Megoldáskereső algoritmusok programozása Racket nyelven

Programming search algorithms in Racket

BALÁZS Patrícia, dr. KÓSA Márk, dr. PÁNOVICS János

Debreceni Egyetem Informatikai Kar
H–4028 Debrecen, Kassai út 26.
telefon: +36/52/512900, honlap: http://www.inf.unideb.hu/
e-mail: balazs.pannaa@gmail.com, kosa.mark@inf.unideb.hu, panovics.janos@inf.unideb.hu

Abstract

Functional programming languages have been closely related to the field of artificial intelligence since it was born. In recent years, we have presented the most important search algorithms using object-oriented approach at the Faculty of Informatics, University of Debrecen. With the modernization of the curriculum of the Computer Science BSc program, a new subject called High-Level Programming Languages 3 was created, which is intended to present programming languages based on the functional paradigm. In this paper, we show how the algorithms introduced in the object-oriented world can be implemented in a purely functional environment, namely in Racket programming language.

Kivonat

A mesterséges intelligencia tudományterületéhez már megszületésétől kezdve szorosan kapcsolódtak a funkcionális programozási nyelvek. A Debreceni Egyetem Informatikai Karán az elmúlt években objektumorientált megközelítésben ismertettük a legfontosabb megoldáskereső algoritmusokat. Amióta megtörtént a programtervező informatikus BSc szak mintatantervének korszerűsítése, megjelent benne a Magas szintű programozási nyelvek 3 tárgy, amely kifejezetten a funkcionális paradigmán alapuló programozási nyelveket mutatja be. Cikkünkben azt mutatjuk be, hogy hogyan lehet az objektumorientált világban már megismert algoritmusokat tisztán funkcionális környezetben, jelesül Racket nyelven is implementálni.

Kulcsszavak: Megoldáskereső algoritmusok, funkcionális programozás, Racket programozási nyelv.

1. Bevezetés

Van-e létjogosultsága a funkcionális programozási nyelveknek az oktatásban? A válasz: igen. Tekintsük át röviden, hogy milyen előnyökkel járhat, ha az imperatív helyett funkcionális megközelítéssel oldjuk meg a problémákat [1]:

- *Elegáns*: Sokak szerint a funkcionális kód szebb, mint az imperatív. Ez persze közvetlenül nem váltható pénzre, nem ez tehát a legfontosabb szempont.
- *Produktív*: A szoftverfejlesztés folyamata gyorsabb és kevesebb hibalehetőséget tartalmaz.
- *Tömör*: Ha összehasonlítunk egy összetettebb probléma megoldására szolgáló imperatív és funkcionális kódot, szinte biztosan az tapasztaljuk, hogy az utóbbi lényegesen rövidebb.
- *Kifejező*: A funkcionális kód a lényegre koncentrál, csak a probléma megoldásának leírását tartalmazza, nincsenek benne olyan kódrészletek, amelyek az imperatív kódban csak ahhoz kellenek, hogy a program egyáltalán működjön. Ennek az egyik oka, hogy míg az imperatív nyelvekben egy algoritmus kódolása során legfeljebb új primitíveket (elemi lépéseket) hozhatunk létre, addig a funkcionális nyelvekben új vezérlési szerkezeteket is, azaz olyan eszközöket, amelyekkel a primitívek összekapcsolhatók (az imperatív nyelvekben ezek a szekvencia, szelekció és iteráció).

Ami az oktatásban betöltött szerepét illeti, a problémák funkcionális megközelítése egyfajta sajátos gondolkodásmódot követel meg a hallgatóktól. Ennek a gyökerei már az általános iskolában megjelennek, hiszen a függvény fogalmával már ott találkoznak. Az itt elsajátított megoldási módszerek könnyen beépíthetők a manapság elterjedt imperatív és objektumorientált programozási nyelvekbe is. Ezek a nyelvek (pl. Java vagy C#) az elmúlt pár évben jelentősen bővültek funkcionális programozási eszközökkel.

