. A skoro tak, to po zmianie znaku w poziomie C wartość ta i tak przegra w konkurencji zapisania do poziomu B, gdyż w węźle LL mamy już -2. Ujmując tę zależność w postaci warunku występującego w algorytmie (patrz procedura alfabeta dołączonego programu, drugi wiersz pod etykietą zbadaj) powiemy, że najmniejsza dotychczas wartość z tego poddrzewa – w tym przypadku 1 – jest mniejsza lub równa minus beta. W węźle LS zapisujemy wartość beta i ignorujemy resztę drzewa.

Zależności pomiędzy wartościami węzłów różnych poziomów badane są w algorytmie przy pomocy zmiennych alfa i beta. Stanowią one odpowiednio dolne i górne ograniczenie akceptowanych wartości węzłów. Stąd zwykłe porównania decydują o tym, które gałęzie zostaną "obcięte" a które nie. Alfa i beta zmieniają się w zależności od poziomu i poddrzewa.

Inaczej jest z poddrzewem LP, czyli węzłami LPL, LPS i LPP. Trzeba je wszystkie zbadać, gdyż najmniejsza wartość tego poddrzewa ma szansę "zajść" wysoko – nie zachodzi warunek przytoczony wyżej, który spowodował "obcięcie" gałęzi LSS i LSP. Po sprawdzeniu całego poddrzewa LP, pojawia się nowy "lider" w postaci wartości 3, który zdobywa kolejne poziomy aż do samego korzenia. Zmienna alfa przyjmuje teraz wartość – 3, co oznacza, że żadna mniejsza liczba nie ma prawa znaleźć się w korzeniu.

W ten sposób zakończone zostało przeszukiwanie lewego poddrzewa. Przyjrzyjmy się teraz węzłom LPL, LPS, LPP o wartościach 4, 8 i 6. Widzimy, że po znalezieniu najmniejszej z nich (4), całe środkowe poddrzewo zostało pominięte. Czy słusznie? Wartość 4 z poziomu D zapisana byłaby do poziomu C (ze zmienionym znakiem oczywiście) i tam miałaby duże szanse na "awans" jeszcze wyżej. Lecz na poziomie B wartość 3 zapamiętana w węźle L nie daje jej żadnych szans na zdobycie "szczytu" w postaci poziomu A. "Niech zostanie na miejscu, skoro nie może dojść do samego końca" – wydaje się być tajemnicą sukcesu tej metody. Analogicznie jest z poddrzewem prawym.

Jak w kategoriach ruchów poszczególnych graczy wytłumaczyć redukcję środkowego poddrzewa? Na bazie założenia, że przeciwnik ma trochę rozsądku. Bo jeśli będzie na tyle naiwny, że w pozycji reprezentowanej przez korzeń zagra ruch S (z poziomu A), wówczas my grając L (z poziomu B), pozostawimy mu co najwyżej ruch L (z poziomu C) dający pozycję o wartości 4. Jest to ewidentna strata dla niego, bo gdyby zagrał L na samym początku, to najprawdopodobniej osiągnąłby pozycję o wartości 3 (czyli lepszą).

Algorytm ten skonstruowali Donald E. Knuth i Ronald W. Moore w 1975 roku[1]). Od tego czasu jest zupełnie nie do pomyślenia stosowanie metody minimaks bez optymalizacji, jaką dostarcza alfa-beta pruning.

