Deklaratív programozás

1. nagy házi feladat (Elixir)
Számtekercs

Tartalomjegyzék

TÖMÖRI PÉTER ANDRÁS

I4RZ0O

2024/25-ÖS TANÉV, ŐSZI FÉLÉV

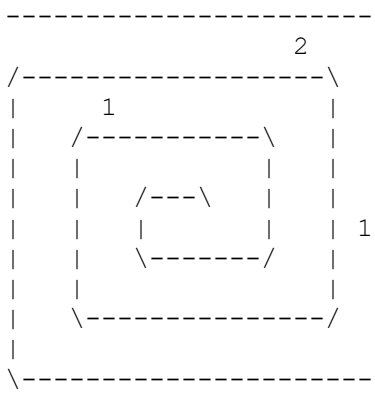
Feladatspecifikáció – a számtekercs feladvány

Adott egy n*n mezőből álló, négyzet alakú tábla, amelynek egyes mezőiben 1 és m közötti számok vannak. A feladat az, hogy *további* 1 és m közötti számokat helyezzünk el a táblában úgy, hogy az alábbi feltételek teljesüljenek:

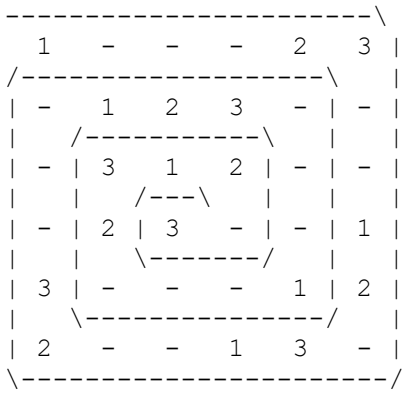
1. Minden sorban és minden oszlopban az 1..m számok mindegyike pontosan egyszer szerepel.
2. A bal felső sarokból induló *tekeredő vonal* mentén a számok rendre az 1,2,...m,1,2,...,m,... sorrendben követik egymást.

A *tekeredő vonalat* a következőképpen definiáljuk. Először a négyzet első sorában haladunk balról jobbra, majd az utolsó oszlopban felülről lefelé. Ezután az utolsó sorban megyünk jobbról balra, majd az első oszlopban alulról fölfelé, egészen a 2. sor 1. mezőjéig. Miután így bejártuk a négyzetes tábla szélső sorait és oszlopait, rekurzívan folytatjuk a bejárást a 2. sor 2. mezőjében kezdődő (n-2)*(n-2) mezőből álló négyzettel.

Az 1. ábra egy feladványt ábrázol, a 2. ábra ennek (egyetlen) megoldását mutatja.



1. ábra. Egy feladvány (n=6, m=3)



2. ábra. A feladvány megoldása

A megvalósítandó helix/1 függvény egyetlen paramétere a feladványt írja le, visszatérési értéke pedig a feladvány összes megoldásának a listája, tetszőleges sorrendben. Ha egy feladványnak nincs megoldása, a visszatérési érték az üres lista. Egy-egy megoldás a kitöltött táblát írja egész számokból álló listák listájaként. Minden szám a tábla egy mezőjének értéket adja meg: ha 0, a mező üres, egyébként pedig az i értek, ahol 1 ≤ i ≤ m

Az 1. ábrán bemutatott feladvány esetén például a helix/1 függvény paramétere:

```
{6, 3, [{{1,5},2},{{2,2},1},{{4,6},1}]}
```

A 2. ábrán látható megoldást írja le például az alábbi (egyelemű) megoldás-lista:

```
[[[1,0,0,0,2,3],
  [0,1,2,3,0,0],
  [0,3,1,2,0,0],
  [0,2,3,0,0,1],
  [3,0,0,0,1,2],
  [2,0,0,1,3,0]]
]
```

Követelmények elemzése

Feladat áttekintése

A feladat egy n*n méretű négyzetes tábla kitöltése olyan módon, hogy:

- Minden sorban és minden oszlopban az 1-től m-ig terjedő számok mindegyike pontosan egyszer szerepeljen.
- A bal felső sarokból induló tekeredő vonal mentén a számok hézagok megengedésével az 1, 2, ..., m, 1, 2, ..., m sorrendben kövessék egymást.

A feladat további korlátozást tartalmaz: A táblán egyes mezők előre megadhatók definiált értékkel, melyek nem módosíthatók.

Paraméterek

A feladat inputja egy hármas {n, m, constraints}, ahol:

- n: a tábla mérete (n*n-es tábla).
- m: a kitöltendő számsorozatok utolsó eleme.
- constraints: egy lista, amely mező-érték párokat tartalmaz. Minden listaelem {{row, col}, value} formátumú, ahol row és col a mező sor- és oszlopindexe, value pedig az adott mező értéke, amit nem lehet megváltoztatni.
- A bemenet helyességét a függvény hívásakor már feltételezzük.