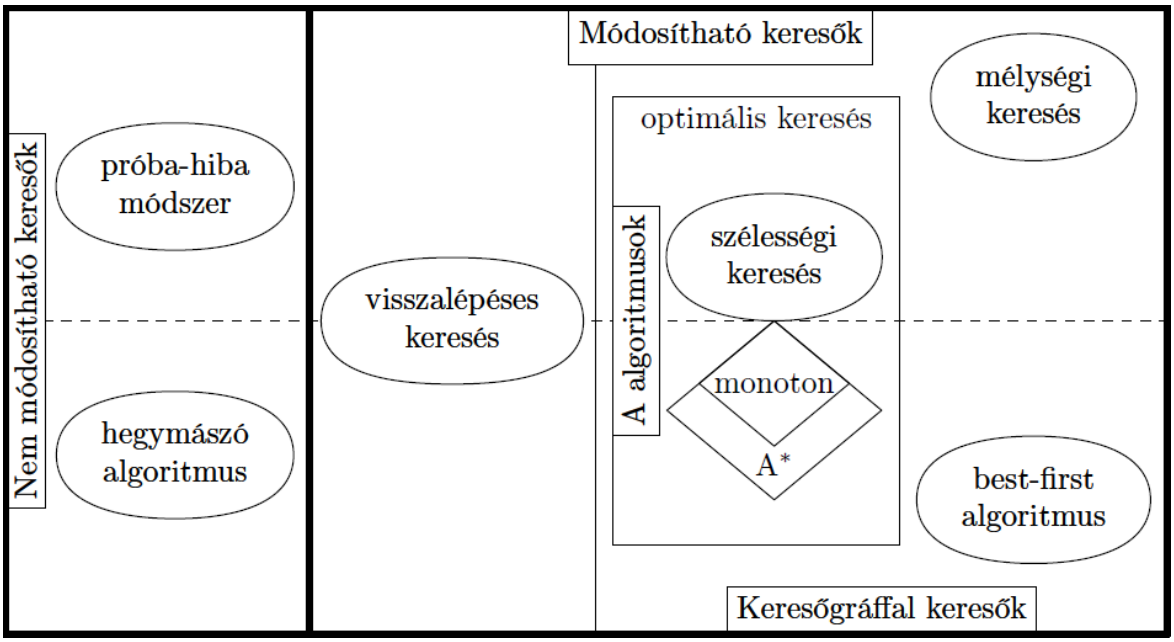
Cikkünkben ismertetjük a mesterséges intelligenciában használt néhány megoldáskereső algoritmus egy konkrét funkcionális nyelven megírt megvalósítását. A nyelv, amit használni fogunk, a Racket [2]. A Racket egy általános célú, multiparadigmás programozási nyelv, amely a LISP Scheme-dialektusán alapul. Eredetileg arra tervezték, hogy új programozási nyelveket lehessen definiálni a segítségével, de számos más területen is felhasználható. A hozzá tartozó függvénykönyvtár tartalmaz eszközöket többek között rendszerprogramozáshoz, hálózati programozáshoz, webprogramozáshoz, logikai programozáshoz és grafikus felhasználó felületek programozásához is. A Racketet ezen lehetőségei miatt széles körben használják a számítástudományok oktatásában és a kutatásban. Az itt bemutatott algoritmusok megvalósításához csak a nyelv alapeszközeit fogjuk használni.

2. Megoldáskereső algoritmusok

Az 1. ábrán néhány széles körben használt megoldáskereső algoritmus és ezek kapcsolata látható. Az ábra horizontálisan és vertikálisan is két nagy részre van tagolva.

A bal oldalon láthatók a nem módosítható keresők, míg a jobb oldalon a módosíthatók. A nem módosítható keresők jellemzője, hogy az adatbázisuk mindössze egyetlen csomópontot tárol, ezért nincs lehetőség egy-egy művelet hatásának a visszavonására. A módosítható keresők adatbázisa ezzel szemben a reprezentációs gráf egy nagyobb részletét tárolja (egy utat vagy egy részfát), így a kereső algoritmus a megoldáshoz vezető útvonal keresése során felülbírálhatja egy korábbi döntését.

A szaggatott vonal szintén két részre osztja az ábrát. Az ábra felső részében láthatók a nem informált keresők, alul pedig olyan algoritmusokat találunk, amelyek valamilyen külső, a reprezentációból nem következő információt is felhasználnak a megoldás keresése során.



1. ábra

Néhány széles körben használt megoldáskereső algoritmus és ezek kapcsolata.

