

# **TDK-dolgozat**

Dobai-Pataky Attila

BABEŞ–BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR  
MATEMATIKA ÉS INFORMATIKA KAR

## **Mértani problémák kollaboratív szerkesztése és valós idejű iteratív megoldása**



**Szerző:**

**Dobai-Pataky Attila**

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Adatelemzés és modellezés  
mesteri szak, I. év, Informatikai didaktikai mesteri szak, I. év

**Témavezető:**

**dr. Csató Lehel**, egyetemi tanár,

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Kolozsvár, 2024 május 5.

# Tartalomjegyzék

|                                                                   |          |
|-------------------------------------------------------------------|----------|
| <b>1. Bevezető</b>                                                | <b>1</b> |
| 1.1. Inspiráció: Sketchpad, SolveSpace . . . . .                  | 1        |
| 1.2. A dolgozat részei . . . . .                                  | 2        |
| 1.3. Bevezető példa . . . . .                                     | 2        |
| <b>2. A megoldó</b>                                               | <b>4</b> |
| 2.1. Bevezetés . . . . .                                          | 4        |
| 2.2. Derivált-alapú iteratív módszerek . . . . .                  | 4        |
| 2.2.1. Keresési irány megválasztása . . . . .                     | 5        |
| 2.2.2. Vonalkeresés . . . . .                                     | 6        |
| 2.2.3. Leállási feltétel . . . . .                                | 6        |
| 2.2.4. Összehasonlítás . . . . .                                  | 6        |
| 2.3. Ábrázolásmód . . . . .                                       | 7        |
| 2.3.1. Elemi változók . . . . .                                   | 7        |
| 2.3.2. Elemek . . . . .                                           | 7        |
| 2.3.3. Megkötések . . . . .                                       | 8        |
| 2.3.4. Kapcsolat elemek és megkötések közt . . . . .              | 10       |
| 2.4. A hibafüggvény differenciálszámítása . . . . .               | 10       |
| 2.4.1. Parciális deriváltak explicit megadása . . . . .           | 10       |
| 2.4.2. Szimbolikus differenciálás . . . . .                       | 11       |
| 2.4.3. Lokális és globális változók közti megfeleltetés . . . . . | 11       |
| 2.4.4. Algoritmikus differenciálás . . . . .                      | 12       |
| 2.4.5. A módszerek sebességének összehasonlítása . . . . .        | 13       |
| 2.5. Szabadsági fokok . . . . .                                   | 13       |
| 2.5.1. Szabadsági fokok feltérképezése . . . . .                  | 14       |

|           |                                                                   |           |
|-----------|-------------------------------------------------------------------|-----------|
| 2.5.2.    | Szabad változók ideiglenes rögzítése . . . . .                    | 14        |
| 2.5.3.    | Megoldhatatlan esetek kiszűrése . . . . .                         | 16        |
| 2.6.      | Egyéb megoldók és mértani reprezentációk az irodalomban . . . . . | 16        |
| 2.6.1.    | Szimbolikus megoldók . . . . .                                    | 17        |
| 2.6.2.    | Szabadsági fok-analízis alapú megoldók . . . . .                  | 17        |
| 2.6.3.    | Logikai és szabály-alapú megoldók . . . . .                       | 17        |
| <b>3.</b> | <b>A CAD-program architektúrája</b>                               | <b>18</b> |
| 3.1.      | Felépítés és kommunikáció . . . . .                               | 18        |
| 3.1.1.    | Közös definíciók szerver és böngészős kliens között . . . . .     | 19        |
| 3.1.2.    | Adatok közvetítése . . . . .                                      | 20        |
| 3.2.      | A szerver . . . . .                                               | 20        |
| 3.2.1.    | Versenyhelyzet elkerülése . . . . .                               | 21        |
| 3.3.      | A böngészős kliens . . . . .                                      | 21        |
| 3.3.1.    | Haskell futtatása a böngészőben . . . . .                         | 21        |
| 3.3.2.    | JavaScript kommunikációs interfész . . . . .                      | 22        |
| 3.3.3.    | Felépítés . . . . .                                               | 22        |
| 3.4.      | A megoldó, mint kliens . . . . .                                  | 24        |
| 3.4.1.    | Kommunikáció . . . . .                                            | 24        |
| <b>4.</b> | <b>Összefoglaló</b>                                               | <b>25</b> |
| 4.1.      | További fejlesztési lehetőségek . . . . .                         | 25        |
| 4.1.1.    | Megoldó . . . . .                                                 | 26        |
| 4.1.2.    | Webes alkalmazás . . . . .                                        | 26        |
| <b>A.</b> | <b>Algoritmusok</b>                                               | <b>27</b> |
| A.1.      | Szabadsági fokok feltérképezése . . . . .                         | 27        |
| <b>B.</b> | <b>Tesztesetek</b>                                                | <b>28</b> |

# 1. fejezet

## Bevezető

### 1.1. Inspiráció: Sketchpad, SolveSpace

A mérnöki tervezőprogramok (Computer-aided Design programs, innentől: CAD) a számítástechnika korai szakaszától kezdve mind az automatizált számítás, mind az ember-gép interakció határait feszegették. Ivan Sutherland 1963-as doktori tézisének témája, a *Sketchpad*, az első CAD programnak tekinthető. Ebből a programból eredeztethető a GUI (Graphical User Interface) koncepciója is. [19]

A hasonlóan fejlett CAD programok széleskörű elterjedéséig hosszú ideig kellett várni, például az ipari standardként befutott CATIA és AutoCAD/AutoDesk programok első verziói 1982-ben jelentek meg. Ezek a szoftverek jellemzően zárt forráskódúak és magas áron licencelhetők.

A SolveSpace [18] és a FreeCAD [8] nyílt forráskódú, ingyenesen elérhető CAD-szoftverek, amelyek, bár szűkebb eszköztárral rendelkeznek, mint az AutoCAD, akár ipari környezetben is sikeresen használhatók.

Ebben a munkában egy olyan programot mutatunk be, amely hasonló, de korlátozottabb síkbeli modellező funkcionalitással rendelkezik, mint a fent említett eszközök. Egy nyílt forráskódú CAD-szoftverekben ritkán látott aspektus, hogy az implementált program lehetővé teszi a tervek több felhasználó általi egyidejű szerkesztését. A forráskód elérhető GitLabon.<sup>1</sup>

---

<sup>1</sup><https://gitlab.com/dobaipatakyattila/draw-constrained>

## 1.2. A dolgozat részei

A dolgozat két részben mutatja be a CAD-szoftveren végzett munkát:

1. A matematikai megoldókról szóló fejezetben szerepel:

- A megoldandó mértani problémák egy megfogalmazása és lehetséges reprezentációja.
- Két, a megoldásra használható iteratív algoritmus, és teljesítményük összehasonlítása.
- A hibafüggvény differenciálszámítására használható módszerek, és teljesítményük összehasonlítása.
- A túl kevés és túl sok megkötéssel rendelkező rendszerek problémája, és egy lehetséges megoldás a magas szabadsági fokok problémájára.
- A szakirodalomban fellelhető alternatív megoldások a megkötések kielégítésére.

2. A szerkesztésért és megjelenítésért felelős webes alkalmazásról szóló fejezetben szerepel:

- A kliens-szerver architektúrára épülő alkalmazás működése.
- A kommunikációs mechanizmus, ami a Haskellben írt programok között az adatok szinkronizálását biztosítja.
- A versenyhelyzetek elkerülésének itt használt módja.
- A Haskellben implementált böngészős kliensalkalmazás.

