
BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM

**Tudományos Diákköri Konferencia 2011**

# Szenzorhálózatos esemény-előrejelző algoritmus vizsgálata közlekedési csomópontokban

**Készítette:**

Kalmár András (IG8JX3)

Villamosmérnöki szak (MSc)

**Konzulensek:**

Dr. Vida Rolland, egyetemi docens

Öllös Gergely, doktorandusz

# Tartalom

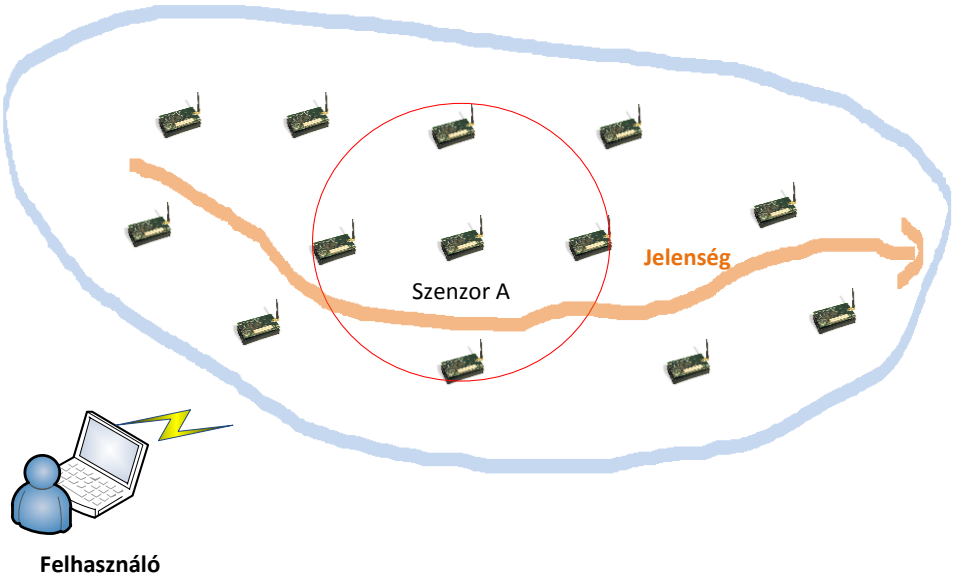
# 1 Bevezetés

Az elektronika, a vezeték nélküli kommunikáció és a digitális rendszerek fejlődésével napjainkra lehetőség nyílt költséghatékony, kis fogyasztású, multifunkcionális szenzorok előállítására. Ezeknek a kisméretű szenzoroknak köszönhetően, melyek érzékelő, adatfeldolgozó és adattovábbító komponensekből állnak, születhetett meg az ötlete egy olyan szenzorhálózatnak, mely nagyszámú szenzorcsomópont összehangolt munkáján alapul. Ez azonban kihívást jelent a szenzorhálózat kommunikációs protokolljainak, mivel azoknak rendelkezniük kell önszervező képességgel a sokszor véletlenszerűen lehelyezett, mozgó vagy lemerülő szenzoroknak köszönhető topológia változások kezelésére.

Vezeték nélküli szenzorhálózatnak (Wireless Sensor Network - WSN) nevezünk nagyszámú független (autonóm) intelligens érzékelőkből álló kooperatív hálózatot, melyben az egyes érzékelők a kitűzött feladat végrehajtását elosztott módon valósítják meg. Az elosztott működés oka, hogy a fizikai rendszer, melyet mérni vagy befolyásolni szeretnénk, térbeli kiterjedése nem teszi lehetővé, hogy egyetlen központi eszközzel valósítsuk meg a feladatot. A szenzorok összehangolt mintavételezésével és jelfeldolgozásával a szenzorhálózat magasabb fokú érzékenységet és zajelnyomást képes biztosítani. A hálózati kapcsolatok (melyek gyakran kis sávszélességű rádiós csatornákat jelentenek) korlátai teszik szükségessé az adatok minél magasabb szintű feldolgozását az érzékelők szintjén. Tipikus elrendezésben az érzékelők különböző környezeti paramétereket mérnek (légnyomás, hőmérséklet, stb.), és a mért értékeket vagy az azokból származtatott mennyiségeket egy központi csomópont - a bázisállomás – felé továbbítják. A hálózatot alkotó autonóm elemek rendelkeznek saját CPU-val, memóriával, saját energiaforrással, különböző érzékelőkkel és A/D átalakítóval, kommunikációs interfészekkel (pl. rádió, soros vonali interfész) és néhány esetben beavatkozó eszközökkel is. Mivel az egyes szenzorok kis memóriával rendelkeznek, és a hálózat telepítése után sok esetben nehezen megközelíthetőek, ezért a kommunikációs interfész tipikusan rádiós csatornát jelent. A szenzor csomópontok elsődleges energiaforrása a saját akkumulátoruk, azonban néhány esetben el vannak látva a működési környezettől függő úgynevezett másodlagos energiaforrással is, amely a környezetből képes energiát kinyerni, továbbá a szenzorok alacsony fogyasztása (~50 mW) kedvez az újszerű energiaforrások alkalmazásának is. (pl napelem cella, mechanikus vibrációs eszközök)

A TDK dolgozatomban bemutatom ezen hálózatok egy hasznos felhasználását, az esemény-előrejelzést, mely során a hálózatot alkotó szenzorok különféle környezeti paraméter-változásokból próbálnak elosztott módon egy-egy esemény megjelenésére következtetni, majd az esemény ismétlődése esetén előre jelezni azt. Az esemény-előrejelzés többek között alkalmazható alvás-üzemmód vezérlő algoritmusok segítésére, közlekedési torlódás előrejelzésre, vagy akár beteg-felügyeleti rendszerekben is.