Az alábbiakban ismertetjük a Hanoi tornyai problémát megoldó visszalépéses, szélességi és mélységi keresők Racket-implementációit.

2.1. A Hanoi tornyai probléma reprezentációja Racketben

A cikkünk további részében a jól ismert Hanoi tornyai probléma háromkorongos változatán keresztül mutatjuk be az egyes algoritmusokat. A probléma állapotait háromelemű listákkal reprezentáljuk. Az egyes elemek azoknak a rudaknak a szimbólumait írják le, amelyeken rendre a legkisebb, a középső, illetve a legnagyobb korong található. A rudakat az `'A`, `'B` és `'C` szimbólumokkal jelöljük. Így például a `'(B A C)` lista azt írja le, hogy a legkisebb korong a B rúdon, a középső az A, a legnagyobb pedig a C rúdon található.

A problémánk kezdőállapota az

```
'(A A A)
```

listával írható le.

Célunk az, hogy az összes korongot sikerüljön átpakolni a C rúdra. Ennek megfelelően ahhoz, hogy egy állapotról eldöntsük, hogy célállapot-e, a következő függvényt használhatjuk:

```
(λ (s)
  (equal? s '(C C C)))
```

Az állapotváltásokat leíró operátorokat szintén egy-egy háromelemű listával reprezentáljuk: az első elem azt mondja meg, hogy melyik rúdról mozgatjuk a legfelső (legkisebb) korongot (a továbbiakban legyen ez a *honnan* rúd), a második elem azt, hogy melyik rúdra szeretnénk őt mozgatni (*hova* rúd), a harmadik pedig azt, hogy melyik korongot szeretnénk mozgatni (*mit* korong). Valójában persze az első és a harmadik elem közül az egyik felesleges, de így egyszerűbb lesz a kódot implementálni. Az operátorok listája tehát a következő lesz:

```
'((A B 1) (A B 2) (A B 3) (A C 1) (A C 2) (A C 3)
  (B A 1) (B A 2) (B A 3) (B C 1) (B C 2) (B C 3)
  (C A 1) (C A 2) (C A 3) (C B 1) (C B 2) (C B 3))
```

Mikor alkalmazhatunk egy ilyen operátort? Akkor, ha a *honnan* rúd nem üres, a *honnan* rúd legfelső (legkisebb) korongja a *mit* korong, és a *hova* rúd vagy üres, vagy a legkisebb korongja nagyobb a *mit* korongnál. Ezt az alkalmazási előfeltételt az alábbi függvénnyel tudjuk implementálni:

```
(λ (s op)
  (define (from s n)
    (cond
      [(empty? s) #f]
      [(equal? (car op) (car s)) n]
      [else (from (cdr s) (add1 n))]))
  (define (to s n)
    (cond
      [(empty? s) #f]
      [(equal? (cadr op) (car s)) n]
      [else (to (cdr s) (add1 n))]))
  (and
    (not(not (from s 0)))
    (= (from s 1) (caddr op))
    (or (not(to s 0)) (> (to s 1) (caddr op)))))
```

A lambda függvényen belül lokálisan definiált `from` és `to` függvények azt mondják meg, hogy mi a *honnan*, illetve a *hova* rúd szimbólumának első előfordulása az állapotot leíró listában. Ha az adott szimbólum nem fordulna elő, akkor a függvények `#f` (hamis) értéket adnak eredményül.

Mi lesz az operátorok hatása, ha alkalmazzuk őket egy konkrét állapotra? Az állapotot leíró elemhármasban fog megváltozni egy elem, mégpedig a *mit* koronghoz tartozó szimbólum fog módosulni a *hova* szimbólumra. Ezt láthatjuk az alábbi függvény kódjában:

```
(λ (s op)
  (let helper ((s s) (res '()) (op op) (change? #f))
    (cond
      [(empty? s) (reverse res)]
```

```
        [change? (helper (cdr s) (cons (car s) res) op change?)]
        [(equal? (car s) (car op))
         (helper (cdr s) (cons (cadr op) res) op #t)]
        [else (helper (cdr s) (cons (car s) res) op change?)])))
```

A lambda függvényben definiált `helper` balról jobbra végighalad az állapotot leíró listán. Amikor elérkezik a *honnan* rúd legkisebb korongjához, akkor kicseréli annak betűjelét a *hova* szimbólumra.

