Драган Урошевић

BACKTRACKING

Кроз ово излагање покушаћемо да презентирамо једну програмску технику која се примењује за решавање задатака или проблема који би се могли сврстати у област вештачке интелигенције. Основна карактеристика тих проблема је чињеница да њихово решење не можемо записати у облику неке математичке формуле или низа математичких формула од којих би се онда лако дошло до програма који даје решење тог проблема.

Обично се у том случају дати проблем (задатак) дели на једноставније проблеме (задатке) који се лакше решавају. Решење полазног проблема се у том случају добија комбиновањем решења новодобијених (једноставнијих) проблема. То комбиновање подразумева да решење сваког од једноставнијих проблема зависи од решења претходно решених једноставнијих проблема, или да решења једноставнијих проблема не смеју бити међусобно у контрадикцији.

Најбоље ће бити да то илуструјемо на једном примеру. Нека је потребно да напишемо програм који на шаховској табли дименизија 8×8 треба да постави осам краљица тако да се узајамно не нападају. Јасно у сваку од 8 колона (нумеришимо их са 0 до 7 слева надесно) шаховске табле треба поставити по једну краљицу. Свака колона има по 8 поља (нумеришимо их са 0 до 7 одозго надоле). Све скупа можемо на 8^8 начина ставити 8 краљица у 8 колона. Од тога велики број могућности отпадне због тога што у две разне колоне не могу стајати краљице на истом пољу. Стога се могући распореди своде на варијације без понављања дужине 8 од 8 елемената којих има 8! (осам факторијел). То је значајно смањење у односу на претходну бројку. Одавде се лако стиже до првог решења. Треба генерисати све варијације без понављања класе 8 од 8 елемената а након тога изабрати оне код којих се никоје две краљице не нападају зато што се налазе на истој дијагонали. Али и то решење је неефикасно. Варијација је укупно $8! = 40320$. Ако би генерисање једне варијације и провера да ли се краљице нападају трајала 1 секунду за проверу свих варијација би требало нешто више од 11 сати.

Како онда скратити поступак? Па нека нам је једноставнији задатак који треба решити да поставимо једну краљицу у једну колону. Тако полазни проблем сводимо на то да у нулту колону поставимо краљицу, а потом да у преосталих седам колона поставимо седам краљица тако да се међусобно не нападају, али исто тако да се не нападају са краљицом у нултој колони. Но постављање седам преосталих краљица, исто тако, можемо извести тако што ћемо поставити краљицу у најлевљу од седам преосталих колона тако да се не напада са претходно

постављеном краљицом и на постављање шест краљица у шест преосталих колона, опет уз ограничење да се не нападају међусобно и са претходно постављеним краљицама.

Тако долазимо до основне процедуре која има један аргумент (цео број, рецимо i), а која уз претпоставку да је у i колона постављена по једна краљица од којих се никоје две не туку треба да постави још $8-i$ краљица у преосталих $8-i$ колона тако да се међусобно не нападају и не нападају са већ постављених i краљица. Јасно, цео задатак се решава позивањем те процедуре за вредност параметра i једнаку 0 (нула). Процедура записана на kvazi-Pascal-у би изгледала овако:

```
procedure Postavi (i:  integer);
begin
  postavi kraljicu u sledecu kolonu tako da se ne napada sa prethodno postavljenim;
  if i = 7 then begin odstampaj resenje i vrati se end
  else Postavi (i+1)
end.
```

На први поглед ово би требало да доведе до решења. Међутим, ако пробате да на папиру отпратите извршавање ове процедуре за $i = 0$ видећете да за неку вредност параметра и нећете моћи нигде да поставите краљицу у наредној непопуњеној колони. То се може и видети на приложеној слици. Другим речима наша процедура неће довести до решења. Где је проблем? Па у нултој колони смо краљицу могли поставити на једно од 8 места, а у зависности од тога где смо је поставили, у првој ће краљица моћи да буде на неком од 5 или 6 места. Наравно то важи и за следеће колоне, али да не анализирамо даље. Горе записана процедура је налазила само једно место на коме би краљица могла стајати у тој колони и прелазила на следећу колону. Комплетно решење би требало да искористи и преостала места у колонама.