Od czego zależy ilość "obciętych" gałęzi? Jest to bardzo istotna sprawa. W przypadku tego rysunku zbadanych zostało około połowy wszystkich węzłów, gdy na rys. 2 z poprzedniego odcinka można było pominąć tylko dwa węzły. Różnice te powoduje uporządkowanie węzłów. Najkorzystniejszym układem jest taki, w którym najlepszy ruch jest rozpatrywany na początku. Zauważmy, że na rys. 1 w korzeniu drzewa znalazła się wartość pochodząca z lewego poddrzewa. Był to więc układ bliski optymalnemu. W programach grających w szachy stosuje się krótkie wstępne przeszukiwanie, którego celem jest tylko uporządkowanie węzłów. Dopiero potem uruchamia się właściwy algorytm alfa-beta, który w pierwszej kolejności bada ruch "prawdopodobnie" najlepszy. W sumie alfa-beta pruning umożliwia przeszukiwanie większych drzew w tym samym czasie, co istotnie wpływa na siłę i skuteczność gry komputerów.

```
function alphabeta: integer;
label 1111, 2222, 3333, 4444, 5555;
{etykieta 1111 to nowe_ruchy.
          2222 to ini,
          3333 to nizej,
          4444 to zbadaj,
          5555 to koniec_poziomu.}
var
   ply, mem: integer;
   a:    array[0..20] of integer;
   r:    array[0..20] of integer;
begin
mem=0; ply=2;
a[0]=a[1]=-P100;
first=next(P3); if first=NULL then remis else goto 2222;
1111:  generuj(ply);
2222:  r[ply+1]=first;
if r[ply+1]=NULL then
begin
     if win=0 then a[ply]=ply-focena(tab[P12,r[ply]]+1,r[ply])
     else a[ply]=ply-win;
end
else
begin
    a[ply]=a[ply-2];
3333:    ply=ply+1; WSTAW(r[ply],kol); win=ocenapoz(r[ply]);
         kol=-kol;
         goto 1111;
4444:    if -a[ply+1]>a[ply] then
    begin
       a[ply]=-a[ply+1];
       if a[ply+1]<=a[ply-1] then
       begin
         USUN(r[ply+1];
         goto 5555;
       end;
    end;
    USUN(r[ply+1]);
    r[ply+1]= next(r[ply+1]);
    if r[ply+1] then goto 3333;
end;
5555:    if ply>=2 then
         begin
             kol=-kol;
             if (ply=2) and (-a[3]>a[2]) then mem=r[3];
             goto 4444;
         end;
alfabeta = mem;
end.
```

Programowanie gier logicznych [4]

Metody heurystyczne

Pomyśleć...

fachowa wiedza, którą człowiek niewątpliwie posiada – przecież w trakcie gry nie zastanawiamy się nad wszystkimi możliwymi ruchami. Problem jedynie w tym, by wiedzę tę skodyfikować i zapisać w postaci algorytmu.

Powyższy wywód ma przekonać Czytelników, że w zasadzie wiemy już wszystko, mimo że wiemy tak niewiele. Reszta to już własna wiedza i inwencja. Może niektórzy czują się zawiedzeni. Jak to, czy to już cała "mądrość" komputerów? Gdzież te szokujące rozwiązania, "generalne" wyniki?

Komputer jest tylko brute-force: siła, szybkość, dokładność, i "pracowitość". Reszta to mozolna i trudna praca człowieka. Genialne programy szachowe, te z po-

Ostatni odcinek zakończyliśmy przykładem zastosowania algorytmu alfa-beta. Chciałbym zwrócić uwagę Czytelnika na fakt, że jest to algorytm nietrywialny i poznanie go wymaga cierpliwości. Po drugie, procedura "alfabeta()" tego programu, po niewielkich przeróbkach, może być zastosowana w innej grze. Stanowi swego rodzaju konstrukcję, do której dobudowuje się inne procedury, jak: USUŃ, WSTAW, generuj, next itd. Zalety metody alfa-beta gwarantują otrzymanie dobrych wyników, o ile reszta zrobiona będzie prawidłowo. Wobec tego możemy uznać, że kwestia przeszukiwania drzewa dla dowolnej gry została rozwiązana. Twórczego wysiłku natomiast wymagają pozostałe elementy istotne dla programu. Takim najważniejszym, i najtrudniejszym zarazem, jest funkcja oceniająca. Gdybyśmy umieli konstruować "idealne" funkcje oceniające, nie trzeba byłoby przeszukiwać drzewa gry!