A fenti lista- és függvénydefiníciókkal megadtuk a Hanoi tornyai probléma egy lehetséges reprezentációját Racket nyelven.

2.2. Visszalépéses megoldáskeresés

A visszalépéses megoldáskeresők a módosítható megoldáskeresők közé tartoznak. Alapváltozataik nem informáltak, a külső forrásból származó információ az operátorok kipróbálásának és alkalmazásának a sorrendjére vonatkozhat.

```
(define (backtrack-search
         initial-state
         operators
         apply-operator?
         apply-operator
         goal?)
  (let helper
    ((lst (list (list initial-state
                      null
                      (filter (curry apply-operator? initial-state)
                              operators)))))
    (cond
      [(empty? lst) lst]
      [(goal? (caar lst)) lst]
      [(empty? (caddar lst)) (helper (cdr lst))]
      [(member (apply-operator (caar lst) (car(caddar lst)))
               lst
               (λ (a b) (equal? a (car b))))
       (helper (cons (list (caar lst) (cadar lst) (cdr (caddar lst)))
                     (cdr lst)))]
      [else
       (helper
        (cons
         (list (apply-operator (caar lst) (car(caddar lst)))
               (car(caddar lst))
               (filter
                (curry apply-operator?
                       (apply-operator (caar lst) (car(caddar lst))))
                operators))
         (cons (list (caar lst) (cadar lst) (cdr (caddar lst)))
               (cdr lst))))])))
```

A visszalépéses keresés a nyilvántartott csúcsok tárolásához egy listát kezel. A lista elemei háromelemű listák lesznek. Ezek első komponensei egy-egy állapotot írnak le (például az `'(A A A)` kezdőállapotot vagy egy hasonlóan kinéző tetszőleges másik állapotot). A második komponens az jelzi, hogy milyen operátor alkalmazásával állt elő az első komponens. A kezdőállapot esetében ez az üres lista lesz, ami azt szimbolizálja, hogy a kezdőállapot eleve adott volt, semmilyen másik állapotból nem kellett őt származtatni. Minden más állapot esetében ez a komponens egy konkrét operátort fog jelölni egy háromelemű lista formájában. Ilyen operátort ír le például az `'(A B 1)` vagy a `'(C B 2)` háromelemű lista. A csúcsokat leíró lista harmadik komponense azoknak az operátoroknak a listája lesz, amelyek az adott állapotra alkalmazhatók, de még nem alkalmaztuk rá őket. Kezdetben tehát, a visszalépéses keresés inicializálásakor, az adatbázis az alábbi egyelemű lista lesz, amelyből kiolvasható, hogy a kezdőállapotra két operátort is lehet majd alkalmazni:

```
(((A A A) () ((A B 1) (A C 1))))
```

A kereső vezérlését esetszétválasztással oldjuk meg, melyet a `helper` nevesített `let` szerkezetben implementálunk:

- Ha üres lenne az útvonalunk, azaz már a kezdőállapotot tartalmazó csúcsból is visszaléptünk, akkor nincs mit csinálni, visszaadjuk az üres listát, így jelezve, hogy nem sikerült megoldást találnunk.
- Ha a lista első elemében tárolt állapot célállapot lenne, akkor is készen vagyunk, de ebben az esetben a problémának a megoldása is a kezünkben (azaz inkább a listánkban) van, így ezt adjuk vissza eredményként.
- Ha a fentiek egyike sem következett volna be, akkor további vizsgálatokat végzünk el. Ha már nem maradt ki nem próbált operátor az útvonal utolsó csúcsában (amit a lista feje tartalmaz), akkor visszalépünk: elfelejtjük a lista eddigi első elemét, és a lista farkával dolgozunk tovább.
- Ha maradt volna még ki nem próbált operátor, akkor kiválasztjuk az elsőt, és alkalmazzuk azt. A létrejövő új állapot mellé odatesszük az operátort, amivel ez az új állapot előállt, valamint mindazokat az operátorokat, amelyek erre az új állapotra alkalmazhatók. Az így létrejött háromelemű listát az útvonal elejére illesztjük, és kezdjük az egész esetszétválasztást elölről.