A dolgozatban ismertetek egy esemény-előrejelző rendszert, mely fuzzy halmazelméletre épít. Az eljárásban az eseményeket a következő három paraméter jellemzi. Az első egy fuzzy (membership) paraméter mely leírja, hogy egy adott esemény mennyire felel meg egy előre definiált referencia eseménynek (melyet detektálni szeretnénk). A második paraméter az esemény bekövetkeztének ideje, a harmadik a detektáló szenzor azonosítója. Abban az esetben, ha egy szenzor eseményt észlel, korlátozott elárasztásos (broadcast) üzenet formájában értesíti erről a környezetét (tehát minden környező szenzor veszi ezt az üzenetet, ami kommunikációs hatókörön belül van), így egy adott jelenség átvonulása a hálózaton eseményszekvenciát generál majd.




1. ábra Egy jelenség okozta esemény-szekvenciából az „A szenzor” csak a kommunikációs hatókörében levő szomszédos szenzorok mérési eredményeit észleli (lokáli szekvencia)

Ebből a teljes hálózatra kiterjedő szekvenciából egy adott szenzor által érzékelt rész-szekvenciát nevezünk lokális szekvenciának. Abban az esetben, ha az egyes szenzorok

képesek a teljes hálózatban gyakran ismétlődő jelenségek által keltett esemény-szekvenciák közül kinyerni az általuk érzékelt lokális esemény-szekvenciákat, akkor a központi csomópont, amely minden szenzorral kapcsolatban van, és felügyeli a hálózat működését, tudatában lesz a teljes esemény-szekvenciának, ez pedig felhasználható céltárgy követésekor illetve esemény-előrejelzéskor. A lokális esemény-szekvenciák kinyerése nehéz feladat, hiszen a szekvenciák a legtöbb esetben zajjal terheltek, valamint a különböző jelenségek által keltett sorozatok át is lapolódhatnak egymással.

A szenzorok az általuk regisztrált, valamint a környező szenzorok által regisztrált mérési eredményeket az úgynevezett TSS adatstruktúrában (Time Space Fuzzy Signature) tárolják idő szerint rendszerezve. Szemléltetésképp tekintsünk egy autópályán elhelyezett szenzorhálózatot. Mivel a megfigyelt útszakaszon egyszerre több autó is lehet, az autók sebessége ingadozik, az események zajosak lehetnek, a két útirány eseményeit pedig a szenzorok nem képesek megkülönböztetni, ezért a szenzoroknak nem problémamentes kinyerni a tiszta esemény-szekvenciákat (itt most a két haladási irányban közlekedő autók által keltett esemény-szekvenciákat) a zajos és kevert eseményhalmazból. A javasolt eljárás erre ad megoldást.

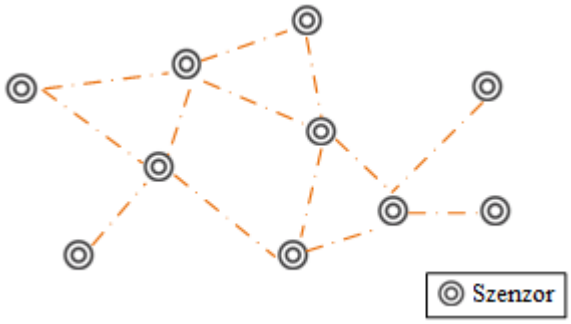
# 2 Elméleti háttér

Ebben a fejezetben röviden bemutatom a vezeték nélküli szenzorhálózatokat, kitérek ezen hálózatok sajátosságaira, tervezési szempontjaira, illetve ismertetek néhány alkalmazási területet is. Ezt követően ejtek néhány szót a klaszterezési eljárásokról, mivel a bemutatni kívánt eljárás tartalmaz klaszterező lépéseket.

## *2.1 Szenzorhálózatokról általánosan*

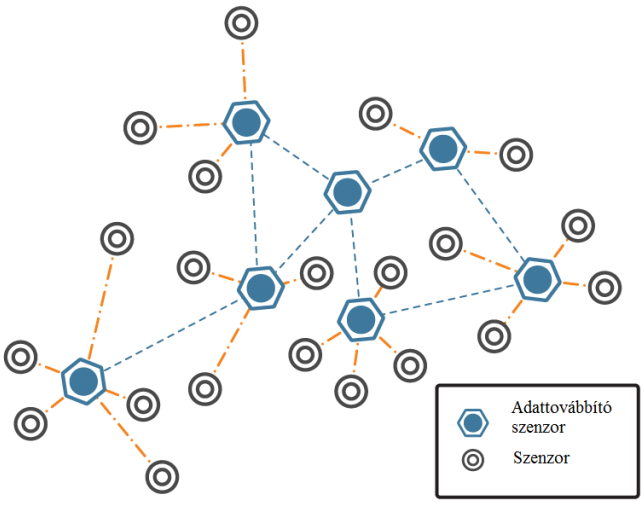
A vezeték nélküli szenzorhálózatok tipikusan kevéssé vagy egyáltalán nem rendelkeznek előzetesen meghatározott infrastruktúrával. Egy WSN hálózat több száz, vagy akár több ezer szenzorból áll, melyek együttesen figyelnek egy adott területet, hogy adatokat gyűjtsenek a környezetről. Alapvetően két típusú hálózatot különböztetünk meg: a strukturált és a strukturálatlan vezeték nélküli szenzorhálózatokat. A strukturálatlan WSN-ek sűrűn telepített

szenzor csomópontokból állnak, melyek ad-hoc módon kommunikálnak. Telepítés után a hálózat magára van hagyva, hogy teljesíthesse a megfigyelési vagy felügyeleti feladatait.




