Zalavári Márton

Deklaratív Programozás NHF

2018 ősz

Előkészítés

A megoldás alapötlete a következő: Választunk egy számot a megengedett határokon belül, azaz az 1 és Max között. Feltesszük, hogy a titkos kódunk ezzel a színnel kezdődik. Ezután módosítjuk a súgás listát, most már azzal az információval, hogy „tudjuk” mi az első szín. Ha ez megvolt, akkor rekurzívan újból „tippelünk” egy színt következőnek, és újból módosítjuk, a korábban már egyszerűsített a súgáslistát. És így tovább. Amennyiben az utolsó lépésre az a súgáslistánkon lévő összes súgás konzisztens lesz (tehát igazából a fekete és fehér pálcikák száma is egyaránt zérus), akkor találtunk egy megoldást. Ellenkező esetben visszalépünk. Tehát lényegében egy visszalépéses keresést kell implementálni.

Új adatstruktúra bevezetése

Természetesen van néhány nem triviális dolog, amit valahogyan meg kell oldani. Ezen kívül a keresési tér is nagyon nagyra nőhet, hiszen Max^L darab lehetséges kód lehet.

Először is a súgás-t újra definiálom. Nem csak a tippkódot, és az arra adott válaszokat tartalmazzon egy súgás, hanem még egy multihalmazt is, aminek első közelítésben az a jelentése, hogy a fehér vagy fekete pálcákkal jelölt bábuk melyik számhalmazból kerülhetnek ki.

Az talán elég világos, hogy kezdetben ez a halmaz az adott súgáshoz tartozó kód elemeiből áll. Az is érthető, hogy ennek szükségszerűen multihalmaznak kell lennie. Viszont ezen a ponton még ködös ennek a szükségessége.

Lényegében arról van szó, hogy amikor a keresési fában egyre mélyebbre megyünk, nem „feledkezhetünk meg” a korábbi elemekről. Nem lehet egy egyszerű rekurzív függvénnyel elintézni a dolgot, hogy mindig egyre csökkenő tippkódokhoz illő titkos kódot keresünk, mert ahhoz, hogy tudjuk, hogy az éppen „letett” bábura vonatkozik-e a vizsgált súgásban fehér pálcika, az attól függ, hogy a tippben van-e *bárhol* azonos színű bábu. Erre még önmagában használhatnánk az eredeti tippkódot is, amit megkaptunk a súgásban, és a tipp csökkentése helyett egy indexszel dolgozunk, de így, hogy kezdetben van egy másolatunk az eredetiről, azt nyugodtan szerkeszthetjük, és segítségével számon tarthatjuk például azt is, hogy egy adott színű bábuból mennyire vonatkozhat még fehér válaszpálca.

A továbbiakban a súgáslista elemeit, a súgásokat úgy fogom érteni, hogy egy tippkód, egy multihalmaz, és két szám, a fehér és fekete válaszok száma. Ez egy kicsit bővebb, (sőt kezdetben redundánsabb), mint az eredeti definíció.

Változók és elnevezéseik

A többi adatstruktúra elég egyszerű, az elnevezésekben pedig igyekeztem a feladat által is megadottakat használni.

A Code jelenti a lehetséges titkos kódot, a Tipp, illetve a Black és White jelöli egy konkrét súgáshoz tartozó tippelt kódot és válaszpálcikákat.

Továbbiakban a Tipp-hez tartozó multihalmazra TippZsak néven fogok hivatkozni, az első kis házi feladatnak ezzel emléket állítva, hiszen a Zsak is egy multihalmaz implementáció.

Egy Hint, azaz egy súgás pedig ezek után tartalmazni fog egy-egy Tipp, TippZsak, Black és White elemet. A HintList pedig súgások listáját jelöli.

Szokásosan a Head és Tail szuffixum jelöli a listák fejét és farkát.

Továbbá a New prefixumot a kimeneti változó jelzésére használtam, a bemeneti azonos névűtől való megkülönböztetésre.