Feltételek és korlátozások

- A függvény a feladvány összes megoldásának listáját adja eredményül.
- Ha nincs megoldás akkor üres listával tér vissza.
- A számtekercsnek megfelelő bejárásban a számoknak növekvő sorrendben kell lenni, kivéve az m utáni számot, ami újra 1.
- Nem kell minden mezőn számnak lennie, maradhatnak hézagok: ezt az implementáció során a 0-k jelentik.
- Minden sorban és oszlopban 1-től m-ig minden szám pontosan egyszer szerepel.
  - ➔ Minden számjegy pontosan n-szer szerepel egy megoldásban.
  - ➔ Pontosan (n*n – n*m) darab 0-s van egy megoldásban, és n-m db minden sorban és oszlopban.
- A constraints-eket nem szabad módosítani.

Tervezés

Alapgondolatok

Fontos részlet, hogy az általam tervezett implementációban előre definiált értékként csak konkrét szám adható meg, az viszont, hogy üresnek kell maradnia egy mezőnek nem köthető ki. Így csak az ilyen jellegű bemenetek megoldására lesz képes.

A feladatot a generate-and-test módszerrel közelítem meg, tehát létrehozom az összes lehetséges elrendezést és csak a követelményeknek megfelelőket tartom meg. A létező összes megoldást úgy hozom létre, hogy a részleges megoldásom egy befejezetlen celláját véglegesítem a lehetséges értékek hozzárendelésével, ezáltal 1 részmegoldásból esetleg több, a megoldáshoz közelebb álló részmegoldást hozok létre. Ugyanakkor ezt több allépéssel optimalizálom, melyeknek alapelve az, hogy ha bármelyik részmegoldásról kiderül, hogy már nem lehet megfelelő, akkor az a rekurzióból (mellyel egy megoldást lépésenként hozunk létre) üres listával visszatér.

- A mezők véglegesítését a számtekercsnek megfelelő sorrendben hajtom végre, így, ha tárolom, hogy a számsorozat növekvő sorrendjében milyen számnak kell következnie, már csak azzal a számmal, vagy 0-val véglegesítem. Ezáltal, ha minden részmegoldást egy csomópontként fogunk fel, melyek közt élek jelölik, hogy honnan jutottunk el oda, akkor az ebből kapott fa elágazási tényezője m helyett 2 lesz.
- A kényszerek előre elhelyezve lesznek a kiinduló állapotban, mivel azoknak úgyis ott kell lenni. A megoldás készítés során minden mezőnél ellenőrizzük, hogy már van-e benne elem a kényszer miatt. Amennyiben a benne levő elem és a növekvő sorrend ellent mond egymásnak, a részmegoldást elvágom.
- Minden sorhoz és oszlophoz segédstruktúrában tárolva a már hozzáadott 0-k számát biztosítható, hogy minden sorban és oszlopban minden szám pontosan egyszer lehessen. A részmegoldás elágazásakor, ha a 0-t választjuk, mint hozzáadandó elem, akkor leellenőrizzük, hogy van-e még hozzáadható nulla az aktuális sorhoz és oszlophoz. Ha nem, akkor a részfát elvágom, ha igen, frissítem a hozzáadott nullák számát.
- Ha egy nem nulla elemet adok hozzá egy mezőhöz, ellenőrzöm, hogy az adott sorban és oszlopban szerepel-e már az adott elem. Ha szerepel, akkor az hibás megoldás lenne, ezért elvágom az ágat.
- Eltárolom a teljes megoldásba még szükséges számsorozatok számát és 0-k számát, így, ha valamelyik 0 lesz, tudom, hogy a két ágból már csak az egyik lehetséges, valamint, ha mindkettő 0, akkor egy megoldás elkészült

Az előbb felsorolt köztes műveletek bár egy-egy sor/oszlop ellenőrzését követelik, így a megoldásgenerálást lassítva, de a lehetséges megoldáshalmazok számát jelentősen lecsökkentik, így sokkal kevesebb részmegoldás létrehozását szükségessé téve.
Az előbbi algoritmusnak köszönhetően nem történik olyan elem hozzáadása, melyből explicit következik a részmegoldás kudarcra ítélése. (Természetesen további algoritmusokkal hatékonyabb működés még elérhető)

Megvalósítandó algoritmus

Az algoritmus az alábbi lépésekből áll:

1. **Adatok Inicializálása**: A részmegoldás tárolásához szükséges struktúrák inicializálása a constraints hozzáadásával. A számtekercs szerinti bejárásnak megfelelő sorrend megállapítása.
2. **Megoldások Generálása kényszerek ellenőrzésével**: Rekurzív megoldásgenerálás, amely a számtekercsbeni bejárás szerint adja hozzá a következő mezőhöz a lehetséges értékeket külön ágak formájában. A kényszerek ellenőrzése is itt történik, nem hoz létre új részmegoldást olyan számmal, ami által biztosan nem megfelelő a részmegoldás.
3. **Eredmények Összegyűjtése**: Minden sikeresen befejezett megoldás leképzése a kimenet elvárt alakjára.