2. ábra Egy strukturálatlan WSN hálózati topológiája

Az ilyen típusú hálózatokban a hálózat menedzselése (kapcsolatok felépítése és bontása, hálózati hibák észlelése) nehézkes a szenzorok nagy száma miatt. Ezzel szemben egy strukturált hálózatban néhány vagy akár az összes szenzor egy előre meghatározott szisztéma szerint van telepítve. Ezen szenzorhálózatok egyik legnagyobb előnye, hogy kevesebb csomópontot kell telepíteni, ezzel pedig csökkennek a hálózat-felügyeleti és menedzsment költségek. A rögzített pozíciókban található szenzorok a rádiós hatósugarukkal mintegy lefedik a figyelni kívánt területet, ezzel biztosítják, hogy legyen kommunikációs útvonal bármely két érzékelő között.




3. ábra Egy strukturált WSN hálózati topológiája

Ezzel szemben a strukturálatlan hálózatokban előfordulhat, hogy olyan klasztercsoportok jönnek létre, a véletlenszerű telepítés következményeként, melyek rádiós összeköttetés szempontjából el vannak vágva egymástól

A vezeték nélküli szenzorhálózatoknak megvannak a saját, egyedi hálózattervezési szempontjaik illetve kötöttségeik. A szenzorhálózatok koncepciójának megalkotásakor az egyik fő szempont volt a hálózatot alkotó szenzorok kis mérete, nagy száma illetve azok alacsony ára. Ez a koncepció komoly megkötést jelent a szenzorok energiatárolási képességeire, ugyanis azok csak kisméretű és átlagos akkumulátorral lehetnek felszerelve. Ebből következően a hálózat által futtatott protokollok szükségszerűen energia-hatékonyak kell, hogy legyenek, annak érdekében, hogy a hálózat élettartama a lehető legnagyobb legyen. További erőforrás megkötéseket jelentenek a rövid hatótávolságú rádiós csatorna, a kis kommunikációs sávszélesség valamint a korlátozott adattárolási és feldolgozási képesség. Strukturált szenzorhálózat esetén az előredefiniált hálózati struktúra kialakítása alkalmazás illetve környezet függő. Beltéri alkalmazás esetén kevesebb csomópont is elég lehet a megfelelő működéshez, míg egy kültéri hálózat esetén a több szenzor csomópont és az ad-hoc (struktúra nélküli) telepítés lehet előnyös. A strukturálatlan telepítés szintén hasznos lehet, amennyiben a környezet miatt a szenzorok telepítésük után elérhetetlenek, valamint ha több száz vagy akár több ezer szenzorból áll a hálózat.

A hálózatot alkotó szenzorok több különböző fajta érzékelővel rendelkezhetnek (szeizmikus, termikus, vizuális, akusztikus) de lehet rajtuk radar is; ezek segítségével sokféle környezeti feltételt képesek megfigyelni egyazon időben, mint például a hőmérséklet, páratartalom, közlekedési mozgások, fényviszonyok, nyomás, zajszint, mechanikai stressz, sebesség, stb.. Az érzékelő szenzorok használhatóak folyamatosan a környezeti feltételek érzékelésére, egy esemény bekövetkeztének detektálására, egy esemény azonosítására, helymeghatározásra, szabályozórendszerek vezérlésére. A kisméretű érzékelők és a vezeték nélküli kommunikáció miatt ezek a szenzorok nagyon ígéretesnek tűnnek számos új alkalmazási területen, például: katonai, környezeti, egészségügyi, otthoni, és kereskedelmi alkalmazások. A hadászatban alkalmazhatóak ezek a hálózatok hadszíntér felügyeletre, céltárgy követésére valamint felderítésre. Szenzorhálózatok segítségével egy adott területen belül pontosan kimutathatóak az idegen kémiai anyagok is, a levegőben vagy a vízben, meghatározható a típusuk, a koncentrációjuk és a kiterjedésük is.

Egészségügyben megvalósíthatóak az intelligens gyógyszer-adminisztrációs rendszerek, illetve a folyamatosan működő beteg-felügyeleti rendszerek, amennyiben az egyes szenzorok a páciensek különféle élettani jeleit monitorozzák. Otthoni alkalmazásként a szenzorhálózatoknak fontos szerepe juthat az intelligens környezet felhasználóhoz történő adaptálásában. A természetben való alkalmazás lehet például az, amikor a különféle környezeti paraméter-változásokból becslik a szenzorok egy katasztrófa bekövetkeztéig hátralevő időt. Abban az esetben, ha a szenzorok egy vulkán krátere mentén vannak elhelyezve és képesek érzékelni a szeizmikus mozgásokat, akkor a hálózat képessé válik detektálni illetve nyomon követni a kráterből kiinduló földrengéseket, valamint vulkáni kitöréseket.