Сл. 1

Тај део наше процедуре који би требало да то обезбеди чини поступак који се зове *backtracking*. Идеја је да се оног тренутка када у некој колони немамо где да ставимо краљицу вратимо у претходну колону и у њој нађемо следећу позицију на коју можемо да је поставимо, па да са новом позицијом пробамо да поставимо краљице у остатку табле. Другим речима покушавамо да остатак проблема решимо варирајући (мењајући) решење првог дела проблема. Тако долазимо до мало модификоване варијанте решења које отприлике изгледа овако:

```
Function Postavi (i:  integer):  tipusp;
var uspesnost:  tipusp;
begin
  uspesnost := neuspeh;
  postavi kraljicu na nulto polje u i-toj koloni;
  repeat
    if zadnja postavljena kraljica se ne napada sa prethodno postavljenim then
```

```
      if i = 7 then begin odstampaj resenje; uspesnost := uspeh end
      else uspesnost := Postavi (i+1);
    if (uspesnost = neuspeh) then pomeri kraljicu na sledece polje u i-toj koloni;
  repeat nemamo vise gde staviti kraljicu u i-toj koloni ili (uspesnost = uspeh);
  Postavi := uspesnost
end.
```

уз претпоставку да имамо дефинисан тип:

```
type tipusp = (neuspeh, uspeh);
```

Као што се види дошло је до мале модификације у смислу да је сада основна програмска целина реализована као функција. То је учињено зато што се мора имати информација о успешности извршења постављеног задатка. У овом случају најважнији је корак који се налази иза позива функције `Postavi`. Уколико, функција `Postavi` није успела да попуни преостале колоне тада се краљица склања са места на коме се тренутно налази и ставља на неко од преосталих места, наравно ако је то могуће. Поступак се понавља све док не попунимо остатак табле или не испробамо све варијанте за ту колону. Јасно, то је поступак који се понавља за сваку колону појединачно.

Коначно стижемо и до практичне имплементације овог алгоритма у неком вишем програмском језику. У имплементацији је, осим на самом поступку, акценат и на томе како ћемо приказати информације о положају фигура на шаховској табли. Варијанта која се одмах намеће је да се табла презентира као дводимензионални низ логичког или целобројног типа у коме би са једном вредношћу записивали да се на одговарајућем пољу налази фугура, а неком другом да се не налази. Оваква реализација има неколико недостатака. Први је свакако да је тиме потрошено превише меморије. Други је много битнији, а то је чињеница да је на тај начин нешто теже реализовати проверу да ли на неко поље можемо ставити краљицу (тј. да ли је поље „нападнуто“ неком претходно постављеном краљицом или не). Најбоље је да сами пробате да напишете процедуру која би проверавала да ли је поље „нападнуто“. Реализација коју ћемо ми направити изгледа овако:

(i) Таблу ћемо приказати помоћу низа који има онолико елемената колико је и димензија табле, целобројног типа код кога сваки елемент носи податак о томе где се налази краљица у одговарајућој вертикали (колони).

(ii) За проверу да ли је неко поље нападнуто или не уводимо три низа
  (a) `vrsta` који има елемената колико је димензија табле а који носи податак о томе да ли у одговарајућој врсти већ стоји краљица,
  (b) `dij45` који има елемената колико има и `/-dijagonala` и који носи податак о томе да ли на одговарајућој `/-dijagonali` стоји краљица. Појединe `/-dijagonale` карактерише чињеница да поља која леже на њој имају исти збир броја колоне и броја врсте, ако усвојимо претходно наведену нумерацију. На тај начин добијамо дијагонале нумерисане бројевима 0 до $2 * n - 2$ где је n димензија табле (код нас 8),
  (c) `dij135` који има елемената колико и `/-dijagonala`, а који носи податак да ли на одговарајуцој дијагонали стоји краљица. Код ових дијагонала је разлика броја врсте и броја колоне константна за сва поља те та-

ко имамо дијагонале нумерисане бројевима од -7 (пролази кроз доњи леви) до 7 (пролази кроз горњи десни угао).