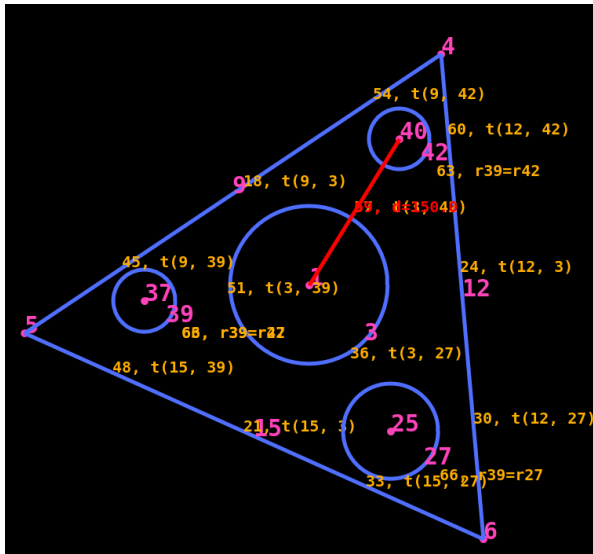
## 1.3. Bevezető példa

Tekintsünk egy bevezető példát a tárgyalt problémára. A felhasználó a grafikus webes felületen keresztül tetszőleges sorrendben beviszi (berajzolja) a következő adatokat:

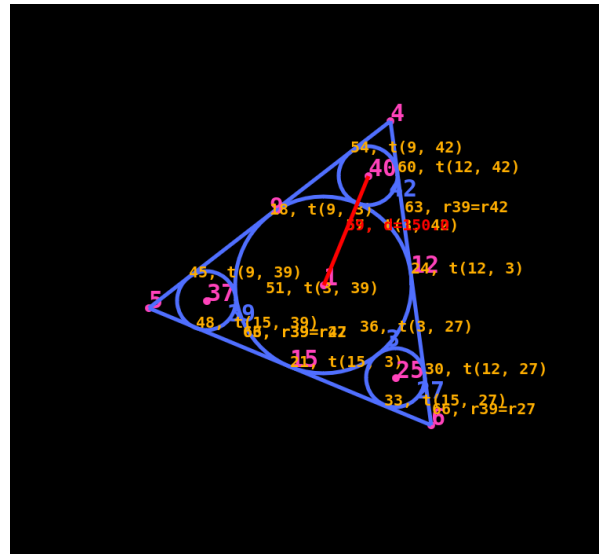
- *Elemeket:* 7 darab pontot; Szakaszokat a 4., 5. és 6. pontok közt; 4 darab kört, az 1., 25., 37., és 40. pontokat választva középpontként;
- *Megkötéseket:* Az 1. és 40. pontok közt legyen konstans 150 egységnyi távolság; A 9. szakasz legyen rendre a 3., 39., 42. körök érintője; A 12. szakasz legyen rendre a 3., 27., 42. körök érintője; A 15. szakasz legyen rendre a 3., 27., 39. körök érintője; A 3.

kör rendre érintkezzen a 27., 39., 42. körökkel; A 39. és 42., illetve a 27. és 39. körök sugara legyen egyenlő;

Ha a mértani megoldó nem aktív, mivel a bemenet csak hozzávetőlegesen pontos, az elemek állapota az 1.1a ábrán láthatóhoz lesz hasonló. Ha a mértani megoldót aktiváljuk az elemek betáplálása után, vagy a teljes folyamat alatt aktív, a 1.1b ábrához hasonló konfigurációhoz jutunk.



(a) A bemenet a megkötések betartatása nélkül



(b) A megoldó által meghatározott új konfiguráció

1.1. ábra. Példa a tárgyalt mértani problémákra. Az elemek azonosítói rózsaszínnel, a megkötések narancs és vörös színnel jelölve.

## 2. fejezet

# A megoldó

### 2.1. Bevezetés

Kétdimenziós térben létező mértani elemekről teszünk állításokat, „megkötéseket” fogalmazunk meg rájuk nézve. Jelöljük a mértani elemek halmazát  $V$ -vel, a rájuk vonatkozó megkötések halmazát pedig  $C$ -vel. A célunk úgy meghatározni  $V$  elemeinek paramétereit, hogy  $C$  lehető legtöbb megkötése teljesüljön.

Ideális lenne minél kevesebb típusú mértani elem definiálása, úgy, hogy ezek lefedjék a mérnöki felhasználók igényeit. A CAD-programokban leggyakrabban használt elemek: pontok, egyenesek / szakaszok, körök / körívek, Bezier-görbék.

Ezekre az elemekre például a következő típusú megkötések értelmezhetőek: (Euklideszi) távolság, fix pozíció, fix sugár, két szakasz által bezárt szög, három pont által bezárt szög, függőlegesség, vízszinteség, párhuzamosság, merőlegesség, érintő.

A legrégebbi megoldási irányzat a CAD területen az, hogy ezeket a megkötéseket hibafüggvényekként definiáljuk és a négyzetes hibák összegének minimalizálását tekintjük a megoldásnak. [19]

Mi is ezt a reprezentációt választjuk általánossága miatt. Széles matematikai eszköztára miatt a megoldót és teszteket a Julia [3] nyelvben implementáltuk.

### 2.2. Derivált-alapú iteratív módszerek

Ha a megkötéseket hibafüggvényekként fogalmazzuk meg, a mértani probléma leírható egy optimalizációs feladatként, ahol az

$$f(x) = \sum_i^n f_i(x), f_i \in C$$

függvény minimumát keressük. Jelöljük az optimumot  $x^*$ -al,  $f(x^*)$ -ot pedig  $p^*$ -al.

A tárgyalt algoritmusok azzal a feltételezéssel élnek, hogy az  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  függvény kétszeresen (a gradiens módszer esetén egyszeresen) differenciálható. Belátható, hogy ez teljesül, ha a  $C$  megkötéshalmaz  $f_i$  függvényei is kétszeresen differenciálhatók. Amikor a függvény nem konvex, a talált megoldás nem feltétlenül lesz a globális optimum.

A megoldás egy szükséges és elégséges feltétele, hogy

$$\nabla f(x^*) = 0$$

legyen. Néhány sajátos esetben az egyenlet megoldható analitikus úton, de a legáltalánosabb megoldási módszer az iteratív algoritmusoké. Az ereszkedésre alapuló iteratív algoritmusok  $x^{(0)}, x^{(1)}, \dots$  közelítő megoldások sorozatát számítják ki, ahol  $f(x^{(k)}) \rightarrow p^*$ , amikor  $k \rightarrow \infty$ . Ezeknél az algoritmusoknál előfeltétel egy  $x^{(0)}$  kezdeti pozíció ismerete. [5]

Egy ilyen kezdeti pozíció a tárgyalt CAD-programban mindig adott lesz, mivel a felhasználó a tervezés során hozzávetőlegesen elhelyezi a mértani elemeket, mielőtt megkötéseket fogalmaz meg rájuk.

A ereszkedésre alapuló iteratív algoritmusok általános alakja a következő [5]:

---

### 1. Algorithm Ereszkedésre alapuló iteratív algoritmus

---

- 1: **Bemenet:** A kezdőpont  $x \in \text{dom } f$ .
  - 2: **repeat**
  - 3:   Meválasztjuk a  $\Delta x$  keresési irányt.
  - 4:   *Vonalkeresés.* Megválasztjuk a  $t$  lépéshosszt.
  - 5:   Frissítés.  $x := x + t\Delta x$ .
  - 6: **until** A leállási feltétel teljesül.
- 

### 2.2.1. Keresési irány megválasztása

A keresési irány megválasztására két módszert vizsgálunk:

- *Gradiens módszer:*  $\Delta x := -\nabla f(x)$ , ahol  $\nabla f$  az  $f$  függvény gradiense.

- *Newton-módszer*:  $\Delta x := -(\nabla^2 f(x))^{-1} \nabla f(x)$ , ahol  $\nabla^2 f$  az  $f$  függvény Hesse-féle mátrixa.

### 2.2.2. Vonalkeresés

A vonalkeresés célja, hogy meghatározzon egy  $t$  lépéshosszt, amelyre  $f(x + t\Delta x)$  a lehető legkisebb. A következő algoritmus egy kellően jó közelítést ad [5]:

---

#### 2. Algorithm Vonalkeresés

---

- 1: **Bemenet**: A keresési irány  $\Delta x$ ,  $\alpha \in (0, 0.5)$ ,  $\beta \in (0, 1)$ .
  - 2:  $t := 1$ .
  - 3: **while**  $f(x + t\Delta x) > f(x) + \alpha t \nabla f(x)^T \Delta x$  **do**
  - 4:      $t := \beta t$ .
  - 5: **end while**
- 

### 2.2.3. Leállási feltétel

Mindkét módszer esetén elégséges feltétel a leálláshoz, ha  $\nabla f(x) < \epsilon$ , ahol  $\epsilon$  egy tetszőleges hibahatár. A tárgyalt megoldó egy heurisztikus, korai leállási feltételt is használ arra az esetre, ha a megoldás túlságosan eltér a kezdeti konfigurációtól. A legtöbb ilyen esetben a megoldó olyan megoldáshoz konvergál, ami a felhasználó szempontjából nem elfogadható.