W poprzednich odcinkach, przy różnych okazjach, wskazywaliśmy czynniki składające się na jakość programów grających, czyli na siłę ich gry. Wymieńmy je teraz razem (chodzi o gry, dla których nie znamy strategii wygrywającej). Są to:

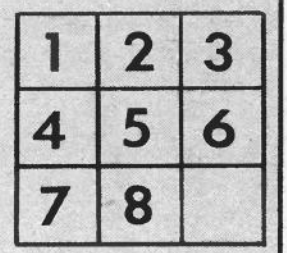
- Metody przeszukiwania drzewa (minimaks, alfa-beta).
- Przeszukiwanie wstępne w celu uporządkowania węzłów przed algorytmem alfa-beta.
- Adekwatność funkcji oceniającej.

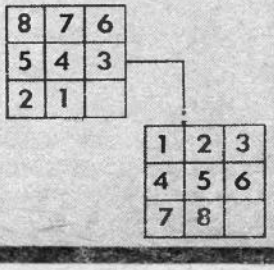
Metodom przeszukiwania poświęciliśmy już sporo miejsca. O funkcji oceniającej z kolei nie możemy powiedzieć zbyt wiele. Jest ona dla każdej gry inna i zależy przede wszystkim od wiedzy programującego o danej grze. Dlatego zespoły pracujące nad coraz lepszymi programami szachowymi mają w swych składach wybitnych szachistów.