## 2.2 A szenzorok fizikai felépítése

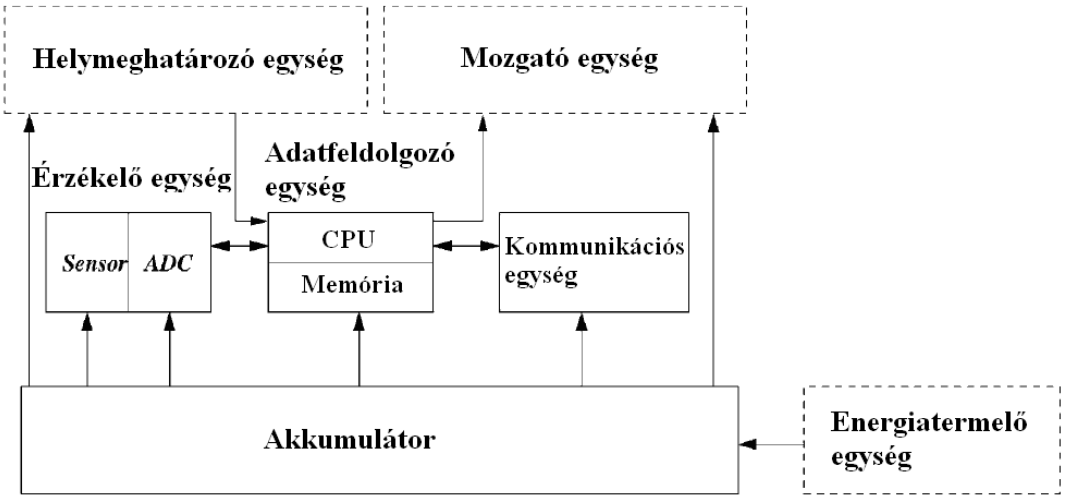
A szenzorok egy lehetséges általános felépítése az 1. ábrán látható. Négy fő komponensük van: az érzékelő egység, az adatfeldolgozó egység, a kommunikációs egység és az akkumulátor. Alkalmazástól függően lehetnek még további részei is, mint például a helymeghatározó egység, az energiatermelő egység (pl. napelem), vagy a szenzor mozgatásáért felelős egység (az ábrán mozgató egység).

Az *érzékelő egység* két további részre osztható, magára az érzékelőre és az analóg-digitál (A/D) átalakítóra. Az A/D átalakító felel azért, hogy az érzékelő által kreált analóg jeleket digitális jelekké alakítsa, majd azokat az adatfeldolgozó egységnek továbbítsa, ami így képes lesz a mért adatok feldolgozására.

Az *adatfeldolgozó egység* szintén két részből épül fel: egy mikroprocesszorból, illetve egy kisméretű adattárolóból. A mikroprocesszor egyrészt az adatok feldolgozását végzi, másrészt futtatja azokat az alkalmazásokat, melyek azért felelnek, hogy a hálózat szenzorjai együttműködve teljesíteni tudják a kijelölt feladatokat.

A *kommunikációs egység*en keresztül kommunikál a szenzor a hálózattal. A rádiófrekvenciás kommunikáció komplex áramkör megvalósítást igényel, ezért a szenzorok árát növeli. A szenzorhálózatok többségében rádiófrekvenciás kommunikációt alkalmaznak, mert kicsik a

csomagméretek és az adatforgalom moderált, valamint a rádiófrekvenciás adattovábbítás robosztusabb, mint az optikai alapú, hiszen utóbbinál a szenzorok helyzete is befolyásoló tényező.




4. ábra Egy általános szenzorfelépítés

Az egyik legfontosabb komponens a szenzorokon az akkumulátor, amely a működéshez szükséges energiát biztosítja. Mivel a szenzorok a telepítésük után gyakran hozzáférhetetlenek, ezért az élettartamukat az akkumulátorban tárolt energia mennyisége határozza meg. A szenzorokra előírt méretkorlátozások miatt az energia is korlátozott mennyiségben áll rendelkezésre.

A legtöbb adattovábbító protokoll és érzékelő alkalmazás számára nélkülözhetetlen a szenzor pontos helyzetének ismerete. Ennek meghatározására szolgál a *helymeghatározó egység*. Ez lehet GPS (Global Positioning System - Globális Helymeghatározó Rendszer), amely 5m-es pontossággal képes meghatározni a célpont helyét. Ez azonban nagyon megemelné az egyes szenzorok árát, és maga a hálózat sem lenne költséghatékony. Ez az elképzelés tehát a szenzorhálózatokkal szemben támasztott egyik alapkövetelményt sértené meg, ezért a jövőben nem biztos, hogy megvalósul. Egy másik megközelítés szerint korlátozott számú érzékelő rendelkezne GPS-el, és ezek a szenzorok segítenének a többi szenzornak saját helyzetük meghatározásában.

A mozgató egység képes a szenzor helyzetének megváltoztatására. Ez akkor lehet hasznos, ha a szenzor csak helyzetének megváltoztatásával képes végrehajtani a kijelölt feladatát.

## 2.3 A WSN-ek hálózati topológiája

A szenzorhálózatok topológiája változó képet mutat az idő előrehaladtával; gondoljunk például a mozgó, illetve a lemerülő akkumulátoruk miatt kikapcsoló szenzorokra. Három kategóriába sorolhatjuk a topológiai változásokat:

Telepítés: A hálózat kezdeti topológiáját annak első telepítése, vagyis a szenzorok első elhelyezése adja. Ez történhet például úgy, hogy repülőgépből kidobják a szenzorokat, valamilyen hordozórakétával átlövik azokat a megfelelő területre, vagy akár valaki egyenként is lehelyezheti őket.