Típusok specifikálása

`puzzle_desc()` egy hármas ami, `size()`, `cycle()` és `[field_value()]`-t tartalmaz
`size()` a feladvány mérete: n szigorúan nagyobb, mint 0
`cycle()` a számsorozat hossza: m, 1-től n-ig tartó szám lehet
`field_value()` egy kettes, ami `field()` és `value()`-ból áll
`field()` egy kettes, ami `row()` és `col()`-ból áll
`row()/col()` a sor/oszlop száma, 1-től n-ig tartó szám lehet
`value()` előre megadott érték, 1-től m-ig tartó szám lehet
`solutions()` `solution()`-ök listája, az összes megoldás
`solution()` `retval()`-ok listájának listája: egy megoldás listák listájaként
`retval()` (rész)eredmény egy mezőjének értéke, 0-tól m-ig tartó szám lehet
`index()` a feladvány egy mezőjének számtekercsbeni sorszáma, 1-től n*n-ig tartó szám
`matrix_map()` a készülő megoldás `Map` struktúrában tárolva. A Map kulcsa egy `field`, értéke pedig a field értéke (`retval`)
`row_map()` a készülő megoldás `Map` segédstruktúrája sorbeli egyediség hatékonyabb ellenőrzéséhez. A Map kulcsa a sor sorszáma (`row`), értéke pedig a sorban szereplő értékek listája `[retval]`
`col_map()` a készülő megoldás `Map` segédstruktúrája oszlopbeli egyediség hatékonyabb ellenőrzéséhez. A Map kulcsa az oszlop sorszáma (`col`), értéke pedig az oszlopban szereplő értékek listája `[retval]`
`row_zero()` a készülő megoldás `Map` segédstruktúrája sorbeli kitöltöttség hatékonyabb ellenőrzéséhez. A map kulcsa a sor sorszáma (`row`), értéke pedig a sorban szereplő 0-k száma, 0-tól (n-m)-ig tartó szám
`col_zero()` a készülő megoldás `Map` segédstruktúrája oszlopbeli kitöltöttség hatékonyabb ellenőrzéséhez. A map kulcsa az oszlop sorszáma (`col`), értéke pedig az oszlopban szereplő 0-k száma, 0-tól (n-m)-ig tartó szám

Elírás: a beadott megoldásban n-m helyett m-et írtam!

Függvények specifikálása

`helix(sd::puzzle_desc()) :: ss::solutions()`
A `helix/1` függvény a megadott puzzle leírása (sd) alapján generálja a lehetséges megoldásokat. Az `init_maps/2` meghívásával inicializálja a szükséges map struktúrákat, majd meghívja a `gen_sols/11` függvényt a tényleges megoldásgenerálásra (átadva a megfelelő paramétereket, többek között a bejárás sorrendjét `next_fields/4` használatával). A visszakapott megoldásokat átalakítja a megfelelő formátumra (ss).

`init_maps(fvs::[field_value()], {sol::matrix_map(), rm::row_map(), cm::col_map()}) :: {sol::matrix_map(), rm::row_map(), cm::col_map()}`
Az `init_maps/2` függvény létrehozza a kiindulási állapotot a megadott kényszerek alapján. Visszatér a frissített map struktúrákkal.

`next_fields(direction::atom(), steps_remaining::integer(), start_step_count::integer(), fields::[field()]) :: fields::[field()]`
A `next_fields/4` függvény a megadott irány `(:R jobb, :D le, :L bal, :U fel)` és lépések alapján határozza meg a következő mezőt és a bejárás folytatását. Az adott irányba hátralevő lépések száma `steps_remaining, start_step_count` pedig az, hogy ez mennyiről indult kanyarodáskor. A függvény rekurzív módon akkumulátorban tárolva hozza létre a teljes bejárást, vagyis a kiterített számtekercset fordított sorrendben!