Przeszukiwanie wstępne polega w znacznym stopniu na umiejętności rozpoznania, niemal na wyczucie, ruchów lepszych i gorszych. Stosuje się tu funkcje (procedury), które na podstawie krótkiej analizy porządkują ruchy (węzły). W tym przypadku również niezbędna jest

```
#include<stdio.h>
#define P3 3
#define P9 9
#define TRUE 1
#define FALSE 0
int Pdoc[P9],Pstart[P9],r[900];
typedef struct poz {int Pr[P9],nrsp,id,ildzieci;
                    struct poz *dzieci[4], *ojciec;} POZ;
POZ *p, *px, *start;
int mem,poziom_max, l_rozw = 0,  l_ruchow = 0;
/* ilość ruchów w danej pozycji spacji */
int Tr[] ={2,3,2,3,4,3,2,3,2};
int T[4][P9]; /* możliwe ruchy */

druk(P)
int P[];
{int k,l;
for(k=0;k<P3;k++){
    for(l=0;l<P3;l++) printf(" %d ",P[k*P3+l]);
    printf("\n");}}

main()
{int spacja=-1, k,l,i,w,z,x,v,vx,rowne=1,s1,s2;

/*inicjalizacja tablicy ruchów spacji*/
for(k=0;k<P9;k++){T[1][k]=1;T[0][k]=-3;T[3][k]=0;T[2][k]=0;}
T[0][0]=T[0][1]=1;T[0][2]=T[1][0]=T[1][1]=T[1][5]=3;
T[1][2]=T[1][8]=T[2][1]=T[2][5]=T[2][7]=T[3][4]=-1;
T[2][3]=T[2][4]=3;

czyt:  printf("Podaj ciąg wejściowy (zero zamiast spacji) \n");
for(k=0;k<P9;k++){scanf("%d",&i); Pstart[k]=i;}
printf("Podaj ciąg docelowy \n");
for(k=0;k<P9;k++){scanf("%d",&i); Pdoc[k]=i;}
printf("\n Czy dobry pucel wejściowy? \n"); druk(Pstart);
printf("\n Czy dobry pucel wyjściowy? ( 1 lub 0) \n"); druk(Pdoc);
scanf("%d",&i); if(i==0) goto czyt;
for(k=0;k<P9;k++){
    if(Pstart[k]==0)spacja=k;
    if(Pdoc[k]==0) v=k;
    if(Pstart[k] != Pdoc[k]) rowne=0;}
if(rowne){printf("Sam dałeś rozwiązanie! \n"); exit();}
if(spacja ==-1){ printf("Brak spacji, czyli zera \n"); goto czyt;}

/*badanie znaku permutacji*/
Pstart[spacja]=P9; Pdoc[v]=P9;
s1=x==1||x==3||x==5||x==7;  s1+=signum(Pstart);
s2=v==1||v==3||v==5||v==7;  s2+=signum(Pdoc);
k=s1/2;  l=s2/2;
if(s1-2*k != s2-2*l) {printf("Dwa układy różnej parzystości! \n");
     printf("PRZEJSCIE NIEMOZLIWE!!!  \n"); exit();}
Pstart[spacja]=0; Pdoc[v]=0;

podaj: printf("Podaj maksymalny poziom \n");
scanf("%d",&poziom_max);
if(poziom_max<0 || poziom_max>100) goto podaj;

/*generowanie i zapis drzewa*/
p=malloc(sizeof(POZ)); start=p; p->ojciec=NULL;
for(k=0;k<P9;k++)p->Pr[k]=Pstart[k]; p->nrsp=spacja;
```

```
/*główna pętla*/
generuj_nast:  l=0; v=Tr[p->nrsp];
 for(i=0;i<v;i++)
{    w=T[i][p->nrsp];
     px=malloc(sizeof(POZ));
     for(k=0;k<P9;k++) px->Pr[k]=p->Pr[k]; /*przepisywanie pucla*/
     px->Pr[p->nrsp]=p->Pr[x=p->nrsp+w];   /*przesunięcie */
     px->Pr[x]=0;                          /*   spacji    */
     if(czydobry()){ px->nrsp=x; px->ojciec=p; p->dzieci[l++]=px;}
          else free(px,sizeof(POZ));
} /*for i */
p->ildzieci=l;
E2:  if(p->ildzieci<=0) goto back;

/*Wybieranie najlepszego ruchu*/
mem=-1;  v=P9+1;
for(k=0;k<p->ildzieci;k++)
    if(v>(vx=focena(p->dzieci[k]))) {v=vx; mem=k;}
p->id=mem; /*Ten węzeł został wybrany*/

if(v==0)
 {px=p->dzieci[p->id];px->dzieci[0]=NULL;
  px->id=0;koniec(); goto cofnij;}
if(++l_ruchow<poziom_max){p=p->dzieci[mem]; goto generuj_nast;}

/*Koniec głównej pętli, początek cofania się w drzewie*/
cofnij: for(k=0;k<p->ildzieci;k++)
         free(p->dzieci[k],sizeof(POZ));
    l_ruchow--;
back: p=p->ojciec;
      if(!p){rozbrak();goto podaj;}
      w=p->id;free(p->dzieci[w],sizeof(POZ));
      p->dzieci[w]=p->dzieci[--(p->ildzieci)];
      l_ruchow--;goto E2;
}/*main end */
koniec()
{POZ *x;
int k,l,w=0;
poziom_max=l_ruchow++;
printf("Mam rozwiązanie, wciśnij klawisz  \n"); scanf("%d",&k);
for(x=start;x;x=x->dzieci[x->id])
{   printf("\n KROK  %d \n",w++);
    druk(x->Pr);}
for(l=l_rozw;l>=0;l--)
    r[l+1]=r[l];
r[0]=l_ruchow;
l_rozw++;
printf("Czy chcesz inne rozwiązania?  (1 lub 0) \n");
scanf("%d",&k);
if(k==0) rozbrak();}

rozbrak()
{int k;
if(l_rozw==0){printf("Brak rozwiązań dla %d poziomów\n",poziom_max);
          return;}
printf("Oto najszybsze rozwiązania: \n");
for(k=0;k<l_rozw;k++)  printf("%d ", r[k]);
exit();}
czydobry()
/*procedura sprawdza czy dany układ nie pojawił się wcześniej*/
{POZ *x;
int k,z;
for(x=start;x!=p;z=x->id,x=x->dzieci[z])
   {  for(k=0;k<P9;k++)
         if(x->Pr[k] != px->Pr[k]) goto en;
     return(FALSE);
en:    /*sprawdzamy dalej */
  }
return(TRUE);}

focena(a)
POZ *a;
{int v=0, k;
for(k=0;k<P9;k++)if(Pdoc[k]){
    if(Pdoc[k]!=a->Pr[k] )v++;}
return(v);}
signum(P)
int P[];
{int k,l,s=0;
for(k=0;k<P9-1;k++)
    for(l=k;l<P9;l++) if(P[k]>P[l])s++;
return(s);}
```





wodzeniem rywalizujące z najlepszymi graczami, to przede wszystkim wspaniałe funkcje oceniające i porządkujące. To umiejętne skojarzenie szybkości maszyn i ludzkiej wiedzy "zaszytej" w liczbach i algorytmie.