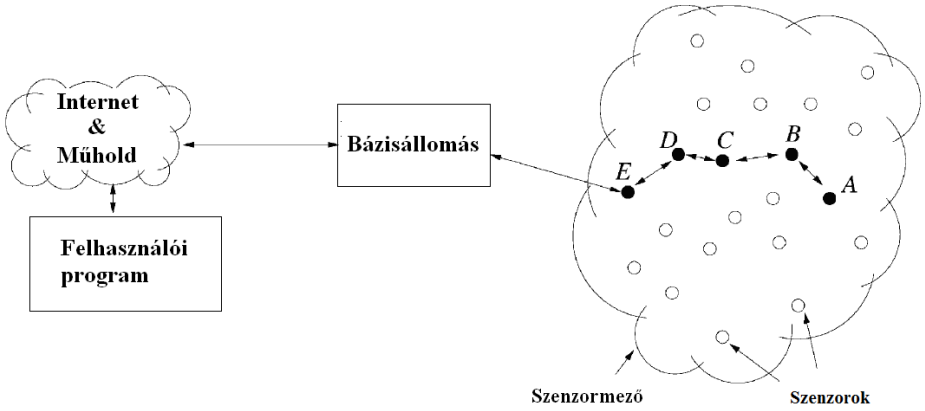
*Működés*: A hálózat működése közben a szenzorok mozoghatnak, akkumulátoruk lemerülésével használhatatlanná válhatnak, megsemmisülhetnek, stb. Törekedni kell a tervezés során arra, hogy a működés közbeni topológiaváltozást is figyelembe véve minimalizáljuk a hálózat költségeit és biztosítsuk annak flexibilitását.

*Újratelepítés*: A hálózatnak működés közben is bővíthetőnek kell lennie, akár új területek monitorozásával kapcsolatban, akár a használhatatlanná vált szenzorok cseréjét tekintve. Mindkét eset bekövetkezése esetén a hálózat újraszervezése szükséges.

A multi-hop típusú hálózatokban az átviteli közeg vezeték nélküli, vagyis általában rádiós vagy optikai összeköttetés áll fenn a csomópontok között. Az optikai adatátvitelt csak akkor lehet alkalmazni, ha a kommunikáló felek egymás látótávolságában helyezkednek el, ami a szenzorhálózatok esetében nem mindig garantálható, ezért az ilyen típusú hálózatokban a rádiós adattovábbítást részesítik előnyben. Az egyik lehetőség a rádiós kommunikáció megvalósítására a szabadon használható ISM (Industrial Sciencific Medical) frekvenciasávok igénybevétele. Tekintetbe véve a szenzorok kis energiatároló kapacitását illetve az antennájuk kis nyereségét, a magasabb frekvenciák alkalmazása lehet előnyös. Említést érdemel, hogy a szenzorhálózatok széles felhasználási lehetőségeiből adódóan bizonyos alkalmazások szokatlan átviteli közeg használatát igénylik. A vízi alkalmazások esetén például a nagy

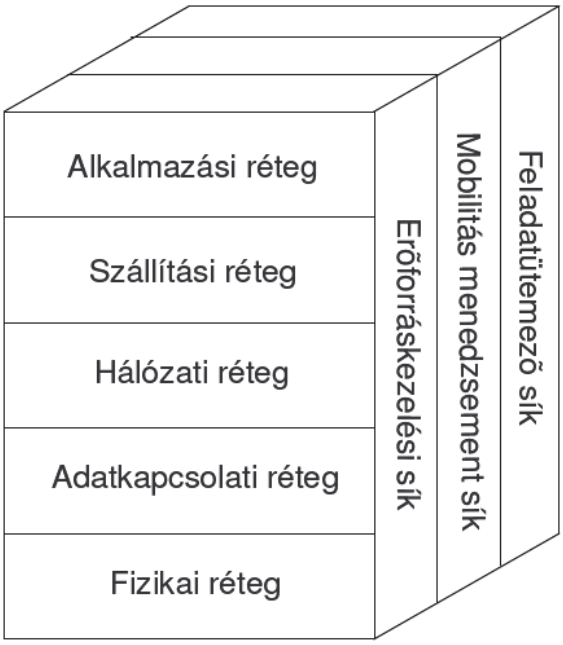
hullámhosszúságú sugárzás lehet az a kommunikációs közeg, amely képes behatolni a vízfelszín alá.

A szenzorok általában elszórva helyezkednek el egy szenzormezőben, mint ahogy azt a 5. ábra mutatja. Minden egyes elszórt szenzor képes adatokat gyűjteni a megfigyelt jelenségről, majd ezeket továbbítani a bázisállomásnak, onnan pedig a végfelhasználónak. Az adat "multi-hop" hálózati architektúra szerint több szenzoron keresztül jut el a bázisállomásig. A bázisállomás az interneten vagy műholdon keresztül kommunikál a felhasználói programmal.