Тако стижемо до програма:

```
program kraljice (tty);
var i:  integer;
  pozicija:  array [0..7] of integer;
  vrsta:  array [0..7] of boolean;
  dij45:  array [0..14] of boolean;
  dij135:  array [-7..7] of boolean;
function postavi(i:  integer):  boolean;
var f:  boolean; j, k:  integer;
begin
  f := false; j := 0;
  repeat
    (* provera da li se na j-to polje u i-toj koloni moze staviti kraljica *)
    if vrsta[j] and dij45[i+j] and dij135[i-j] then
    begin
      (* ako moze, staviti na to polje, sto znaci da su vrsta i dijagonale
        koje prolaze kroz to polje zauzete *)
      pozicija[i] := j;
      vrsta[j] := false;
      dij45[i+j] := false;
      dij135[i-j] := false;
      (* ako je to osma kraljica, stampati resenje, ako nije osma nastaviti
        popunjavanje ostatka table *)
      if i = 7 then f := true
      else
      begin
        f := postavi (i+1);
      (* ako se ostatak table ne moze popuniti skinuti malopre postavljenu kraljicu.
        To znaci da su vrsta i dijagonale koje su prolazile kroz to polje sada
        slobodne *)
        if not f then
        begin
          vrsta[j] := true;
          dij45[i+j] := true;
          dij135[i-j] := true;
        end
      end
    end;
    j := j+1
  until (j > 7) or f;
  postavi := f
end;

procedure stampaj;
var i, j:  integer;
begin
  for i := 0 to 7 do
  begin
    for j := 0 to 7 do
      if pozicija [j] = i then write (' 1') else write (' 0'); writeln;
  end
end;
begin
  for i:= -7 to -1 do dij135[i] := true;
  for i:= 8 to 14 do dij45[i] := true;
  for i:= 0 to 7 do
  begin
```

```
    vrsta[i] := true;
    dij45[i] := true;
    dij135[i] := true
  end;
  if postavi (0) then stampaj
  else
    write ('Ne postoji resenje!');
end.
```

Згодна страна изложеног решења лежи у чињеници да уз мале модификације може послужити за налажење свих решења. Промена се састоји само у томе да се промени петља у којој се налази следеће решење мањег проблема и рекурзивно позива процедура за остатак. Промена се састоји у промени критеријума за излазак. Сада је једини критеријум за излазак да смо потрошили сва решења за издвојени мали проблем. Програм који то ради изгледао би овако:

```
program kraljice (tty);
var n:  integer;
  pozicija:  array [0..7] of integer;
  vrsta:  array [0..7] of boolean;
  dij45:  array [0..14] of boolean;
  dij135:  array [-7..7] of boolean;
procedure stampaj;
var i, j:  integer;
begin
  for i := 0 to 7 do
  begin
    for j := 0 to 7 do
      if pozicija [j] = i then write (' 1') else write (' 0'); writeln
  end
end;
procedure postavi(i:  integer);
(* vise ne mora biti funkcija, jer nam nije bitno da li je vec nadjeno neko
  resenje *)
var f:  boolean; j, k:  integer;
begin
  f := false; j := 0;
  repeat
    if vrsta[j] and dij45[i+j] and dij135[i-j] then
    begin
      pozicija[i] := j;
      vrsta[j] := false;
      dij45[i+j] := false;
      dij135[i-j] := false;
      (* n sluzi da prebroji ukupan broj *)
      if i=7 then begin stampaj; n := n+1 end
      else postavi (i+1);
      vrsta[j] := true;
      dij45[i+j] := true;
      dij135[i-j] := true;
    end;
    j := j+1
  until (j > 7);
  (* promenjen je uslov za izlazak i vise se ne proverava
    da li je vec nadjeno ili ne neko resenje *)
end;
begin
  for n:= -7 to -1 do dij135[n] := true;
  for n:= 8 to 14 do dij45[n] := true;
```

```
  for n:= 0 to 7 do
  begin
    vrsta[n] := true;
    dij45[n] := true;
    dij135[n] := true
  end;
  n := 0;
  postavi (0);
  write ('Ukupno ima ', n, ' resenja!');
end.
```