Jednak pomysłowości człowieka nie ma końca. I tu mają trochę racji ci zawiedzeni. Jest coś więcej – może będzie coś więcej! Ale o tym, czyli o kierunkach rozwoju w informatyce, powiemy następnym razem. Tymczasem chciałbym jeszcze pozostać przy poprzednim temacie, gdyż przeszukiwanie drzew to nie tylko element służący programowaniu gier. Wszędzie tam, gdzie spośród wielu możliwości trzeba wybrać jedną, lub kilka, można zastosować tę metodę. Na przykład gdy mamy ustawić osiem hetmanów na szachownicy, by żaden nie stał w polu bicia innego, lub znaleźć najkrótszą drogę pomiędzy pewną ilością miast (tzw. problem komiwojażera). Każdy tego typu problem można rozwiązać przy pomocy drzewa: wygenerować wszystkie możliwe układy i sprawdzić, które stanowią rozwiązanie. Co jednak czynić, gdy problem jest zbyt złożony i rozwiązanie tą metodą jest niemożliwe? W takich przypadkach trzeba odwołać się do metod heurystycznych. Ogólnie, heurystyka polega na "inteligentnym" (w przeciwieństwie do brute-force) rozwiązywaniu problemów, zorientowanym na osiągnięcie celu. W tym przypadku oznacza po prostu wybieranie tych gałęzi drzewa, które na podstawie pewnej funkcji zdają się przybliżać nas do celu.

Wyjaśnimy to sobie na przykładzie tzw. puzzle-8, który będziemy nazywali puclem-8. W kwadracie 3 na 3 mamy osiem ruchomych numerków i jedno wolne miejsce, spację. Możemy przesuwać numerki pionowo i poziomo, ale tylko w miejsce spacji. Bardziej znany jest pucel-15, czyli kwadrat 4 na 4 działający na tej samej zasadzie.

Nie jest to gra w ścisłym sensie, gdyż nie ma przeciwnika, poza, być może, własną niecierpliwością. Chodzi w nim o ułożenie zadanej kolejności liczb. Numerki składają się z cyfr od 1 do 8. Jeżeli przyjmiemy, że puste miejsce to cyfra 9, wówczas przesuwanie numerków będzie permutowaniem ciągu 1, 2, 3, 4, 5, 6, 7, 8, 9. W przypadku pucla wygląda to tak:

Nam wygodniej jest utożsamiać go z ciągiem, co wychodzi na to samo. Liczba wszystkich różnych ułożeń cyfr od 1 do 9 równa jest liczbie permutacji tego ciągu, czyli 9! (9 silnia).

Kiedy w 1878 roku po raz pierwszy skonstruowano pucel, nie znano teorii związanej z permutacjami. Ogłoszono wówczas wysokie nagrody dla tych, którzy z ciągu od 1 do 9 zdołają "przejść" do zadanych z góry permutacji. Później okazało się, że były to układy "niemożliwe", do których nie można "dojść". Są dwie równe i rozłączne klasy (rodzaje) permutacji zamknięte na operacje "przejścia", co oznacza, że z żadnego układu jednej klasy nie można "przejść" do żadnego układu drugiej. Własnością, która decyduje o przynależności do klasy, jest tzw. znak lub parzystość permutacji, która zależy od tego, czy dana permutacja zawiera parzystą czy też nieparzystą liczbę inwersji (odwróceń, takich, że liczba mniejsza stoi za większą). W przypadku pucla

34

trzeba jeszcze dodać 1 do liczby inwersji, jeżeli spacja stoi w miejscu o parzystym numerze (patrz stosowny fragment programu, który liczy znaki zadanych permutacji).