5. ábra Szenzorok elhelyezkedése

A kommunikációs protokollok összetettebb rétegszerkezete a 3. ábrán látható, azonban legtöbbször a protokoll vermek nem implementálják az összes réteget. Ezek a protokollok képesek hatékonyan felhasználni a rendelkezésre álló energiát, az adattovábbítás útvonalát a lehető legkedvezőbben megválasztani, illetve elősegítik a szenzorok közös munkáját. A hálózati protokoll vermek az alkalmazási rétegből, a szállítási rétegből, a hálózati rétegből, az adatkapcsolati rétegből, a fizikai rétegből, az erőforrás kezelési síkból, mobilitás menedzsment síkból, valamint a feladat-ütemező síkból állnak. (3. ábra)




6. ábra A hálózati protokoll rétegszerkezete

Alkalmazástól függően különböző alkalmazási szoftver telepíthető és használható az alkalmazási rétegben. A szállítási réteg kezeli a hálózatban áramló adatcsomagokat. A hálózati réteg felel a szállítási rétegtől kapott adatcsomagok útvonalválasztásáért. A közeghozzáférést szabályozó hálózati alrétegre (MAC – Media Acces Control sublayer), amit az adatkapcsolati réteg részeként tekinthetünk, komoly feladat hárul, mivel a környezet zajossága illetve a szenzorok mobilitása mellett is képesnek kell lennie az adatfolyamok ütközésének minimalizálására, ugyanakkor az energiafelhasználást is alacsony szinten kell tartania. A fizikai réteg felel az egyszerű, de robosztus modulációért, valamint az adatok fogadásáért és továbbításáért.

A hálózati protokoll három síkja az erőforrás kezelési sík, a mobilitás menedzsment sík, és a feladat-ütemező sík felelősek az erőforrások szétosztásáért az egyes rétegek között, a szenzor mozgásának felügyeletéért, valamint az egyes feladatok ütemezéséért. Ezek a síkok segítenek a szenzoroknak összehangolni az egyes érzékelői feladataikat, és minimalizálni a teljes energiafelhasználást.

Az *erőforrás-kezelési sík* irányítja a szenzor energiafelhasználását, például kikapcsolja a vevőegységet egy adatcsomag érkezése után; ezzel elkerüli az esetleges duplikátumok vételét, és energiát spórol. Az is elképzelhető, hogy a szenzor „broadcast" üzenetet küld a szomszédainak, ha alacsony már az energiaszintje, értesítve a többi szenzort, hogy ezentúl nem fog tudni részt venni az adatcsomagok továbbításában, és a továbbiakban csak méréseket végez.

A *mobilitás menedzsment sík* érzékeli és regisztrálja a szenzor mozgását, így a szenzor mindig tudatában van a felhasználóhoz vezető kommunikációs útvonalnak, illetve vezethet egy listát a hatókörében található többi szenzorról is. Ismervén azt, hogy mely szenzorok vannak a kommunikációs határon belül, a szenzorok képesek megosztani egymás között az elvégzendő feladatokat.

A *feladatütemező sík* kiegyenlíti és ütemezi az érzékelési és mérési feladatokat egy adott régióban található szenzorok között. Egy adott régióban nem kell az összes szenzornak érzékelési feladatokat ellátnia; ha például egy szenzor energiaszintje már alacsony, akkor ő bizonyos feladatok elvégzése alól felmentődik, az így kimaradt méréseket pedig a többi szenzor végzi el.

A menedzselő síkok valósítják meg a szenzorok együttműködését az energia-felhasználásuk minimalizálása mellett, biztosítják az adatokat hozzáférhetőségét a hálózatban, és elérhetővé teszik az egyes elemek erőforrásait a hálózat egésze számára. A hálózati protokoll síkok nélkül az egyes szenzorok csak önállóan működhetnének.

A dolgozatban ismertetett esemény-előrejelző algoritmus szenzorhálózatos környezetet igényel, amely hálózatban az algoritmust futtató szenzorok bizonyos időközönként mintavételeznek. Ha a mintavételezett értékhez rendelt fuzzy tagsági függvény értéke meghalad egy bizonyos küszöbértéket, akkor ez az esemény tárolásra kerül a szenzor adatbázisában, illetve a szenzor „broadcast" üzenet formájában értesíti erről a kommunikációs hatókörében található szomszédos szenzorokat. Ebből következően egy jelenség átvonulása a hálózat területén esemény-szekvenciákat generál. Az algoritmus azt a feladatot látja el, hogy megbecsüli ezen szekvenciák számát, megkísérli az azonos szekvenciákat reprezentáló

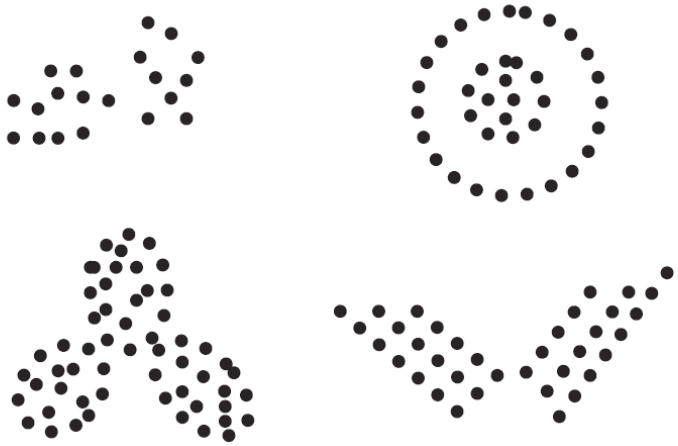
esemény-szekvenciákat összerendelni (azonos klaszterbe szervezni), majd pedig ezekből kiszűrni a „tiszta szekvenciákat”.

## 2.4 Klaszterezés

Mivel a bemutatni kívánt algoritmus sarkalatos pontját képezi két különböző klaszterező eljárás is, ezért ebben a fejezetben be kívánom mutatni röviden a klaszterező algoritmusok elméleti alapjait.