Покушајмо да сумирамо. *Backtracking* је поступак за налажење решења неког проблема којим се тај проблем разбија на низ малих проблема (подзадатака). Број подзадатака на које је разбијен полазни задатак зовимо у даљем тексту димензија задатка (проблема). У том случају се полазни задатак може третирати као целина која се састоји од првог подзадатка и задатка истог типа као и полазни, али за један мање димензије. Решење целог задатка се онда своди на то да се реши први подзадатак, а да се потом поново решава полазни проблем али за један умањене димензије. Међутим решење другог дела (полазни задатак за један умањене димензије) зависи од решења првог дела, тако да у неповољнијим случајевима уопште неће моћи да се реши за изабрано решење првог дела. У том случају се поступак понавља за друга решења првог дела све док се не исцрпе сва решења за први део или док се не успе решити и други део. Са овим делом добијамо комплетан поступак који се у литератури обично назива backtracking.

Рекурзивно решење је као и у већини случајева за програмере најјасније (најприродније). Међутим, као и у већини случајева може се направити и нерекурзивно. Ипак, то решење бисмо оставили за неку другу прилику.

Нешто што је заједничко за све програме написане коришћењем backtrackinga било би следеће: решење полазног проблема је неки низ X при чему за сваки елемент x_k тог низа постоји скуп A_k из кога бирамо кандидата за k-ти члан решења. (У конкретном примеру низ има 8 елемената а за скупове кандидата важи $A_0 = A_1 = \cdots = A_7 = \{0, 1, \ldots, 7\}$). Осим тога постоји нека довољно једноставна функција која проверава да ли се произвољно делимично решење $(x_1, \ldots, x_i)$ може проширити ка потпуном решењу. (У конкретном случају функција проверава да ли се краљице постављене на $(x_0, \ldots, x_i)$ позиције редом у колонама $(0, \ldots, i)$ међусобно нападају, а то се своди на то да се провери да ли се краљица постављена у i-тој колони на позицију x_i напада са неком од претходно постављених краљица).

Интуитивно, backtacking подразумева обилазак једног дрвета — дрвета решења. У корену тог дрвета налази се поставка полазног задатка. Гране које полазе из корена су могућа решења првог подзадатка издвојеног из полазног проблема, а свака грана се завршава чвором у коме се налази поставка задатка истог типа као и полазни задатак, али за један умањене димензије. Максимална дубина дрвета је онда једнака димензији полазног задатка, зато што се са сваким следећим нивоом за један смањује димензија задатка (у чворовима на дубини једнакој димензији полазног задатка је поставка задатка димензије нула, тј. ти чворови су листови дрвета). Некад се наравно може десити да од неког чвора у коме је поставка проблема чија је димензија већа од нуле не води ниједна грана.

То је у случајевима када се не може решити први подзадатак датог задатка и у том случају се преко тог чвора не може доћи до решења. Backtracking је поступак којим се обилази дрво са циљем да се нађу чворови који се налазе на дубини једнакој димензији полазног задатка. Пут који води од корена до тих чворова (листова) представља решење полазног задатка. Међутим, битно је да се у овом случају то обилажење обавља тако да се издваја једна по једна комплетна грана слева надесно. При том под гранама у овом случају подразумевамо пут од корена до листа, а издвајање грана слева надесно подразумева налажење свих листова дрвета слева надесно, а потом издвајање само оних листова који су довели до комплетног решења. Део дрвета до првог решења за проблем 8 краљица дат је на слици 2.

Сл. 2

На крају да наведемо и неколико задатака који се могу на сличан начин решити:

1. Нека је дата шаховска табла дименизја $m \times n$ и скакач који се налази на пољу (i, j). Написати програм који проверава да ли скакач који се креће по правилима шаховске игре може посетити свако поље тачно једанпут, и ако је могуће исписује „маршруту“.

2. Нека је дата табла димензија $2m \times 2n$. Написати програм који проверава на колико различитих начина је могуће покрити таблу фигурама датог облика, при чему је могућа ротација фигуре за 90, 180 и 270 степени.

3. Нека је дат скуп земаља и подаци о суседности. Написати програм који проверава да ли је могуће обојити дате земље са 4 различите боје тако да никоје две суседне не буду обојене истом бојом и ако је могуће приказати бојење.