### 2.2.4. Összehasonlítás

A 2.1 táblázatban a Newton- és gradiens módszerek futási idejének összehasonlítása szerepel, vonalkereséssel és vonalkeresés nélkül. A gradiens módszer a legtöbb esetben nem konvergált vonalkeresés nélkül, ezért ez az eset nem szerepel az összehasonlításban. A szürke szín azokat az eseteket jelzi, amikor a végeredmény hibája nem volt közel 0. A tesztesetek részletes leírása a B függelékben található.

A gradiens módszer bizonyos esetekben jelentősen lassabbnak bizonyult a Newton-módszernél. A Newton-módszer esetén a vonalkeresés nem hozott javulást.

| Teszt eset                | Változók száma | Gradiens módszer vonalkereséssel | Newton-módszer | Newton-módszer vonalkereséssel |
|---------------------------|----------------|----------------------------------|----------------|--------------------------------|
| eqTriangle                | 6              | 1.084                            | 0.757          | 0.755                          |
| hexGrid                   | 38             | 17.649                           | 5.963          | 5.942                          |
| hexGrid2                  | 48             | 16.197                           | 11.176         | 11.231                         |
| inscribedCircle           | 9              | 0.261                            | 0.236          | 0.245                          |
| inscribedCircle2          | 18             | 7.913                            | 1.269          | 1.338                          |
| inscribedWithDistUnsolved | 18             | 122.507                          | 1.668          | 1.755                          |
| parallelogram             | 20             | 144.607                          | 12.49          | 35.053                         |

2.1. táblázat. Iteratív algoritmusok átlagos futási idejének összehasonlítása. Az időtartamok *ms*-ban értendők.

## 2.3. Ábrázolásmód

Azt, hogy az elemeket és megkötéseket hogyan érdemes reprezentálni, elsősorban az szabja meg, hogy milyen módszerrel oldjuk meg a megkötéseket. Mivel iteratív, derivált-alapú algoritmusokat használunk, a megkötések hibafüggvények formájában kerülnek rögzítésre.

A fejezet elején felsorolt gyakori megkötéseknek és elemeknek csak egy részhalmazát implementáltuk. Az alábbiakban ezek reprezentációi következnek.

### 2.3.1. Elemi változók

Elemi változókként hivatkozunk a mértani elemek valós paramétereire. Például egy pont esetében ezek az  $x$  és  $y$  koordinátái. Egy elemi változó több elem direkt- vagy indirekt részét is képezheti. Az iteratív algoritmusokkal az elemi változóknak keresünk olyan értékeket, amelyek eleget tesznek a megkötéseknek. Ebből következik, hogy a megkötések is az elemi változók függvényében fogalmazhatók meg.

### 2.3.2. Elemek

A pontok és szakaszok ábrázolására két dimenzióban a legkézenfekvőbb módszert választjuk:

- pont:  $(x, y)$ , ahol  $x$  és  $y$  elemi változók.

- szakasz:  $(p_1, p_2)$ , ahol  $p_1$  és  $p_2$  pontok.

A körök esete kevésbé magától értetődő:

- Egy kör meghatározható három rajta elhelyezkedő ponttal. Ez a reprezentáció akkor lenne előnyös, ha végtelen, vagy közel végtelen sugarú köröket is szeretnénk ábrázolni; a másik két tárgyalt módszer ezt nem teszi lehetővé. Viszont a sugárra vonatkozó megkötések ábrázolását és megoldását bonyolultabbá tenné.

Egy kör ábrázolása 6 darab elemi változót igényelne.

- Egy másik lehetőség a kör egy középponttal és egy rajta elhelyezkedő ponttal való ábrázolása. Ennek a módszernek az előnye, hogy továbbra sem kell bevezetni a sugárra vonatkozó elemi változót, kifejezhető a már definiált pont típus függvényében. (Ez implementációs szempontból egyszerűsít.) Viszont hátrány, hogy a sugárra vonatkozó megkötések továbbra is bonyolultak: a sugarat a két pont közti távolságként lehet kifejezni.

Egy kör ábrázolása 4 darab elemi változót igényelne.

- A középpontból és sugárból álló reprezentáció lehetővé teszi, hogy a megkötésekben direkt módon szerepeljen a sugár.

Egy kör ábrázolásához elegendő 3 darab elemi változó.

A gyors megoldhatóság szempontjából a lehető legegyszerűbb formában célszerű megfogalmazni a megkötéseket, ezért a körök esetén a sugár hosszát explicit módon tároló reprezentációt választjuk:

- kör:  $(p, r)$ , ahol  $p$  pont és  $r$  elemi változó.

### 2.3.3. Megkötések

Mivel a megkötéseknek kétszeresen deriválhatónak kell lenniük, négyzetes hibafüggvényekként fogalmazzuk meg őket. Egy megkötés egyidejűleg több hibafüggvény létezését is magával vonhatja. Az implementált megkötések a 2.2 táblázatban szerepelnek.

| Megkötés      | Résztvevő<br>elemek | Hibafüggvény                                                                                                             |
|---------------|---------------------|--------------------------------------------------------------------------------------------------------------------------|
| Fix pozíció   | 1 pont              | $(x_{p1} - xVal)^2, (y_{p1} - yVal)^2$                                                                                   |
| Fix Távolság  | 2 pont              | $(x_{p1} - x_{p2})^2 + (y_{p1} - y_{p2})^2 - d^2$                                                                        |
| Fix Sugár     | 1 kör               | $(r_{c1} - rVal)^2$                                                                                                      |
| Egyenlő Hossz | 2 szakasz           | $((x_{p1} - x_{p2})^2 + (y_{p1} - y_{p2})^2 - ((x_{p3} - x_{p4})^2 + (y_{p3} - y_{p4})^2))^2$                            |
| Egyenlő Sugár | 2 kör               | $(r_{c1} - r_{c2})^2$                                                                                                    |
| Érintő        | 2 kör               | $(\sqrt{(x_{c1} - x_{c2})^2 + (y_{c1} - y_{c2})^2} - (r_{c1} + r_{c2}))^2$                                               |
| Érintő        | 1 kör,<br>1 szakasz | $(\frac{ (x_{p2}-x_{p1})(y_c-y_{p1})-(x_c-x_{p1})(y_{p2}-y_{p1}) }{\sqrt{(x_{p2}-x_{p1})^2+(y_{p2}-y_{p1})^2}} - r_c)^2$ |
| Szög          | 3 pont              | $( atan2(y2 - y1, x2 - x1) - atan2(y3 - y2, x3 - x2)  - \alpha)^2$                                                       |
| Szög          | 2 szakasz           | $( atan2(y2 - y1, x2 - x1) - atan2(y4 - y3, x4 - x3)  - \alpha)^2$                                                       |
| Jelölés       | 1 pont              | —                                                                                                                        |

2.2. táblázat. Az implementált megkötések és hibafüggvényeik.

### Jelölési megkötés

A *jelölés* egy speciális megkötés, ami a jelölt elem elemi változóit konstansnak nyilvánítja. A jelölt elemi változók az összes többi megkötés által generált egyenletben is konstansként viselkednek a deriváltak kiszámítása során.

### Megkötések szögekre

A szögekre irányuló megkötéseknél a két szakasz (vagy három pont) által bezárt szöget a résztvevő szakaszok (vagy pontpárok)  $X$ -tengellyel bezárt szögeinek különbségeként határozzuk meg. Az  $X$ -tengellyel bezárt szögeket a 2 paraméterű arkusztangens (innenről:  $atan2$ ) függvénnyel számítjuk ki.

Az  $atan2$  előnye más inverz szögfüggvényekhez képest, hogy az értékkészlete  $(-\pi, \pi]$ , a szokásos  $(-\frac{\pi}{2}, \frac{\pi}{2}]$ -vel vagy  $(0, \pi]$ -vel szemben. Ez azt jelenti, hogy egyértelműen meghatározza

a  $(-180^\circ, 180^\circ]$  intervallumbeli szögeket.

Az  $\text{atan2}$  hátránya, hogy nagy  $\frac{\Delta y}{\Delta x}$  értékekre a derivált kicsi. Ez azt eredményezi, hogy lassú lesz a konvergencia, ha például párhuzamossági megkötést teszünk két közel függőleges szakaszra. Egy lehetséges megoldás, hogy a párhuzamossági és merőlegességi megkötésekre sajátos, hatékonyabb hibafüggvényeket adjunk. Újabb hátrány, hogy az  $\text{atan2}$  nem deriválható a  $(0, 0)$  pontban, illetve nem deriválható kétszeresen, ha  $x = 0$ . Az ilyen helyzetek elkerülése érdekében, amikor  $\Delta x$  kisebb, mint egy tetszőlegesen kicsi  $\epsilon$ ,  $\epsilon$ -t használjuk a függvényben  $\Delta x$  helyett.