Az eljárások specifikálása

mmind

A mmind számunkra most fő programként jelenik meg. Kis előfeldolgozást végez. Átalakítja a feladat által értelmezett súgás listát az általam definiáltra, és kiszámolja a titkos kód hosszát egy tippkód segítségével, hogy később bármikor rendelkezésre álljon. Ezután a kapott Max értéket, illetve a súgáslistát és a kiszámolt L hosszt paraméterként átadja a tényleges algoritmust végző eljárásnak, amit mmm-nek neveztem el. (Kreatívan a MyMasterMind rövidítésből.)

mmm

A backtracking algoritmust implementálja. Szisztematikusan „tippel” egy számot, ami a titkos kód következő bábuját reprezentálja, azaz a CodeHead-et. Ezután a simplifyHintList-tel egyszerűsítteti a súgás listát, majd az egyszerűsített listára és a CodeTail-re rekurzívan meghívja magát. Így felépülnek a titkos kódok.

simplifyHintList

Egyszerűsíti a súgás listát. Igazából mindegyik elemére meghívja a simplifyHint-et, és azok eredményeiből egy új listát épít fel, ez lesz a saját eredménye.

simplifyHint

Az utolsó építőkő a megoldáshoz. A kapott súgást egyszerűsíti azt feltételként használva, hogy a kód első bábuja a kapott szám (CodeHead). Itt történik a súgások elemenkénti kezelése. Összeségében 4 eset lehet, amiket mind különböző módon kell kezelni.

Először vizsgáljuk meg azokat az eseteket, amikor a Tipp első helyén a kapott CodeHead szám áll. Ekkor tudhatjuk, hogy egy fekete pálcának kell erre vonatkoznia, tehát a feketék számát eggyel csökkentjük. Továbbá a Tipp első elemét elhagyjuk, és a TippZsak-ból is kiveszünk egy CodeHead-nek megfelelő értéket, hiszen amit elhasználtunk feketére, azt már nem használhatjuk el fehérre.

Itt viszont ismét két esetre bomlik a megoldásunk, mert nem biztos, hogy a TippZsak-ban van még CodeHead érték. Természetesen az előfeldolgozás során biztosan kellett, hogy kerüljön bele ilyen érték. Ha most nincs, azt csak az okozhatta, hogy korábban azt hittük, hogy fehér válasz vonatkozik egy bábura, most viszont kiderült, hogy valójában fekete, ami mivel „erősebb”, ezt felülírja. Ilyenkor semmi gond, szerencsére utólag is meg tudjuk javítani a dolgot, egyszerűen csak a fehér válaszpálcákhoz hozzáadunk egyet.

A másik két eset akkor jön elő, ha a Tipp eleje nem egyezik CodeHead-el.

Ekkor, ha sikerül kivennünk a TippZsak-ból CodeHead-et, az azt jelenti, hogy a Tippben valahol van CodeHead értékű bábu tehát fehér pálcának is kellett vonatkoznia rá, ezért a fehérek számát csökkentjük. (Persze, ez később még felülíródhat egy fekete válasszal.)

Ha alapból sincs a TippZsak-ban CodeHead értékű bábu, akkor ez az igazán potyabábu, nem vonatkozhat rá semmilyen válasz, tehát a feketék és fehérek számát is változatlanul hagyjuk.

Keresési tér szűkítése

Ez eddig egy működő megoldás, csupán egy ellenőrzést kell elvégeznünk az eljárás után, hogy konzisztensek maradtak-e a súgásaink. Tehát a feketék és fehérek száma egyaránt pontosan nullára csökkent.

Viszont ez így egy Generate-and-Test típusú megoldás, ami minden lehetséges esetet ellenőriz. Ehelyett vágásokkal csökkenthetjük a keresési teret.