A klaszterező algoritmusok adat-objektumokat (mintázatok, entitások, egyedek, megfigyelések, egységek) partícionálnak bizonyos számú klasztercsoportba. (csoportok, részhalmazok, kategóriák) Noha érdemes megjegyezni, hogy nincs egységesen elfogadott és precíz definíciója a klaszterezésnek. A definíciók megegyeznek abban a pontban, hogy a közös klasztercsoportba sorolt egyedek sokkal inkább hasonlítanak egymáshoz, mint a más klasztercsoportokba sorolt egyedekhez.

Adott $X = \{x_1,...,x_j,...,x_N\}$ bemenet, ahol $x_j = (x_{j1}, x_{j2},...x_{jd}) \in R^d$, melyben minden egyes $x_{jd}$ érték egy tulajdonság (jelző, dimenzió, változó).



7. ábra A klaszterezés szubjektivitásának szemléltetése. Egy „durvábban” klaszterező eljárás két csoportba, míg egy szofisztikáltabb 6 klasztercsoportba sorolja az egyedeket

1. „Hard partitional” klaszterezés megkísérli k darab partícióra osztani az X-t , ahol $C = \{C_1....,C_k\}$ $(K \le N)$ , ahol:

$C_i \neq 0, i = 1...K$ ;

$\bigcup_{i=1}^{K} C_i = X$ ;

$\mathrm{C_i} \cap \mathrm{C_j} = 0, \mathrm{i, j} = 1...\mathrm{K}$ és $i \neq j$

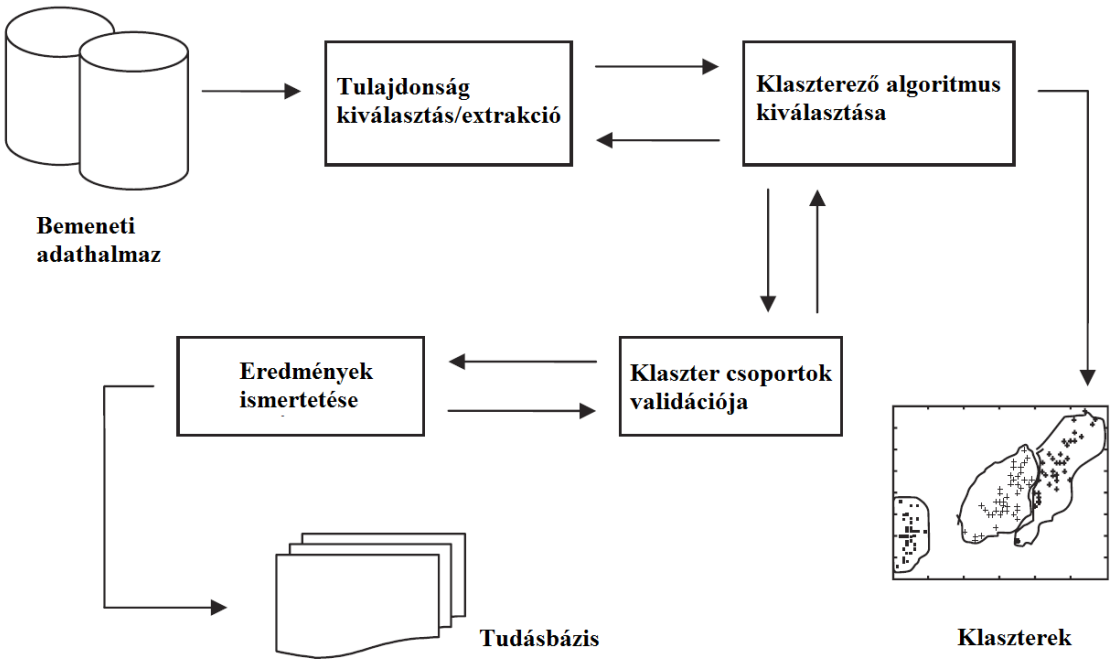
2. Hierarchikus klaszterezés egy faszerű, egymásba-ágyazott struktúrát épít fel X-ből, ahol $H = \{H_0...H_Q\}(Q \leq N)$ amennyiben $C_i \in H_m, C_j \in H_l$, és $m > l$, akkor következik, hogy $C_i \subset C_j$ vagy $C_i \cap C_j = 0$ minden $i \neq j$ és $m, l = 1,...., Q$

A klaszterező eljárások alapvetően a következő négy lépésből állnak [5]:

1. *Tulajdonság kiválasztás vagy extrakciója.* A tulajdonság kiválasztás annak vagy azoknak a tulajdonságoknak a kiválasztása, melyek szerint az adat-objektumok leginkább megkülönböztethetőek (differenciálhatóak), legkönnyebb őket klaszterekbe szervezni. Az extrakció során különféle transzformációkat hajtunk végre a tulajdonság halmazon, annak érdekében, hogy szemléletesebb és jobban használható tulajdonságokat kapjunk az eredeti halmazból. Az extrakcióban benne rejlik a lehetőség, hogy a klaszterezés szempontjából használhatóbb tulajdonságokat hoz létre, melyek alkalmasabbak arra, hogy az objektum halmazban található mintázatokat felderítsük. Fontos megjegyezni, hogy gyakran olyan tulajdonságok jönnek létre ennek eredményeképpen, melyekhez nem tudunk fizikai tartalmat rendelni. A klaszterezéshez felhasznált tulajdonságok elegáns megválasztása vagy generálása jelentősen csökkentheti a klaszterezéshez szükséges tárhelyet, a mérések költségét, egyszerűsítheti a későbbi tervezési folyamatokat, valamint könnyítheti az adatok megértését. Általánosan azt lehet mondani, hogy az ideálisan megválasztott tulajdonságok a folyamat során segítenek szétválasztani a különböző klasztercsoportokba tartozó elemeket, kevéssé érzékenyek a zajnak tekinthető objektumokra a halmazban, könnyű ezeket kinyerni illetve értelmezni.