Egy másik lehetőség a koszinusztétel alapján megfogalmazni a megkötéseket.

### 2.3.4. Kapcsolat elemek és megkötések közt

Az elemek és megkötések között több-a-többhöz kapcsolat áll fenn. Ebből kifolyólag minden elem esetén tároljuk a rá vonatkozó megkötések indexeit (ez egy fordított függőségi lista: azt mutatja, hogy milyen megkötések létezése függ az elem létezésétől). Egy elem definíciójában szerepelhetnek más elemek is (például egy szakaszt két pont alkot), ezért minden elemre vonatkozóan egy újabb fordított függőségi listában tároljuk, hogy milyen más elemek létezése függ tőle.

Amennyiben egy elem törlésre kerül, a törlés tovább propagálódik: minden tőle függő elemnek és megkötésnek is törlődnie kell.

## 2.4. A hibafüggvény differenciálszámítása

A tárgyalt numerikus módszerek feltételezik az optimalizálni kívánt függvény első- és másodrendű parciális deriváltjainak ismeretét. Mivel a teljes hibafüggvény a megkötések hibafüggvényeinek összegeként íródik fel, a parciális deriváltjai is felírhatók az összetevő hibafüggvények parciális deriváltjainak összegeként.

### 2.4.1. Parciális deriváltak explicit megadása

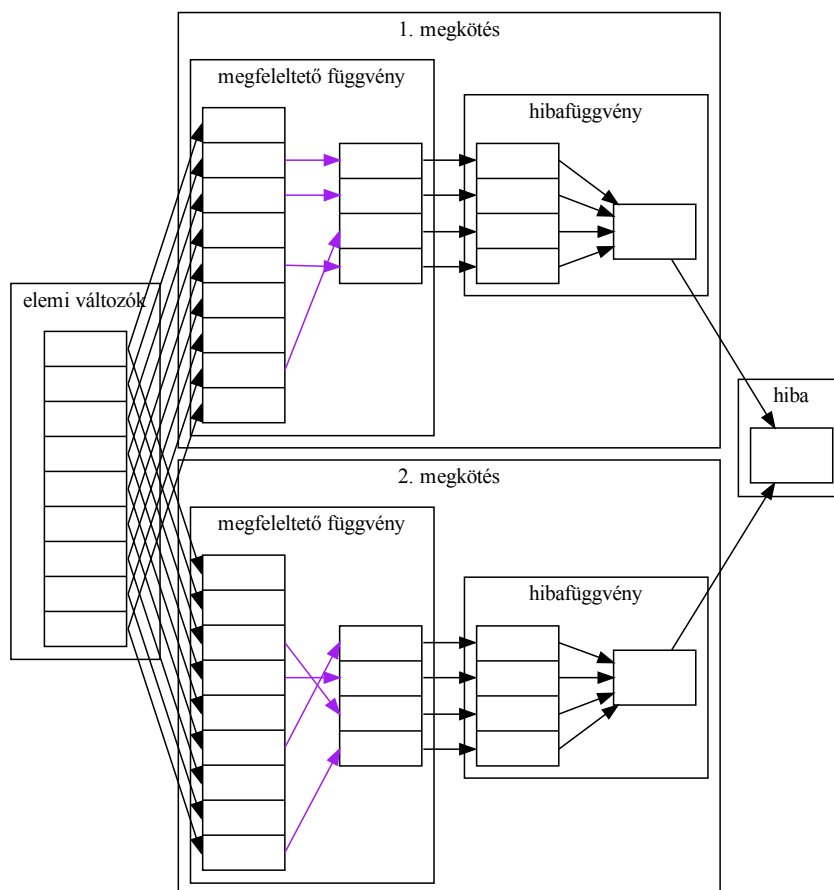
A legegyszerűbb megközelítés, hogy minden megkötéstípus hibafüggvényének explicit módon tároljuk az első- és másodrendű parciális deriváltjait. Ez nehezen bővíthetővé és nehezen változtathatóvá teszi a megkötéskészletet; a nyolcváltozós „egyenlő hossz” megkötés

definíciójához például ki kellene számolnunk és le kellene írunk 8 darab elsőrendű és 64 darab másodrendű parciális deriváltat.

## 2.4.2. Szimbolikus differenciálás

Egy kevesebb manuális munkát igénylő megközelítés, hogy a hibafüggvényeket szimbolikus kifejezések formájában írjuk le, és egy szimbolikus algebrai rendszer segítségével generáljuk a nullad-, első- és másodrendű deriváltjaikat. Erre a célra a *Symbolics.jl* [9] Julia könyvtárat használtuk.

## 2.4.3. Lokális és globális változók közti megfeleltetés



2.1. ábra. Megkötések megfeleltetésekkel és általános hibafüggvényekkel

Mivel a függvények megkötéstípusonként generálódnak vagy adottak explicit módon, szükség van egy mechanizmusra, amivel meghatározható, hogy egy konkrét megkötés esetén a függvényt melyik elemi változókra kell alkalmazni. Erre a célra minden konkrét megkötéshez hozzárendelünk egy saját függvényt, ami az összes elemi változó és a megkötés által igényelt változók közt valósít meg egy megfeleltetést. Egy példa ilyen megfeleltetésre a 2.1 ábrán látható.

#### 2.4.4. Algoritmikus differenciálás

Egy másik lehetőség az algoritmikus vagy automatikus differenciálás használata. Ez a technika a láncszabály alkalmazására alapszik: mivel egy program végrehajtása redukálható elemi műveletek végrehajtásainak sorozatává, a láncszabály segítségével meghatározható a program deriváltja, ha ismertek a benne résztvevő elemi aritmetikai műveletek és elemi függvények deriváltjai. Megkülönböztetjük az *előrehaladó* (*forward*) és a *visszafejtő* (*reverse*) módot, az szerint, hogy a láncszabály rekurzív alkalmazásából felírható műveleti gráf elemei milyen sorrendben értékelődnek ki.

- Az előrehaladó (*forward*) mód esetén a műveleti gráfot a differenciálandó függvény minden bemeneti változójára nézve be kell járni. Ez a mód olyan esetekben előnyös, amikor a bemenetek száma kisebb, mint a kimenetek száma.[1][16]

Az előrehaladó mód tesztelésére a *ForwardDiff* [16] Julia könyvtárat használtuk.

- Visszafejtő (*Reverse*) mód esetén a műveleti gráfot egyszer kell bejárni előre haladva, majd minden kimeneti változóra nézve fordított irányba. Ez a mód olyan felhasználási esetekben előnyös, amikor a kimenetek száma kisebb, mint a bemenetek száma, például jellemzően gépi tanulásban.[1][11]

A visszafejtő mód esetén a *Zygote* [11] Julia könyvtárat használtuk.

algoritmikus differenciálás egy előnye a szimbolikus módszerrel szemben, hogy nem csak matematikai értelemben vett függvényekre alkalmazható; ciklusokat, elágazásokat és változókat tartalmazó függvények deriváltjai is kiszámíthatóak vele. [1]

Ez lehetővé teszi a korábban leírt *jelölés* megkötés implementációját: az összes megkötés hibafüggvényét olyan módon generáljuk, hogy az elemi változók a differenciálás szempontjából konstansak minősüljenek, ha a hibafüggvény kiértékelésekor *jelölés* alatt állnak.