`gen_sols(sol::matrix_map(), rm::row_map(), cm::col_map(), m::cycle(), rem::integer(), zero::integer(), nextval::value(), next_fields::[field()], max_zero::integer(), rzm::row_zero(), czm::col_zero()) :: sols::[matrix_map()]`
A `gen_sols/11` függvény rekurzív módon generálja az összes megfelelő megoldást, figyelembe véve a minden meghatározott feltételt. A részmegoldást tárolja `sol`, az egyediség ellenőrzéséhez használt struktúrák `rm` és `cm`, a kitöltöttséghez használt struktúrák `rzm` és `czm`, a számsorozat hossza `m`, a hátralévő számsorozatok száma `rem` (értéke 0-tól n-ig tartó szám), a hátralévő 0-k száma `zero` (értéke 0-tól (n*n-n*m)-ig tartó szám), a növekvő sorrend szerinti következő érték `nextval`, az egy sorban/oszlopban megengedett 0-k száma `max_zero` (értéke n-m). A kiterített számtekercs hátralevő része `next_fields`, melynek első eleme az éppen kitöltendő mező (`field`). Ez a függvény felel a részmegoldások létrehozásáért és megoldások tárolásáért. A kényszerek ellenőrzését és a rekurzív hívást a `check_and_create_sol/12` függvényre bízza.

`check_and_create_sol(sol::matrix_map(), rm::row_map(), cm::col_map(), m::cycle(), rem::integer(), zero::integer(), value::retval(), nextval::value(), next_fields::[field()], max_zero::integer(), rzm::row_zero(), czm::col_zero()) :: sols::[matrix_map()]`
A `check_and_create_sol/12` függvény ellenőrzi, hogy az új mező értéke megfelel-e a kényszereknek, és ha igen, frissíti a megoldást és folytatja a generálást a `gen_sols/11` megfelelő meghívásával. Új paramétere a `gen_sols`-hoz képest a `value`, ami az éppen hozzáadandó értéket jelenti (vagy `nextval` vagy `0`). A kényszerek ellenőrzését és struktúrák frissítését az `update_and_check_maps/8` függvényre bízza

```
update_and_check_maps(sol::matrix_map(), rm::row_map(),
cm::col_map(), field::field(), value::retval(), max_zero::integer(),
rzm::row_zero(), czm::col_zero()) :: {result::boolean(),
{new_sol::matrix_map(), new_rm::row_map(), new_cm::col_map(),
new_rzm::row_zero(), new_czm::col_zero()}}
```
Az `update_and_check_maps/8` függvény frissíti a megoldást és a segéd map-eket az új elemmel, amennyiben minden kényszer teljesül. Az aktuális mező `field`, `value` a mezőbe írandó érték, `result` az eredmény, hogy a szám megfelelt-e a követelménynek. A constraints szerinti teljesülést a `satisfy_constraints/2` függvényre bízza, a kitöltöttség és egyediség ellenőrzését ezután végzi el.

```
satisfy_constraints(value::retval(), map_value::integer()) ::
{result::boolean(), should_update::boolean()}
```
A `satisfy_constraints/2` függvény ellenőrzi, hogy az új mezőre vonatkozó kényszerek teljesülnek-e. A hozzáadandó érték `value`, az aktuális mező helyén szereplő elem pedig `map_value`, ami `nil`, ha nem szerepel ott még érték (tehát nem volt rá kényszer). Visszatér a kényszerek teljesülésének eredményével (`result`) és azzal, hogy a map-eket frissíteni kell-e (`should_update`).