2.3. Szélességi keresés

A szélességi kereső egy nem informált módosítható megoldáskereső. A megoldás lépésszámára vonatkozóan optimális megoldás előállítására alkalmas.

```
(define (breadth-first-search
         initialState
         operators
         apply-operator?
         apply-operator
         goal?)
  (let helper
    ((open (list (list initialState '() 0 '()))) (closed '()))
    (cond
      [(empty? open) open]
      [(goal? (caar open))
       (let loop
         ((item (car open))
          (result (list (list (second (car open))
                              (caar open)
                              (third (car open))))))
         (cond
           [(empty? (fourth item)) result]
           [else (loop (fourth item)
                       (cons (list (second (fourth item))
                                   (car (fourth item))
                                   (third (fourth item)))
                             result))]))]
      [else
       (let loop
         ((s (caar open))
          (op operators)
          (depth (third (car open)))
          (open (cdr open))
          (closed (cons (car open) closed)))
         (cond
           [(empty? op) (helper open closed)]
           [(and (apply-operator? s (car op))
                 (not (member
                       (apply-operator s (car op))
                       open
                       (λ (a b) (equal? a (car b)))))
                 (not (member (apply-operator s (car op))
                              closed
```

```
                        (λ (a b) (equal? a (car b))))))
          (loop s (cdr op) depth
                (append open (list (list (apply-operator s (car op))
                                         (car op)
                                         (add1 depth)
                                         (car closed))))
                closed)]
          [else (loop s (cdr op) depth open closed)]))]))))
```

A szélességi kereső a nyilvántartott csúcsok tárolásához két listát használ: a zárt csúcsokét és a nyíltakét. Mindkét listának az elemei négyelemű listák lesznek. A négy komponens közül az első egy állapotot ír le. A második komponens azt jelzi, hogy milyen operátor alkalmazásával állt elő az első komponens. A harmadik komponens a startcsúcstól megtett lépések száma lesz. Ennek a nyilvántartására valójában nincsen szükség, csak nyomkövetési célokból szerepeltetjük a listában. A negyedik komponens a startcsúcsból az adott csúcsba vezető teljes útvonalat reprezentálja. Ez a komponens egy rekurzív adatszerkezet, amely egy négy komponenst tartalmazó lista. Ennek első három eleme az előbb tárgyalt három komponens, a negyedik pedig az a lista, amely a szülő csúcsot írja le ugyanilyen módon. A startcsúcsban ez a negyedik komponens egy üres lista. Ezzel a konstrukcióval, amikor a keresőalgoritmusunk célállapotot talál, rögtön a rendelkezésünkre áll egy, a startcsúcsból valamely terminális csúcsba vezető megoldási útvonal is.

A kereső vezérlését ezúttal is esetszétválasztással oldjuk meg:

- Ha üres a nyílt csúcsok listája, azaz nincsen több kiterjeszthető csúcs, akkor a feladatnak nincs megoldása.
- Ha a nyílt csúcsok listájában az első elem (amelynek egyébként az összes többi nyílt csúcshoz képest kisebb vagy egyenlő a mélységi száma) célállapotot tartalmaz, akkor a `loop` nevesített `let` szerkezettel előállítjuk a megoldást leíró listát.
- Egyébként kipróbáljuk az összes operátort a nyílt csúcsok listájának az első elemére. Ha alkalmazható egy operátor, és az ő alkalmazásával olyan állapotot állítanánk elő, amely sem a nyílt, sem a zárt csúcsok listájában nem szerepel, akkor előállítunk egy új csúcsot a korábban részletezett négy komponenssel, és hozzáfűzzük őt a nyílt csúcsok listájának a végéhez, majd folytatjuk a következő operátor vizsgálatával. Ha már minden operátort kipróbáltunk volna, akkor a `helper` nevesített `let` szerkezet meghívásával ismét visszatérünk a keresőt vezérlő esetszétválasztáshoz.