Ezt a simplifyHint eljáráson belül fogjuk csak megtenni. Azt kell észrevegyük, hogy egyszerűsítés után a feketék és fehérek számának (NewBlack, NewWhite) mindig teljesítenie kell(ene) a következő feltételeket. Bármelyik feltétel sérülése esetén biztosak lehetünk, hogy ebbe az irányba nem érdemes tovább keresni, mert nem lesz olyan titkos kód, amely megfelel a súgás listának. A feltételek a következők:

1. NewBlack >= 0
    - Elég triviális, hogy a feketék száma nem mehet negatívba.
2. NewBlack =< L
    - Szintén biztos az ellenmondás, ha még több bábut kellene pontosan eltalálnunk, mint maga a kód hossza.
3. NewBlack+NewWhite >= 0
    - A fehérek száma időközben lehet negatív, ám annyira semmiképpen sem, hogy ezt a feketék ne tudják később kompenzálni
4. NewWhite+NewBlack =< L
    - Hasonló a helyzet, több bábut kell vagy pontosan, vagy csak a színét eltalálni, mint amilyen hosszú a kód

Továbbá észrevehetjük azt, hogy amikor az eljárás L=0 értékkel hívódik, akkor a NewBlack és NewWhite értéke is szükségszerűen nulla, tehát szerencsére nem szükséges utólagos ellenőrzés, mert ez a fajta vágás ezt is megcsinálja.

Egy kicsit még egyszerűsíthetünk a dolgon azzal, hogy ezek után feltehetjük, hogy a simplifyHint-be úgy kerültünk bele, hogy ezek a feltételek már teljesültek egy eggyel nagyobb L számra. Ezért nem muszáj mindig mind a négy esetet vizsgálni, csak azokat, amelyek sérülhettek. Például ha NewBlack=Black-1, akkor tudhatjuk, hogy 2)-es feltétel biztosan igaz, viszont az 1)-es megsérülhetett stb.

Kódolás Prolog nyelven

Először a Prolog nyelvű megoldást mutatnám be, mert az jobban alkalmazkodik az elméleti megoldáshoz. Hiszen maga a Prolog gép is egy visszalépésest keresést végez, és én ezt használtam ki.

Az mmm predikátum törzsében hivatkozunk egy max perdikátumra, ezzel állítjuk elő CodeHead értékét, amely gyakorlatilag Max db választási pontot hoz létre, felsorolva a lehetséges értékeket. Ezek után meghívjuk rekurzívan az mmm predikátumot. Meghiúsulás esetén a visszalépés automatikus, és egy másik számmal próbálkozunk.

Érdemes még megemlíteni, hogy a TippZsak-ot egy egyszerű listával reprezentáltam, és a select/3-el vettem ki belőle az elemeket. Meghiúsulás esetén tudtam, hogy nincs az elem a listában, siker esetén pedig egyből előállt az új TippZsak is.

Eredetileg ezt saját, az első háziból ihletett adattípussal oldottam meg. Ez valamennyit gyorsíthatna a futáson, hiszen tömörebben tartalmazza az adatokat, így rövidebb listában kell keresni. Ekkor azonban meg kellet írni ezt az adatstruktúrát menedzselő logikát, (pontosabban másolni az első kisháziból) és saját select-et írni hozzá. Végül, mivel ez nagyságrendileg nem okoz gyorsulást, főleg nem rövid vagy sok különböző elemet tartalmazó listákra, azonban a prolog kód hosszát megduplázza, és kicsit bonyolítja az előfeldolgozást is, végül ezt a módszert elvetettem. Azonban Erlang esetében mégis megtartottam ezt a módszert, ott leírtak miatt.

Úgy gondolom, hogy minden más, amire most nem tértem ki, az a kódból világosan látszik, még jobban is, mintha folyószöveggel kifejteném.

Kódolás Erlang nyelven

A megoldás természetesen ugyanaz, mint Prolognál, azonban más nyelvi elemeket kellett használnom.