Implementáció

`helix/1`

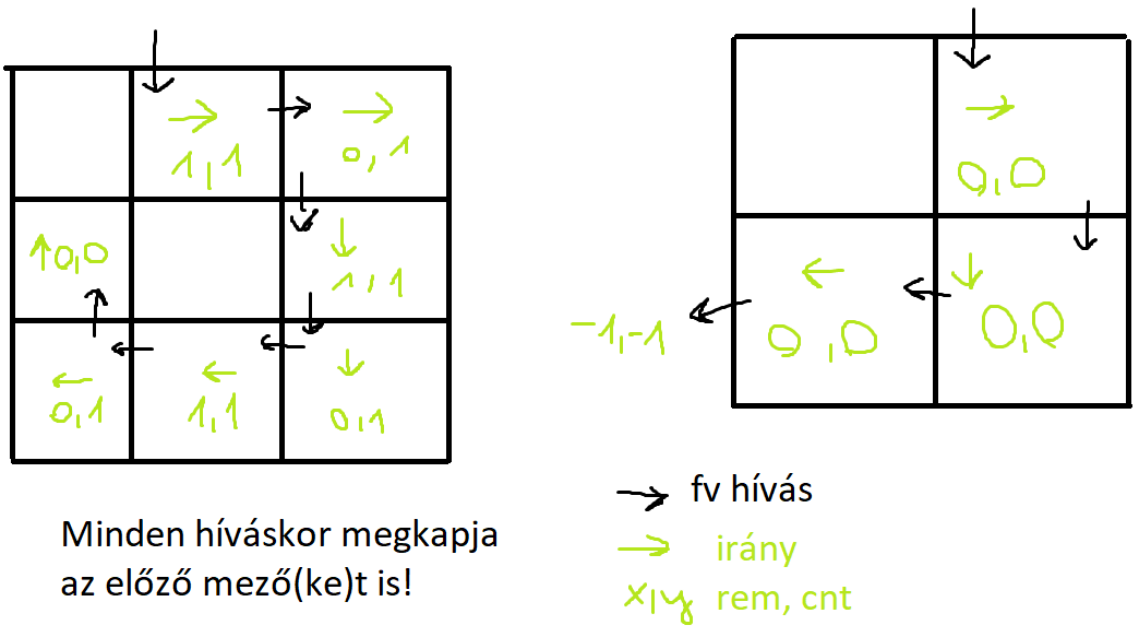
`init_maps/2`-nek átadom a kényszerek listáját és három üres `Map`-et. A viszatérési értékét elmentem három változóba, amit `gen_sols/11`-nek adok tovább. A további paraméterek értékei értelemszerűen: `m`=m, `rem`=n, `zero`=n*n-n*m, `nextval`=1, `max_zero`=n-m, `rzm` és `czm` üres `Map`-ek. `next_fields` értékét úgy határozom meg, hogy meghívom a `next_fields/4` függvényt és az eredmény listát `Enum.reverse/1` segítségével megfordítom! A `next_fields`-et úgy hívom meg, hogy az első elemet {1,1} már hozzáadom, így csak n-2 lépést kell megtenni, :R irányba. A `gen_sols/11` visszatér az összes helyes megoldással, mint `matrix_map()`-ek listájával. A `List.flatten/1` segítségével az üres listákat (elvágott megoldások) kiszűröm, majd a maradékon végigiterálok és sorfolytonosan lekérem a map adott oszlopának értékét ezzel létrehozva egy-egy sort, majd azok listáját (ha nem található érték adott kulccsal, akkor ott üres maradt, ekkor 0-t adok vissza, mint default érték!). Ezeket a megoldásokat szintén listákba fűzöm és visszatérek vele. Ennek megvalósításához 3 `for komprehenzió` éppen ideális, a listák-listájának-listája felépítés miatt.

`init_maps/2`

Két klóz aszerint, hogy a kényszerek listája kiürült-e vagy sem. Ha még van legalább egy kényszer, akkor meghívom az `init_maps/2`-t a frissített `Map`-eket átadva, és a hozzáadott kényszert kivéve. Ha már üres a kényszerek listája, akkor visszatérek a kapott `Map`-ekkel. A frissítés `sol`-nál csak egy `Map.put`, de `rm` és `cm`-nél a listát kell frissíteni, amihez `Map.get`-tel meg kell szerezni a listát, majd belerakni a listába az új értéket, majd az így létrejött listát `Map.put`-tal berakni a megfelelő sor/oszlopba.

`next_fields/4`

Klózokra bontás elsősorban az aktuális irány és a hátralevő lépések száma szerint. Mivel a rekurziónak egyszer véget kell érnie, ezért további klóz szükséges a végállapot ellenőrzésére. Mivel ez csak `:U`-nál fordulhat elő, ezért azt, raktam a végére, hogy a várható illesztések száma kevesebb legyen. Amennyiben úgy megy egy adott irányba, hogy `rem` még nem 0, akkor a rekurzív hívás során `fields` elejére hozzáadom az aktuális mezőt (az iránynak megfelelően lépve a legutóbbi mezőből számítva), és csökkentem a `rem` értékét eggyel. `Dir` és `cnt` változatlan. Fontos, hogy adott iránynál először kell szerepelnie a `rem`=0 klóznak, mint az általános klóznak, különben sosem fog kanyarodni. Amennyiben `:R, :D vagy :L` irányba kell menni, de a `rem` már 0, akkor rekurzív híváskor még hozzáadom az aktuális mezőt az előbbi esethez hasonlóan, viszont az irányt megváltoztatom (:R → :D → :L → :U). A `cnt` értékére be kell állítani a `rem`-et, viszont ha az irány `:L` volt és `:U` lett, akkor előbb a `cnt`-t csökkenteni kell eggyel, mivel felfele már nem megy a sarokig (és `rem`-nek is a csökkentett értéket kell átadni). Ha az irány `:U, rem=0 és cnt=0`, akkor elértünk a páratlan méretű mátrixunk utolsó előtti elemére. Ekkor még hozzáadjuk az aktuális és a következő/utolsó/középső mezőt a `fields`-hez, és ezzel visszatérünk. Ha a `cnt` még nem 0, akkor is hozzáadjuk az említett két mezőt (vö. helix is hozzáadta már {1,1}-et, a konzisztens algoritmushoz ennek adottnak kell lennie), csak a bejárás nem ért még véget. A `cnt`-t eggyel csökkenteni kell, a `rem`-et pedig erre beállítani (az irány természetesen `:R` lesz). Páros méretű mátrix esetén a bejárást tulajdonképpen `:L` irányba kellene abbahagyni. Azért, hogy ne kelljen még egy irányt 2-nél több klózra felbontanom, inkább úgy döntöttem, hogy az ilyenkor is haladjon tovább. Ekkor a `cnt`-t csökkenti eggyel, de mivel az amúgy is 0 volt, így -1 lesz. Tehát ha az irány `:U` és a `cnt/rem` értéke -1, akkor páros mátrix végére értünk úgy, hogy már minden elem hozzá is lett adva, szóval csak `fields` a visszatérési érték. Fontos, hogy az általános `:L` klóz csak ezek a speciális 0 és -1 feltételű klózok után állhat.