### 2.4.5. A módszerek sebességének összehasonlítása

Az előrehaladó- és visszafejtő módú algoritmikus, illetve a szimbolikus differenciálásra alapuló módszerek összehasonlítása tesztesetekre a 2.3 táblázatban látható. A probléma természetéből adódóan (a bemeneti- és kimeneti paraméterek száma megegyezik) a visszafejtő mód jelentősen lassabbnak bizonyult az előrehaladó módnál. A szimbolikus módszer teljesítménye a két algoritmikus módszeré között helyezkedik el. A tesztesetek részletes leírása a B függelékben található.

| Teszteset                 | Változók száma | Előrehaladó mód | Visszafejtő mód | Szimbolikus módszer |
|---------------------------|----------------|-----------------|-----------------|---------------------|
| inscribedCircle           | 9              | 0.24            | 5.492           | 2.298               |
| inscribedCircle2          | 18             | 1.25            | 34.403          | 9.132               |
| inscribedWithDistUnsolved | 18             | 1.602           | 45.655          | 11.596              |
| hexGrid                   | 38             | 5.98            | 79.485          | 20.496              |
| hexGrid2                  | 48             | 11.162          | 95.34           | 23.965              |
| parallelogram             | 20             | 13.121          | 322.414         | 94.547              |

2.3. táblázat. A megoldás sebessége különböző differenciálási módszerekkel (Newton-módszert használva). Az időtartamok *ms*-ban értendők.

## 2.5. Szabadsági fokok

A megkötésekből felírható egy

$$\begin{aligned}f_1(x_1, \dots, x_m) &= 0 \\&\vdots \\f_n(x_1, \dots, x_m) &= 0\end{aligned}$$

formájú nemlineáris egyenletrendszer, ahol a mértani probléma elemi változóinak száma  $m$ , a megkötések száma pedig  $n$ .

Amennyiben az egyenletrendszerben több változó van, mint ismeretlen, az egyenletrendszer határozatlan, és a megoldások száma lehet végtelen, akár több változóban is. Nevezzük a változók száma és a megkötésekből származó, egymástól lineárisan független egyenletek száma

közti különbséget *szabadsági fok*nak. [14]

A megoldó jelenlegi változata nem szűri ki a redundáns (egymástól lineárisan függő) megkötéseket, így a szabadsági fok meghatározása nem lesz helyes redundáns megkötésekre.

### 2.5.1. Szabadsági fokok feltérképezése

Előfordulhat, hogy az elemi változók egy része egyértelműen meghatározott a megkötések által, miközben egy másik részük 0-nál magasabb szabadsági fokkal rendelkezik.

A különböző szabadsági fokkal rendelkező komponensek feltérképezésére az A.1 függelékben leírt algoritmust használjuk.

### 2.5.2. Szabad változók ideiglenes rögzítése

A korábban tárgyalt iteratív algoritmusok akkor is találnak megoldást, ha a rendszer szabadsági foka nem 0. Viszont mivel mindig csak 1 darab megoldást találnak, nincs garancia arra, hogy a potenciálisan végtelen megoldásból ez az, amelyet a felhasználó el szeretne volna érni.

Egy módszer arra, hogy egy nem teljesen kötött rendszerben több helyes megoldást is megtaláljunk iteratív algoritmusokkal, az, ha a 0-nál nagyobb szabadsági fokú komponensekből rendre konstansnak tekintünk változókat, úgy, hogy a komponens szabadsági foka továbbra se csökkenjen 0 alá. Legyen  $Locks(K)$  a  $K$  komponenshalmazra így kapható összes rögzítési kombináció:

$$Locks(K) = \mathcal{P}_{dof(K_1)+1}(K_1) \times \mathcal{P}_{dof(K_2)+1}(K_2) \times \dots \times \mathcal{P}_{dof(K_n)+1}(K_n)$$

ahol  $\mathcal{P}_\kappa(K_i)$  jelöli  $K_i$  azon részhalmazait, amelyek számossága kisebb, mint  $\kappa$  (vagyis  $\mathcal{P}_\kappa(K_i) = \{U \subseteq K_i : |U| < \kappa\}$ ) és  $dof(K_i)$  a  $K_i$  komponens szabadsági foka. A definíció alapján belátható, hogy a lehetséges rögzítések száma kombinatoriális:

$$\binom{|K_1|}{dof(K_1)} \cdot \binom{|K_2|}{dof(K_2)} \cdot \dots \cdot \binom{|K_n|}{dof(K_n)}.$$

Az összes lehetséges rögzítésnek viszont általában csak a töredékét kell kiértékelni, mert egy megoldás jónak tekinthető, ha egy választott hibametrika szerint hibahatáron belül van. A rögzítések ideiglenes *jelölés* megkötésekként értelmeződnek a megoldó által.

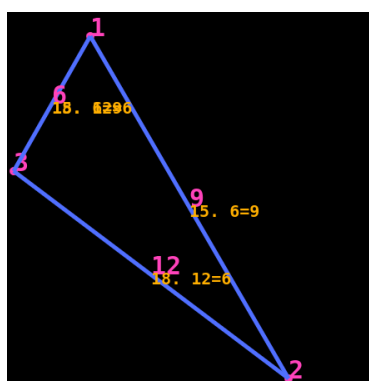
## Az eltérés mértéke, mint hibametrika

Amennyiben több helyes lehetséges megoldás is adott, választani kell közülük. Feltételezhetjük, hogy a felhasználó által elvárt megoldás az, amelyik a legkevésbé tér el az általa felvázolt kezdeti konfigurációtól, úgy, hogy közben a megkötések szempontjából helyes. Ennek számszerűsítésére az eltérések abszolút értékeinek átlagát vettük alapul.

### Példa

Például, ha a felhasználó rajzol egy háromszöget, és két oldalegyenlőségi megkötést tesz rá (2.2a ábra), a legkézenfekvőbb (és az iteratív algoritmus által leggyakrabban megtalált) megoldás az, amikor az egyenlő oldalú háromszög minden oldala 0 hosszúságúra csökken, vagyis mindhárom csúcsa azonos pozícióba kerül (2.2b ábra).

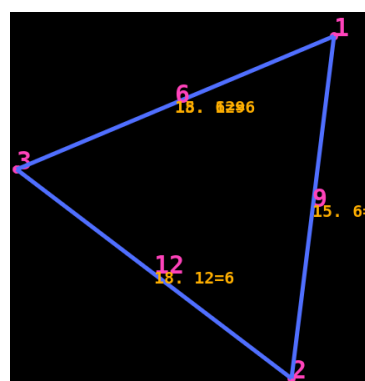
Egy, a felhasználó által elfogadhatóbb megoldás látható a 2.2c ábrán. A 6 darab elemi változóra 2 darab megkötés volt adott, a rendszer szabadsági foka 4. Egy lehetséges rögzítés a 2. és 3. pontok 2-2 elemi változójának konstansnak tekintése. Ebben az esetben az 1. pont pozicionálásával található megoldás.



(a) Kezdőállapot, megkötések betartatása nélkül



(b) Lehetséges megoldás 204.65 eltérési mértékkel



(c) Lehetséges megoldás 48.13 eltérési mértékkel

2.2. ábra. Egyenlő oldalú háromszög különböző eltérési mértékkel rendelkező megoldásokkal.

### Rögzítések elkerülése az eltérés büntetésével

A rögzítések elkerülésével a megoldás jelentősen hatékonyabb lenne. Ideális esetben az iteratív algoritmus egyetlen futására lenne szükség. Egy általunk vizsgált lehetőség rögzítések nélküli megoldásra a hibafüggvény módosítása volt. Ebben az esetben a hibafüggvénybe belefoglaltuk a változók kezdeti értéktől való eltéréseinek négyzetösszegét, egy alacsony együtthatóval.

Ez a megközelítés nem bizonyult működőképesnek: a megoldások általában nagy eltérési mértékű konfigurációkhoz konvergáltak, az eredeti megkötések pedig gyakran sérültek.

### **2.5.3. Megoldhatatlan esetek kiszűrése**

Előfordulhat, hogy a felhasználó ellentmondó megkötéseket ad meg. Ez akkor is megtörténhet, ha a rendszer szabadsági foka pozitív vagy 0, de a leggyakrabban akkor következik be, ha a szabadsági fok negatív, vagyis több megkötés adott, mint elemi változó.

A megoldó jelenlegi verziója nem jelzi a megoldhatatlan eseteket. Ennek ellenére ismertetjük az egyéb nyílt forráskódú megoldók által alkalmazott módszereket.

#### **Negatív szabadsági fok esetén**

Amikor a rendszer szabadsági foka negatív, legjobb esetben is redundánsak a megkötések. Ezért sok CAD-program megoldója ilyen esetben jelzi a redundanciát és hibásnak nyilvánítja a tervet (például [18], [8]).