Załóżmy, że z ciągu 1, 2, 3, 4, 5, 6, 7, 8, 0 (zero zamiast spacji) chcemy "przejść" do permutacji 8, 7, 6, 5, 4, 3, 2, 1, 0. Lub inaczej, chcemy, żeby komputer znalazł najkrótsze rozwiązanie, czyli najmniejszą liczbę ruchów, jakie trzeba wykonać. Ile mamy tu możliwości do przeanalizowania? Nieskończenie wiele! Można przecież przesuwać numerki bez końca. Dlatego na początek ustalmy maksymalną liczbę ruchów, która powinna wystarczyć do znalezienia rozwiązania. Będzie to granica, do jakiej rozwijane będzie drzewo ruchów (zmienna "poziom-max" w programie). Żeby wyeliminować cykle, czyli ciągi ruchów odtwarzające poprzednią sytuację, trzeba porównywać nowy układ z każdym jego poprzednikiem (procedura "czydobry()"). Liczba możliwych posunięć waha się od 1 do 4 w zależności od położenia spacji (tablica "T[ ] [ ]" w programie).

Czy powinniśmy przeszukiwać całe drzewo, wszystkie możliwe i nie powodujące cykli układy? Taka metoda byłaby zbyt wolna i nie dostarczałaby zadowalających wyników. A jaki mamy wybór?

Wiemy, że olbrzymia część drzewa ruchów nie zbliża nas do rozwiązania i można byłoby ją pominąć. Słusznie, to jest właśnie heurystyka. Pozostaje, jak zwykle zresztą, drobiazg zapisania w postaci algorytmu metody rozpoznawania właściwych dróg (węzłów). Pomińmy na razie ten szczegół i załóżmy, że mamy już funkcję "focena()", która wybiera z nowo wygenerowanych węzłów "najlepszy". Musimy jeszcze ustalić, co zrobimy, gdy mimo zalet naszej funkcji i po wykonaniu maksymalnej liczby ruchów (zmienna "poziom-max") nie otrzymamy pożądanego rozwiązania. Chodzi o fragment w programie rozpoczynający się etykietą "cofnij". Jest to ważna technika programowania, tzw. backtracking, czyli wycofywanie. Wycofujemy ostatnio wykonane ruchy, które nie doprowadziły nas do celu i zamiast nich wykonujemy inne – jak w metodzie prób i błędów.

Zajmijmy się teraz poważniej funkcją odpowiedzialną za wybór węzła. Stanowi ona podstawę metody heurystycznej i od niej zależy skuteczność znajdywania rozwiązań. Tak jak w przypadku innych omawianych tu funkcji, nie jest łatwo ją wymyślić. Funkcja "focena()" jest zbyt elementarna. Działa na prostej zasadzie zliczania numerków stojących na pozycjach docelowych. Oznacza to, że spośród wszystkich możliwych ruchów w danej pozycji wybiera taki, który ma największą liczbę "dobrze stojących" numerków. Niezbyt często prowadzi ona do celu bezpośrednio, bez wycofywania. Do poziomu 18, czyli do osiemnastoruchowych różnic pomiędzy permutacjami, znajduje w miarę szybko rozwiązania. Jednak – na czym mi najbardziej zależało – nie jest w stanie dać rozwiązania następujących permutacji:

Skądinąd wiem, że jest ono 30-ruchowe. Próbowałem kilku modyfikacji tej funkcji – na przykład uzależniałem generowanie nowych węzłów od numeru poziomu i pewnej zadanej stałej. Udało mi się osiągnąć rozwiązanie nieco innego układu, również 30-ruchowe i to po 40 sekundach (na IBM PC), ale to wszystko. Dlatego apeluję do Czytelników, którzy będą mogli poświęcić temu więcej czasu, by próbowali wymyślić taką funkcję heurystyczną, która w rozsądnym czasie prowadziłaby do rozwiązania podanych wyżej permutacji.

JANUSZ KRASZEK