3. ábra: számtekercs-bejáró algoritmus bemutatása

`gen_sols/11`

Két klóz aszerint, hogy egy megoldás végére értünk-e. Ha a `rem` és `zero` is 0, akkor elkészült a megoldás, vissza lehet térni vele egyelemű listaként. Különben még folytatni kell a részmegoldás fejlesztését. Ezt egy kételemű lista létrehozásával valósítom meg úgy, hogy a `check_and_create_sol/12` függvény eredménye lesz a két elem. Az egyik függvényhívásban a `value`-t `nextval`-ra állítom, a másikban pedig `0`-ra.

`check_and_create_sol/12`

Három klóz a hozzáadandó elemtől függően. Az első klózba akkor kerül bele, ha a `value` `0` és `zero>0` (őrfeltételként, különben már nem lehetne 0-t hozzáadni). A második klózra akkor illeszkedik, ha a `rem>0` (itt tudjuk, hogy `value=nextval`). Különben pedig a harmadik klózba fog esni: ide akkor kerül, ha a hozzáadandó értékből már nem rendelkezünk többel, ekkor üres listával tér vissza, elvágva ezt az ágat. Az első két klózban egyformán az `update_and_check_maps/8` meghívásával ellenőrizzük a kényszereket és frissítjük az adatszerkezeteket, majd, ha a visszatérési értéke igaz volt, rekurzívan meghívja a `gen_sols/11`-et a frissített adatszerkezetekkel és további értékekkel: a `value=0` klóznál `zero` értékét kell csökkenteni eggyel, minden más marad, míg a második klózban `value` értékétől függően kell frissíteni az értékeket. Ha a `value` megegyezik `m`-mel, akkor egy számsorozat végére értünk. Ilyenkor a `rem` értéke csöken eggyel és a `nextval` 1 lesz. Ha `value` nem egyezik meg `m`-mel, akkor a `nextval` értékét eggyel növeljük. Amennyiben az `update_and_check_maps/8` visszatérési értékének `bool`-ja hamis, akkor üres listával térünk vissza mindkét klózban, elvágva a nem megfelelő megoldás ágát.

`update_and_check_maps/8`

Kettő klóz szintén value-tól függően (0/nextval). Mindkét esetben egyformán a `satisfy_constraints/2` meghívásával ellenőrzi, hogy az előre megadott mező-érték kényszereknek megfelel-e az aktuális érték. Ehhez a `sol`-ból lekéri a `field`-hez tartozó értéket és `value` mellett ezt is átadja. Fontos, hogy a `Map.get` visszatérési értéke `nil`, ha az adott kulcs nem található. Ebben az esetben ez azt fogja jelenteni, hogy a mezőre nem vonatkozik érték kényszer. A függvény visszatérési értékeként megtudjuk, hogy a kényszernek megfelel-e az érték, és hogy kell-e a struktúránkat frissíteni. Ha a kényszernek nem felel meg (`constraint_passed=false`), akkor `hamis` értékkel tér vissza mindkét klóz (a többi paraméter lényegtelen, mert úgyis el lesz vágva a megoldás). A két klóz szétválasztása azért is előnyös, mert a két különböző esetben épp különböző további feltételeket kell ellenőrizni. Ha a `value` 0, akkor a `should_update_map` értékével nem is kell foglalkozni, mert az `false`: A struktúráimat úgy építem a megoldások létrehozása során, hogy 0-kat nem adom hozzá (vö. `helix`-nél megoldások átalakításakor default érték). Mivel 0-t adok hozzá, a sor/oszlop egyediségét sem kell ellenőröznöm, viszont a kitöltöttséget, azaz, hogy adhatok-e még hozzá 0-t a sor/oszlophoz, igen. Ehhez lekérem `rzm`-ből és `czm`-ből az adott `row` és `col` értékét, és amennyiben azok még kisebbek, mint `max_zero igaz` értékkel, változatlan `sol`, `rm` és `cm`-mel és az adott `row` és `col`-ban eggyel növelt `rzm` és `czm`-mel térek vissza, de ha nem (azaz valamelyik már elérte a max nullák számát), akkor `hamis` értékkel térek vissza. A `max_zero` azért segít a megoldáshalmaz szűkítésében, mert ha pl. van egy 10 hosszú sor