#### **Nemnegatív szabadsági fok esetén**

A legtöbb CAD-program felépítéséből kifolyólag mindig egyesével kerülnek új megkötések a rendszerbe. Feltételezzük, hogy a rendszer mindig egy megoldható állapotból indul. Ebben az esetben, ha a megoldó nem konvergál egy (közel-) 0 hibájú megoldáshoz, kijelenthető, hogy a legutóbb hozzáadott megkötés tette megoldhatatlanná a rendszert, és legalább egy darab korábban megadott megkötéssel van ellentmondásban. Az ellentmondó megkötések megtalálásához elegendő kiértékelni a hibás megoldás által ajánlott értékekre a megkötések egyenkénti hibafüggvényeit. Ha egy megkötés valamelyik hibafüggvényének értéke nem (közel-) 0, akkor ellentmondásban van a legutóbb megadott megkötéssel. (Feltételezve, hogy a megoldó minden helyes esetben konvergálni fog egy közel 0 hibájú megoldáshoz.) [18], [12]

## **2.6. Egyéb megoldók és mértani reprezentációk az irodalomban**

A numerikus módszerek feltétele, hogy adott legyen a  $V$  elemhalmaz egy kezdeti  $x^{(0)}$  konfigurációja, amelynek viszonylag közel kell állnia az elvárt megoldáshoz. Léteznek olyan

megoldási módszerek, amiket nem korlátoz ez az előfeltétel.

### **2.6.1. Szimbolikus megoldók**

A szimbolikus megoldók a megkötéseket egy egyenletrendszerként írják fel, amit szimbolikus algebrai úton oldanak meg. A leggyakrabban használt szimbolikus algebrai egyenletrendszer-megoldó algoritmusok idő-komplexitása exponenciális. [4] [13]

### **2.6.2. Szabadsági fok-analízis alapú megoldók**

A szabadsági fok-analízis alapú megoldók az adott elemekre inkrementálisan elégtik ki a megkötéseket, minden lépésben csökkentve a rendszer szabadsági fokát. Megkülönböztetődnek a forgatási, skálázási és transzlációs szabadsági fokok. A megkötések kielégítésére tett lépések szimbolikus érvelési algoritmusok által következtetődnek ki, polinomiális idő-komplexitáshoz vezetve. [13]

### **2.6.3. Logikai és szabály-alapú megoldók**

A megoldók ezen osztálya arra alapszik, hogy a mérnöki rajzokban a legtöbb részlet kimérhető vonalzóval, körzővel és szögmérővel. A mértani elemek és megkötések szekvenciálisan helyeződnek el a megoldás során, szabály- vagy gráf-behelyettesítő algoritmusok szerint. [2] [4]

## 3. fejezet

# A CAD-program architektúrája

Az itt leírt alkalmazás célja, hogy a korábban bemutatott megoldót használva egyszerre több felhasználó is szerkeszthesse vagy vizualizálhassa ugyanazt a tervet. Mivel a kezelendő adatok jól struktúráltak és a vizualizálás és interakció folyamata szabályszerű, kézenfekvő lenne deklaratív, funkcionális módon leírni ezeket.

A Glasgow Haskell Compiler segítségével az utóbbi években lehetővé vált a Haskellben írt programok böngészőben való futtatása. Ez lehetővé teszi, hogy ugyanabban a programozási nyelvben implementáljunk egy böngészős kliensalkalmazást és egy WebSocket protokollon keresztül kommunikáló szervert, így választásunk erre a nyelvre esett.

### 3.1. Felépítés és kommunikáció

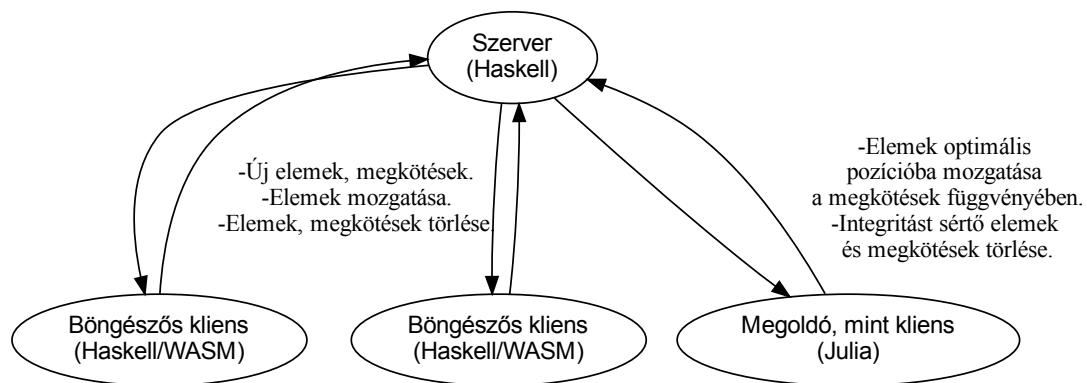
Az alkalmazás a következő részekre bontható:

- egy szerver, ami a terv állapotát szinkronizálja a kliensek között.
- egy böngészős kliens, amin keresztül a felhasználók szerkeszthetik a tervet.
- a megoldó, mint kliens, ami valós időben biztosítja a terv helyességét.

A két klienstípus szerepét, illetve a kommunikáció menetét szemlélteti a 3.1 ábra.

Az, hogy a tervek validálását és megoldását nem a szerver, hanem egy kliens végzi, modulárisabbá teszi az alkalmazást; akár egyéb nyílt forráskódú CAD-programok megoldói is viszonylag könnyen integrálhatók lennének, mint kliensek.

Mivel közel valós idejű és gyakori üzenetváltásokra van szükségünk, a WebSocket protokollt használjuk.



3.1. ábra. Az egymással kommunikáló programok, és a különböző kliensek által küldött üzenetek típusai.

### 3.1.1. Közös definíciók szerver és böngészős kliens között

Az azonos szerver- és kliensoldali nyelv egy előnye, hogy újrahasználhatók adattípus- és függvénydefiníciók. A megosztott adattípusok:

- A mértani elemek és megkötések, illetve a felhasználók adatai — az alapstruktúrák.
- A rendszer állapota — az alapstruktúrákat tároló hash-táblákból áll.
- A „műveletek” — a hozzáadást, módosítást vagy törlést jelképezik.
- Az üzenetek — közvetítik, hogy egy alapstruktúrával elvégezzünk egy műveletet, az állapot változását okozva.

Az alapstruktúrák esetén általános elvárás, hogy adott alapállapot és művelet esetén a művelet alapstruktúrára való végrehajtása változtassa az állapot megfelelő hash-tábláját. Ezt a funkcionalitást a következő típusosztály formájában fogalmaztuk meg:

```

1 class (Show a, Read a) => RoomStateAppliable a where
2   apply :: RoomState -> Action -> a -> (RoomState, a)
3   apply roomState act object = (setContainer updatedContainer, objWKey)
4   where
5     updatedContainer = case act of
6       Add    -> insert key objWKey container
7       Update -> insert key objWKey container
8       Remove -> delete key container
9     container = relevantContainer roomState
10    setCont    = setRelevantContainer roomState
11    key        = determineKey roomState obj
12    objWKey    = withKey obj key
  
```

```

13
14 determineKey :: RoomState -> a -> Key
15 -- A soron következő kulcs meghatározása szintén általánosan értelmezett,
16 -- de a tömörség kedvéért a definíció itt nem szerepel.
17
18 -- Az alábbi függvényeket a konkrét típusnak kell implementálnia:
19 relevantContainer    :: RoomState -> Map Key a
20 setRelevantContainer :: RoomState -> Map Key a -> RoomState
21 getKey    :: a -> Key
22 withKey :: a -> Key -> a
23

```

Ennek az általánosításnak köszönhetően, ha a későbbiekben egy új alapstruktúra típust vezetünk be, az könnyebben integrálható lesz a rendszer állapotába.

### 3.1.2. Adatok közvetítése

A kommunikáció a szerver- és kliensalkalmazások között a korábban leírt üzenet adattípus által történik. A közös típusdefiníciók egy további előnye, hogy az üzenetek mindkét Haskell-programban triviálisan szerializálhatóak és deszerializálhatóak az alapértelmezetten generált *read* és *show* függvényekkel.

A szerializált üzenetek a WebSocket protokollon keresztül kommunikálódnak. Az üzenetek méretét és feldolgozási idejét jelentősen csökkenteni lehetne bináris szerializációs és deszerializációs függvényekkel, de mivel a kommunikáció sebessége a szöveges reprezentációval is kielégítő, erre nem került sor.