2.4. Mélységi keresés

A mélységi kereső is egy nem informált módosítható megoldáskereső. A vezérlési stratégiája nagyon hasonló a szélességi keresőéhez. Az eltérés annyi, hogy a mélységi kereső mindig a legnagyobb mélységi számú nyílt csúcsok közül választ egyet kiterjesztésre, amennyiben még nem talált célállapotot. A gyakorlatban ezt úgy szokták megvalósítani, hogy a szélességi keresésnél a nyílt csúcsok tárolásához használt sor adatszerkezetet verem adatszerkezetre cserélik. Alábbi példánkban mi is így járunk el: a nyílt csúcsok listáját nem a végén, hanem az elején bővítjük az újonnan előállított csúcsokkal. Emellett azonban az alábbiakban leírtak szerint a csúcsok reprezentációját is megváltoztattuk.

```
(define (dfs init-state operators precond? apply-op goal?)
  (let helper ((open (list (list init-state '() 0 '()))) (closed '()))
    (cond
      [(empty? open) '()]
      [(goal? (caar open))
       (let solution ((needle (car open))
                      (haystack closed)
                      (result (list (list (third (car open))
                                          (fourth (car open))
                                          (caar open)))))
         (cond
           [(empty? (cadr needle))
            (cons (list (third needle) (fourth needle) (first needle))
                  result)]
           [else (solution (car (memf (lambda (x)
```

```
                                        (equal? (car x) (cadr needle)))
                                  closed))
                            haystack
                            (cons (list (third needle)
                                        (fourth needle)
                                        (car needle))
                                  result))]))]
        [else
         (helper (append
                  (filter (lambda (x)
                            (and (not (member (car x) (map car open)))
                                 (not (member (car x) (map car closed)))))
                          (map (lambda (op)
                                 (list (apply-op (caar open) op)
                                       (caar open)
                                       (add1 (third (car open)))
                                       op))
                               (filter (curry precond? (caar open))
                                       operators)))
                  (cdr open))
                 (cons (car open) closed))])))
```

A mélységi keresőnk a nyilvántartott csúcsok tárolásához – a szélességi keresőhöz hasonlóan – ugyancsak két listát használ: a zárt csúcsokét és a nyíltakét. Mindkét listának az elemei négyelemű listák lesznek. A négy komponens közül az első három megegyezik a szélességi keresőnél ismertetettekkel. A negyedik komponens ezúttal a szülő csúcsban tárolt állapot lesz. Így ugyan kicsit nehézkesebb lesz a megoldás előállítása, hiszen az adott csúcshoz vezető útvonal elemeit külön-külön, egyesével kell majd előkeresni a zárt csúcsok listájából, viszont így egyszerűbbnek tűnik az ábrázolás.

A kereső vezérlését ezúttal is esetszétválasztással oldjuk meg:

- Ha üres a nyílt csúcsok listája, azaz nincsen több kiterjeszthető csúcs, akkor a feladatnak nincs megoldása.
- Ha a nyílt csúcsok listájában az első elem (amelynek egyébként az összes többi nyílt csúcshoz képest nagyobb vagy egyenlő a mélységi száma) célállapotot tartalmaz, akkor a `solution` nevesített `let` szerkezettel előállítjuk a megoldást leíró listát.
- Egyébként kipróbáljuk az összes operátort a nyílt csúcsok listájának az első elemére. Miután összegyűjtjük az alkalmazható operátorok alkalmazásával előállított összes olyan állapotot, amely sem a nyílt, sem a zárt csúcsok listájában nem szerepel, és előállítjuk az őket tartalmazó csúcsok listáját, ehhez a listához fűzzük hozza a nyílt csúcsok eddigi listáját.

3. ÖSSZEFOGLALÁS

Cikkünkben a problémamegoldás egy lehetséges módjaként funkcionális megközelítésben mutattuk be a mesterséges intelligencia területén közismert megoldáskereső algoritmusok Racket nyelvű implementációit. Bár a bemutatott kódok sok mindenben hasonlítanak egymásra, törekedtünk arra, hogy ne csak az elméletből ismert elvi eltérésekre helyezzünk hangsúlyt, hanem a nyelvi eszközök és lehetőségek engedte apró finomságokat is beépítsük a kódokba. Ezen törekvésünket a különböző keresők által használt adatszerkezetek megválasztásában, valamint azok kezelésében realizáltuk.

HIVATKOZÁSOK

[1] Tomas Petricek, Jon Skeet: Real-World Functional Programming, With examples in F# and C#, Manning, 2010.

[2] A Racket programozási nyelv, https://racket-lang.org.