amit 1-6 fel kell tölteni, nem hozunk létre olyan megoldásokat amik még nem értek végig a soron, de már nem lehetne kitölteni 6-ig minden számmal a helyhiány miatt. A másik klózban ugyanakkor éppen ezt nem kell ellenőrizni, viszont a sor- és oszlopbeli egyediséget igen. Először is, ha nem kell frissíteni a struktúrákat, akkor `igazzal` tér vissza és azokkal a paraméterekkel, amiket kapott. Különben leellenőrzi a sor és oszlop egyediségét és ha mindkettőnek megfelelt, csak akkor tér vissza `igazzal` és a frissített struktúrákkal, különben `hamissal` tér vissza. Az egyediség ellenőrzését két külön feltételvizsgálattal csináltam, azért, hogy ne kelljen feleslegesen lekérni mindkét `Map` listáját, ha az egyiknek már úgysem felelt meg. A lekért listákat elmentem egy változóba, így a frissítéskor (amit csak akkor végzek el, ha mindennek megfelelt), nem kell ismételten lekérni. Tehát a sorbeli egyediség ellenőrzéséhez az `rm`-ből lekérem a `row`-hoz tartozó listát, majd `Enum.all?/2`-vel megvizsgálom, hogy a `value` egyik elemmel sem egyezik meg. Oszlopbeli egyediséget hasonlóan végrehajtom. A struktúrák frissítésekor `sol`-ba egyszerűen `put`-olom `field`-hez a `value`-t, `rm` és `cm`-nél pedig a már lekért listába beleteszem `value`-t és azt `put`-olom `row`-hoz és `col`-hoz. `Rzm` és `czm` változatlan marad.

`satisfy_constraints/2`
A két érték összehasonlításához 4 klóz szükséges. Ha nincs az adott mezőn kényszer, akkor az `nil` lesz. Ekkor `value` értékétül függ a `should_update` értéke, de a `result` értéke biztosan `true` lesz. Ha `value` 0, akkor nem kell frissíteni a struktúrákat, egyébként igen. A másik az, hogy a mezőn van kényszer. Ekkor a `should_update` értéke biztosan `false` lesz, a `result` pedig attól függ, hogy a `value` mi. Amennyiben a `value` és a kényszer értéke megegyezik, akkor `true`, különben `false`.

Tesztelési megfontolások

A végleges megvalósítás hosszú úton ment keresztül, mely során számos apróbb és jelentősebb rész került módosításra/megvalósításra, a minél hatékonyabb működés érdekében. Ezekből szeretnék pár lényegesebbet kiemelni:

A rm és cm segédstruktúrák használata nélkül is megvalósítható a feladat, ugyanis ezek redundáns adatot tárolnak (sol-ban benne van ugyanúgy minden). Ugyanakkor, a sorbeli és oszlopbeli egyediséget sokkal egyszerűbben lehet így leellenőrizni, hogy közvetlenül egy listát ad eredményül, mint a field kulcsú mátrixból leképezni adott sor/oszlop elemeit. Teszteimen futtatva a mix modult, mértem a futásidőt és jelentős teljesítményjavulást eredményeztek a segéd Map-ek. Érdekesség továbbá, hogy kipróbáltam olyan Map-et is, ahol a kulcs ugyanúgy sor/oszlop sorszám, de értékként nem listát tárolok, hanem még egy Map-et, aminek kulcsai a számok (1-től n-ig), értékük pedig azt jelzi, hogy már szerepel-e az adott sor/oszlopban. Ezáltal a keresés O(1)-é válik az O(n) helyett, de jelentős hatékonyságjavulást mégsem mutatott. Mivel a listás megoldás karbantartása nekem szimpatikusabb volt (nem kell újra példányosítani mindig egy Map-et, hanem a láncolt listához csak +1 elemet teszünk pointerrel), így azt tartottam meg.