Mind a kliensek, mind a szerver kizárólag beérkező üzenetek hatására módosítja a tárolt elemek és megkötések állapotát. Ez csökkenti a szerver és kliens közti inkonzisztencia lehetőségét: egyik kliens sem módosít úgy a saját állapotán, hogy az állapotmódosítás információját előbb ne vette volna tudomásul a szerver.

## 3.2. A szerver

A szerver szerepe az üzenetek közvetítése a kliensek között és a terv állapotának nyilvántartása. A beérkező (létrehozási-, módosítási-, törlési-) üzeneteket végrehajtja a saját állapotára nézve, majd továbbítja őket az összes kliensnek (beleértve az eredeti üzenet forrását). Amennyiben létrehozásra érkezik üzenet, a szerver oszt ki azonosítót az új értéknek. Viszont a terv integritása nem a szerver felelőssége: például, ha törlési utasítás érkezik egy elemre, az elemtől függő megkötések nem fognak automatikusan törlődni; ez a megoldó feladatköre.

A szerver a *websockets* [20] Haskell könyvtárra alapszik. A tárolt állapot a korábbi fejezetben leírt megosztott (elemekből, megkötésekből és felhasználói adatokból álló) állapotból, illetve az aktív WebSocket kapcsolatokra való hivatkozásokból áll.

### 3.2.1. Versenyhelyzet elkerülése

A *websockets* könyvtár működéséből adódóan minden új klienshez egy-egy saját futási szál rendelődik, amelyek blokkoló módon várják az üzeneteket. A szálaknak konkurens módon ugyanazt a globális állapotot kell módosítaniuk, ami versenyhelyzetek kialakulásához vezethet. Ennek kivédésére a GHC *MVar* típusát használjuk.

Egy *MVar*  $t$  típusú érték egy változtatható területre hivatkozik, amely vagy tartalmaz egy  $t$  típusú értéket, vagy üres. Az *MVar* típusra értelmezettek a *takeMVar* és *putMVar* műveletek, amelyek blokkolják az őket végrehajtó szálakat, ha az *MVar* nincs megfelelő állapotban (*takeMVar* esetében üres, *putMVar* esetén nem üres). [15].

A szerver globális állapotát *MVar* (*Connections*, *RoomState*) típusúként határozzuk meg, így biztosítani tudjuk, hogy egyszerre mindig csak egy szál férjen hozzá.

## 3.3. A böngészős kliens

A böngészős kliens minden bejövő üzenetre való reakciója leírható egy transzformációként, amely az üzenet és az aktuális állapot alapján meghatároz kimenő üzeneteket és egy új állapotot. A kimenő üzenetek lehetnek megjelenítési hívások, amelyek a böngészőben lokálisan értelmeződnek, illetve a szervernek címzett üzenetek, amelyek elemek vagy megkötések létrehozását, változtatását vagy törlését kérvényezik. A megjelenítés egy HTML *canvas* elemen történik.

### 3.3.1. Haskell futtatása a böngészőben

A Glasgow Haskell Compiler a 9.6-os verziójától kezdve két lehetőséget is ad a programok böngészőben való futtatására:

- JavaScriptbe való fordítás [10]
- WebAssemblybe való fordítás [17]

A JavaScriptbe fordított prototípusok jelentősen lassabbnak bizonyultak a WebAssemblybe fordított változatnál, ezért az utóbbit fejlesztettük tovább.

### 3.3.2. JavaScript kommunikációs interfész

A böngészők nem adnak direkt hozzáférést a WebAssembly Virtuális Gépnek (WebAssembly Virtual Machine) a DOM-hoz (Document Object Model), de megengedik JavaScript függvények meghívását, amik viszont már hozzáférnek a DOM-hoz. Ezért a Haskellben írt főprogram külvilággal való kommunikációja egy JavaScript interfészen keresztül valósul meg.

A kommunikációs interfész a főprogram számára bejövő üzeneteket egy várakozási sorba helyezi. A bejövő üzenetek érkehetnek a szervertől, WebSocket üzentekként, illetve a böngészőtől, mint események, amiket a felhasználó vált ki, billentyűk lenyomásával, vagy egérmozgással, kattintással.

A főprogram szempontjából kimenő üzenetek lehetnek megjelenítéssel kapcsolatos, DOM-ot manipuláló függvényhívások, vagy a szervernek szóló üzenetek.

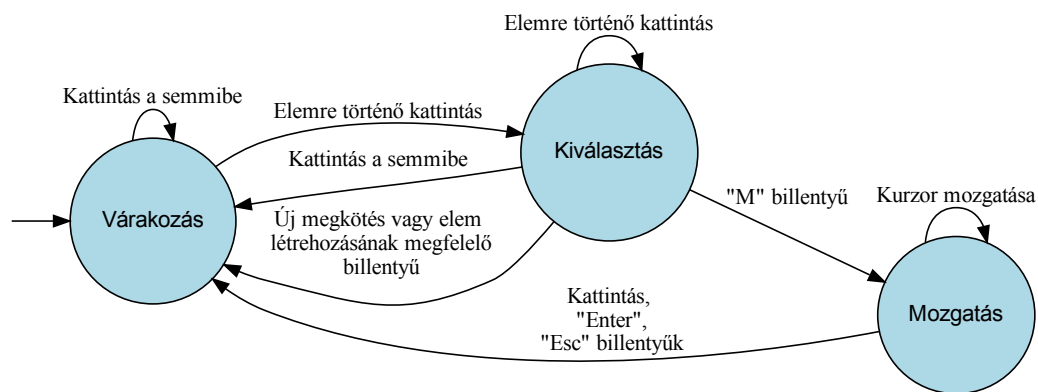
### 3.3.3. Felépítés

A program belépési pontja egy végtelen rekurzív függvény, aminek paramétere a rendszer állapota. Ennek a fő rekurciónak a során:

1. Beolvassódik egy feldolgozatlan üzenet a kommunikációs interfésztől.
2. A rendszer állapota és az üzenet függvényében meghatározódik a rendszer új állapota, és az esetleges kimeneti üzenetek.
3. A kimeneti üzenetek továbbítódnak a kommunikációs interfészhez.
4. Meghívódik a rekurzió az új állapottal.

### Beviteli módok

A 3.1.1-ben leírt megosztott állapot mellett a böngészős kliens tárol egy felhasználói bevitellel kapcsolatos állapotot is, ami lehetővé teszi az összetett beviteli funkciókat. A beviteli állapota „módokra” bontható, amik meghatározzák, hogy egy adott kontextusban hogyan



3.2. ábra. Beviteli módok véges automatája.

értelmezendők a felhasználó által billentyűzettel és egérrel generált üzenetek. (Az elképzelés a *vim* szövegszerkesztő szerkesztési módjaival analóg.)

A beviteli módok közti összefüggések leírhatók egy véges automatával, ami az 3.2 ábrán látható. A felhasználó által generált üzenetek mindegyike egy állapotváltozást von maga után ebben az automatában. Egyes átmenetek generálhatnak a szervernek szóló állapotmódosítási üzeneteket.

A jelenleg implementált beviteli módok:

- Várakozási mód — az alapállapot.
- Kiválasztási mód — lehetővé teszi egy vagy több elem kijelölését. Ebből a módból kiindulva lehet megkötéseket és összetett elemeket törölni és létrehozni.
- Mozgatási mód — az előzőleg kijelölt pontok a kurzormozgással arányosan elmozdulnak.

### Kijelölt elemek zárolása

Amikor egy felhasználó kiválaszt egy elemet, arra az elemre egy ideiglenes „jelölés” megkötés helyeződik. Ez a megkötés kettős hasznú: lehetetlenné teszi az elem kijelölését az összes többi felhasználó számára, illetve megtiltja az elem módosítását a megoldó számára. Pontok esetén a jelölést egy kör jelzi a pont körül, aminek a színe a jelölés birtokosának felhasználónevéből generálódik egy hashfüggvény által.

### 3.4. A megoldó, mint kliens

A megoldó az emberi felhasználókkal egyenrangú kliensként működik. A bejövő üzenetekre adott automatikus reakciói:

- Minden bejövő üzenet esetén frissíti a tárolt állapotát.
- Minden bejövő üzenet esetén ellenőrzi a rendszer hivatkozási integritását. Az olyan elemekre és megkötésekre, amik nem létező elemekre hivatkoznak, kibocsájt egy törlési üzenetet. (Így valósul meg a törlések propagálása.)
- Ha olyan üzenetet kap, aminek a kezdeményezője egy másik kliens, kiszámolja a mértani elemek optimális állapotát, úgy, hogy a megkötések ne sérüljenek. Ha egy elem nem a megfelelő pozícióban helyezkedik el, kibocsájt egy módosítási üzenetet.