Először is a visszalépéses keresést, pontosabban bejárást, listanézettel valósítottam meg. Az mmm olyan listát generál, melyek elemei olyan listák, melyek első eleme 1 és Max közötti érték, a listák farkát pedig a rekurzív hívás szolgáltatja. Azonban, hogy a keresési teret szűkítsük, szükségünk van visszalépésekre is. A Prolog meghiúsulásos viselkedését throw(error)-ral váltottam ki. Amennyiben a rekurzív hívás „meghiúsul” (hibát dob), a farok generátorlistájának üres listát adunk, ezzel biztosítva, hogy ilyen kezdetű kódot ne generáljunk egyet sem.

Egy másik pont, ahol az Erlang megoldás jelentősen eltér, az a TippZsak manipulálása. A különbség itt is az, hogy Prologban kihasználtuk a select függvény meghiusulást, Erlangban viszont a delete függvény változatlanul visszaadja a kapott listát, amennyiben nem találta a törlendő elemet. Persze, egy egyenlőségvizsgálattal megtudhatjuk, hogy a törlés sikeres volt-e, de érezzük, hogy ekkor kétszer is végig kell mennünk a listán, egyszer törlésnél, másodszor ellenőrzésnél. Ez motiválja, hogy saját törlő függvényt írjunk. Ez hasonlóan működik, mint a delete, de ha nincs találat not_found értéket ad vissza. Ez a függvény a remove_from_zsak. Továbbá, „ha már lúd, legyen kövér,” felhasználtam, a korábban már említett Zsák alakú listát is, mint multihalmaz implementáció. Az itt felhasznált függvények elég egyszerűek, és a remove_from_zsak kivételével az első kisházi újra felhasználása.

A megoldás értékelése

A megoldás legnagyobb előnyének értékelem az egyszerűségét. Kicsit hosszúra nyúlt a dokumentáció, de úgy gondolom, hogy aki megérti az alapötletet, és a négy esetet, amit kezelni kell, az átlátja már a kódot és az algoritmust is. A vágásokkal a program viszonylag jó

futásidőket mutat, a kiadott teszteseteket egy kivételével 1 másodperc alatti idő alatt megoldja, viszont egyre – programnyelvtől függően – 12-20 másodpercig is képes futni.

Nyilvánvalóan az algoritmus helyett lehet másikat, jobbat csinálni, vagy ezt is lehetséges még tovább fejleszteni. Fejlesztési irány például, hogy jobban figyelünk, hogy amikor fehér pálcával jelölünk a kód egy bábuját, akkor azt azért tesszük mert a tipp sorrendi feldolgozása során az aktuális pozíció előtt találtunk azonos színűt, vagy azért, mert az után. Nyilván azért nem mindegy, mert az első esetben nem árthatunk senkinek, utóbbi esetben viszont elképzelhető, hogy egy feketével jelzendő bábu elől „happoljuk el" a lehetőséget. Ezek segítségével új kényszereket is felírhatunk, amelyekkel tovább csökkenthetjük a keresési teret. Ezt egyébként el is végeztem, de mivel a kódot rendkívüli módon elbonyolította, ezért a végleges megoldásban nem hagytam benne, bár valamivel gyorsabban futott.

Az algoritmusnak viszont nagy gyengesége, hogy a súgások egymástól függetlenül döntenek, hogy az adott titkos kód lehetséges-e. Vagyis a súgások közötti egymásra hatások lehet, hogy csak nagyon későn derülnek ki. Például, a következő esetben a program kb 7 másodpercig fut, pedig, ha az utolsó számjegyeket vizsgáljuk, rögtön láthatjuk, hogy egy titkos kód sem lehet ilyen. És ami rossz hír, hogy ugyanezt a bemenetet növelve a futásidő exponenciálisan nő.

```
5
1 1 1 1 1 1 1 1 1 1 1 0/0
1 1 1 1 1 1 1 1 1 1 2 0/1
1 1 1 1 1 1 1 1 1 1 3 0/1
1 1 1 1 1 1 1 1 1 1 4 0/1
1 1 1 1 1 1 1 1 1 1 5 0/1
```