A next_fields függvény átalakítható next_field függvénnyé. Eredetileg nem azt csináltam, hogy előre elkészítettem a bejárás listáját és azt továbbadtam levéve az első elemet, hanem mindig átadtam a következő mező meghatározásához szükséges paramétereket (direction, steps_remaining stb.) és a következő mezőt, majd a rekurzó folytatásakor a következő mezőt úgy adtam át, hogy meghívtam a next_field függvényt átadva neki a szükséges paramétereket. Ennek az előnye, hogy nem kell egy teljes mátrixot listaként továbbadni, ráadásul kevesebb klózzal rendelkezett, mert a bejárás végi edge-casekre sem kell figyelni (mert ha véget ér a bejárás akkor a rem=0 zero=0 miatt úgyis leáll, nem használja fel next_field értékét). A hátránya talán az, hogy a függvények paraméterszáma nőne (de a paraméterek által elfoglalt hely kisebb lenne). Ugyanakkor meglepődve tapasztaltam, hogy a listás változat kisebb és nagyobb feladványok esetén is gyorsabb volt. Ehhez hozzátartozik érdekességként, hogy magának a bejárásnak is a végleges algoritmusa már egy második változat (az első a leírás alapján is intuitív rétegenkénti haladás alapú), ami jobbnak bizonyult, mint az első. Továbbá mivel így előre be lehet tölteni a listát kipróbáltam, hogy mi van, ha kis n-ekre (20-ig mentem) előre definiálok klózokat, ami iterálás helyett rögtön visszaadja a bejárást, amit én a programba égetve 'cache'-eltem. Szintén meglepett, hogy ez növelte a futásidőt.

Kitöltöttség ellenőrzése: a programom megvalósítása során először kifelejtettem annak a követelménynek az ellenőrzését, hogy minden sor/oszlopban pontosan egyszer kell minden számnak szerepelni. A `rem` és `zero` értékek és az egyediség vizsgálat ugyanúgy csak a helyes megoldásokat adták vissza, de jelentősen nagyobb eseménytér bejárásával. Ezt mutatja, hogy az eredeti megoldás a kiadott tesztekből az utolsó hármat már nem tudta időkorlátra megoldani. Ennek a javtására először azt találtam ki, hogy a `next_fields`-ben nem csak a mezőt adtam hozzá, hanem azt is, hogy a mező által befejeztünk-e sort vagy oszlopot (3 atommal: sor/oszlop ért véget, vagy semmi). Az ellenőrzéskor ennek függvényében ellenőriztem le, hogy az adott sorban oszlopban `m` db érték szerepel-e. Ezáltal már sikerült 10/10-et elérni, de a leghosszabb futás még 36 sec volt. A végleges megoldással (nem csak sor/oszlop végén ellenőrzöm a kitöltöttséget, hanem `max_zero` és `rzm+czm` segítségével mezőnként) négyzetgyökére csökkent a maximális futásidő.

`gen_sols`-ban a két esemény szétválasztását eredetileg `for` komprehenzióval csináltam, ahol megadtam a `0`, `zero`-t és a `nextval` `rem`-t, végül a megoldásokat `Enum.concat`-tal és `Enum.filter`-rel szűrtem. A listás megoldás a tesztelés során végül hatékonyabbnak mutatkozott.

`satisfy_constraints/2`:
Valószínűbb, hogy nincs constraint, ezért azt az esetet raktam előre. Ha sok a constraint, akkor a megoldástér úgyis jelentősen kisebb lesz, ezért hasznosabb a másik esetre fókuszálni.

A következő alternatív módosításaim nem javítottak az implementáción:
`Map`-ek inicializálása 0 értékekkel, tuple-k mellőzése, klózok felcserélése guard használatával

Kipróbálási tapasztalatok

A tervezés és fejlesztés során folytonosan végeztem verziókezelést és a különböző verziók tesztelését, így mindig az aktuális változtatásokat kipróbálva tudtam mérlegelni a módosítások végleges változatba olvasztását. A végleges megoldás a közepes méretű feladványokat 1.5 másodperc alatt oldotta meg.

Ugyanakkor azt tapasztaltam, hogy már n=7 esetén is problémát okoz a feladvány megoldása, ha egyáltalán nincs kényszer megadva. 7*7-es mátrixnál m=2-re még kaptam megoldást másodpercek alatt, de nagyobb m-re, illetve n=8-tól már m=2-re is, elszállt a program futásideje. Ebből is lehet következtetni, hogy a generate-and-test módszer akármennyire is minimalizálva van, nem a legoptimálisabb megoldást adja.