#### 3.4.1. Kommunikáció

A matematikai megoldót a Julia nyelvben implementáltuk, ezért az üzenetekre nem használható a két Haskell-program között működő szerializációs / deszerializációs technikát. Azért, hogy ne kelljen változtatni az üzenetek formátumán, ennél a kliensnél egy PEG-t (Parsing Expression Grammar) [7] definiáltunk, ami felismeri a Haskell programok által szerializált üzeneteket. Erre a célra a *PEG.jl* [6] könyvtárat használtuk.

## 4. fejezet

# Összefoglaló

Leírtuk a CAD-programokban releváns síkbeli mértani probléma egy lehetséges ábrázolásmódját. Ismertettük az ennek megoldására használt numerikus módszerek alapjait, valamint implementáltunk egy megoldót a Julia nyelvben.

A szükséges deriváltak kiszámítására az előrehaladó módú algoritmikus differenciálási technika bizonyult a leghatékonyabbnak. Mivel a megoldó magas szabadsági fokú esetekben nem mindig a kívánt megoldáshoz konvergál, rendre rögzítjük a változókat, amíg a kiinduló konfigurációhoz megfelelően közeli megoldást nem találunk.

Implementáltunk egy CAD-stílusú webes alkalmazást Haskellben, amely segítségével több felhasználó szimultán szerkesztheti ugyanazt a tervet. A megoldó is egy kliens, amely valós időben biztosítja a terv helyességét.

Az alkalmazás és a megoldó sebessége kielégítő a 15-20 változóból álló problémák szerkesztésénél. A megoldó bizonyos esetekben instabil, a tárgyalt rögzítési algoritmus ellenére sem mindig konvergál intuitív megoldáshoz. Ilyenkor a rögzítési esetek kombinatorikus száma miatt a megoldás túl sok időt vesz igénybe.

### 4.1. További fejlesztési lehetőségek

Mind a megoldó, mind a webes alkalmazás további fejlesztésekre és optimalizációkra szorul.

### **4.1.1. Megoldó**

#### **Független részek független megoldása**

Azokban az esetekben, amikor a rendszer egymástól független komponensekre bontható, ezek a komponensek külön-külön rendszerekként kezelhetők. Ez javítaná a változók számára nézve négyzetes- és magasabb fokú polinomiális bonyolultságú algoritmusok teljesítményét, így az iteratív módszereket és a differenciáló algoritmusokét is. Valamint ilyen esetekben csökkenne a kiértékelendő rögzítések száma.

#### **Eltérési hibahatár adaptív meghatározása**

Az eltérési hibahatár teszi lehetővé, hogy pozitív szabadsági fok esetén a különböző rögzítési módok közül ne kelljen az összeset kiértékelni. Ez a hibahatár jelenleg konstans. Egy lehetséges adaptív megközelítés alapulhatna a felhasználók által legutóbb kiadott mozgatási utasítás nagyságán.

#### **Párhuzamosítás**

Azokban az esetekben, amikor a megoldó több rögzítésre is kiértékeli a rendszert, a megoldás gyorsítható párhuzamosítással, mert az egyenkénti megoldási folyamatok egymástól függetlenek.

### **4.1.2. Webes alkalmazás**

#### **Felhasználói felület**

A konstans paraméterekkel rendelkező megkötések (pontok közti kötött távolság, kötött sugár, kötött szög) esetén szükség van számértékek bevitelére a felhasználó részéről. Mivel jelenleg nincs ilyen beviteli mód, az említett megkötések hozzáadása egy előre definiált értékkel köti meg a rendszert. Egy lehetséges megoldás egy új szám-beviteli mód hozzáadása a webes kliens állapotát leíró véges automatához.

# Irodalomjegyzék

- [1] Atilim Gunes Baydin és mások. „Automatic Differentiation in Machine Learning: a Survey”. *Journal of Machine Learning Research* 18.153 (2018), 1–43. oldal. URL: <http://jmlr.org/papers/v18/17-468.html>.
- [2] Bernhard Bettig és Christoph M. Hoffmann. „Geometric Constraint Solving in Parametric Computer-Aided Design”. *Journal of Computing and Information Science in Engineering* 11.2 (2011). DOI: [10.1115/1.3593408](https://doi.org/10.1115/1.3593408).
- [3] Jeff Bezanson és mások. „Julia: A fresh approach to numerical computing”. *SIAM Review* 59.1 (2017), 65–98. oldal. DOI: [10.1137/141000671](https://doi.org/10.1137/141000671). URL: <https://epubs.siam.org/doi/10.1137/141000671>.
- [4] William Bouma és mások. „A Geometric Constraint Solver”. (1993). Bouman-ComputerAidedDesign-1995-GeometricConstraintSolver.
- [5] S. P. Boyd és L. Vandenberghe. *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2004. URL: <http://www.stanford.edu/~boyd/cvxbook.html>.
- [6] William DeBeaumont. URL: <https://github.com/wdebeaum/PEG.jl>.
- [7] Bryan Ford. „Parsing expression grammars: a recognition-based syntactic foundation”. *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL04. ACM, 2004. jan. DOI: [10.1145/964001.964011](https://doi.org/10.1145/964001.964011). URL: <http://dx.doi.org/10.1145/964001.964011>.
- [8] FreeCAD Contributors. *FreeCAD Documentation*. FreeCAD. URL: <https://www.freecadweb.org/> (elérés dátuma: 2024. ápr. 24.)
- [9] Shashi Gowda és mások. „High-performance symbolic-numerics via multiple dispatch”. *arXiv preprint arXiv:2105.03949* (2021).
- [10] Sylvain Henry és mások. *JavaScript backend merged into GHC*. 2022. URL: <https://engineering.io/2022-12-13-ghc-js-backend-merged/>.

- [11] Michael Innes. *Don't Unroll Adjoint: Differentiating SSA-Form Programs*. 2018. DOI: 10.48550/ARXIV.1810.07951. URL: <https://arxiv.org/abs/1810.07951>.
- [12] Matt Keeter. *Constraint Solver*. 2015. URL: <https://www.mattkeeter.com/projects/constraints/> (elérés dátuma: 2024. ápr. 24.)
- [13] Glenn A. Kramer. „Using degrees of freedom analysis to solve geometric constraint systems”. *Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications - SMA '91*. SMA '91. ACM Press, 1991. DOI: 10.1145/112515.112566. URL: <http://dx.doi.org/10.1145/112515.112566>.
- [14] E. Pennestri, M. Cavacece és L. Vita. „On the Computation of Degrees-of-Freedom: A Didactic Perspective”. *Volume 6: 5th International Conference on Multibody Systems, Nonlinear Dynamics, and Control, Parts A, B, and C*. DETC-CIE2005. ASME, 2005. jan. DOI: 10.1115/detc2005-84109. URL: <http://dx.doi.org/10.1115/DETC2005-84109>.
- [15] Simon Peyton Jones, Andrew Gordon és Sigbjorn Finne. „Concurrent Haskell”. *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96*. POPL '96. ACM Press, 1996. DOI: 10.1145/237721.237794. URL: <http://dx.doi.org/10.1145/237721.237794>.
- [16] Jarrett Revels, Miles Lubin és Theodore Papamarkou. *Forward-Mode Automatic Differentiation in Julia*. 2016. DOI: 10.48550/ARXIV.1607.07892. URL: <https://arxiv.org/abs/1607.07892>.
- [17] Cheng Shao. *WebAssembly backend merged into GHC*. 2022. URL: <https://www.tweag.io/blog/2022-11-22-wasm-backend-merged-in-ghc/>.
- [18] SolveSpace Contributors. *SolveSpace*. URL: <https://solvespace.com/tech.pl> (elérés dátuma: 2024. ápr. 24.)
- [19] Ivan E. Sutherland. „Sketchpad: a man-machine graphical communication system”. *Proceedings of the May 21-23, 1963, spring joint computer conference on - AFIPS '63 (Spring)*. AFIPS '63 (Spring). ACM Press, 1963. DOI: 10.1145/1461551.1461591. URL: <http://dx.doi.org/10.1145/1461551.1461591>.
- [20] Jasper Van der Jeugt. URL: <https://jaspervdj.be/websockets/>.