2. *Klaszterető algoritmus megválasztása.* Ebben a szakaszban választunk egy megfelelőnek gondolt távolság definíciót, mellyel képesek vagyunk meghatározni bármely két adathalmazbeli objektum közti távolság (vagy közelség) értékét, valamint megállapítunk egy kritérium függvényt. Az egyes halmazbeli elemek a szerint lesznek

klasztercsoportokba sorolva, hogy a korábban definiált távolság függvény alapján mennyire vannak közel (hasonlítanak) egymáshoz. A távolság függvény megválasztása után a klaszterezés optimalizálási problémának tekinthető, melynek célfüggvénye a korábban meghatározott kritérium függvény. A végeredményként adódó klasztercsoportok és azok száma a kritérium függvénytől függ. A klaszterezés az élet majdnem minden területén előfordul, nem véletlen, hogy számtalan klaszterező algoritmust fejlesztettek ki, melyek különféle alkalmazási területeken oldanak meg problémákat. Mindezek ellenére nem létezik egységesen alkalmazható eljárás, ezért általános alapelvnek tekinthető, hogy előbb célszerű részletesen feltérképezni a megoldandó probléma természetét, majd az ahhoz leginkább passzoló eljárást alkalmazni.

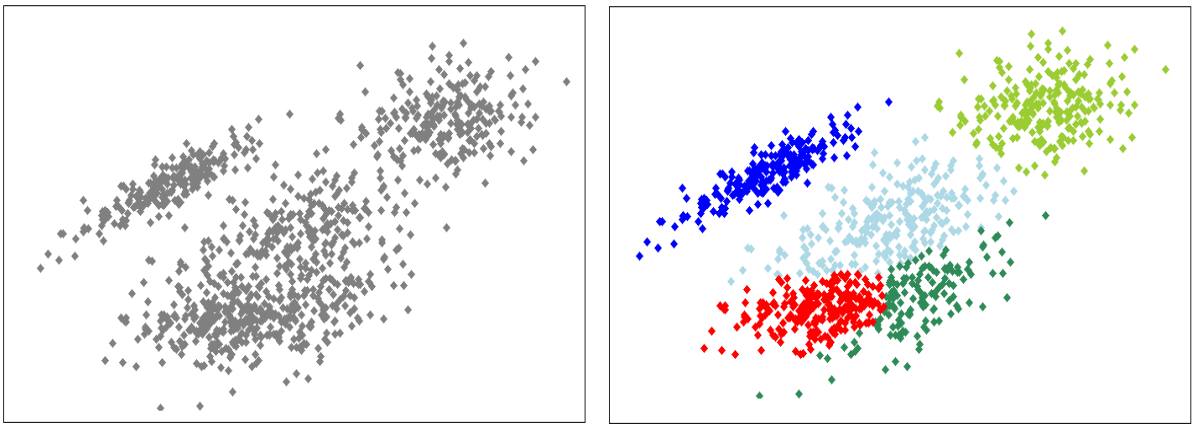



*8.* **ábra** A klaszterezés folyamata

3. *Klasztercsoportok validálása.* Adott bemeneti adathalmazt bármely klaszterező eljárás fel fog bontani bizonyos számú partícióra, annak ellenére is, hogy nem feltétlenül létezik struktúra vagy mintázat a halmazbeli elemek között. Mindezek mellett a különféle megközelítésű klaszterező eljárások más-más eredmény halmazokat adhatnak kimenetül, sőt ugyanazon klaszterező algoritmus más-más paraméter

beállításokkal más-más eredményre vezethet. Ezért fontos a kapott eredmények kiértékelése és azokhoz bizonyos fokú konfidencia rendelése. Ezeknek az értékeléseknek objektívaknak kell lenniük, és pártatlannak a különböző eljárásokkal szemben. Mélyreható és releváns válaszokat kell tudniuk adni olyan kérdésekre, mint például hány darab klasztercsoport van rejtve az adathalmazban, az eredményül kapott halmazok hasznosak e, vagy csak a klaszterezési eljárás melléktermékei, illetve miért előnyösebb az adott helyzetben egy adott eljárás a többivel szemben.

4. *Eredmények ismertetése.* A klaszterezés legfőbb célja az, hogy elősegítse a felhasználó számára felismerni az adathalmazokban a háttérben meghúzódó mintázatokat, struktúrákat, melyek első pillantásra nem mindig tűnnek ki. Ezáltal a probléma rejtett összefüggései is kirajzolódnak, így az egyszerűbben megoldhatóvá válik.



**9. ábra** Példa klaszterezésre, a bemeneti ponthalmazt – a pontok koordinátája alapján- öt klaszter csoportra osztotta az eljárás (Normális eloszlású ponthalmazokat feltételezett az algoritmus)

Érdemes megfigyelni, hogy a 8. ábra folyamatábráján szerepel egy visszacsatolási irány is. A klaszterezés nem egyirányú. A legtöbb helyzetben számos próba és ismétlés után állnak elő a kívánt és az elvárásokat kielégítő klasztercsoportok. Említést érdemel, hogy nincs egységesen használható kritérium függvény, ami segítene a megfelelő tulajdonság és az ideális klaszterezési módszer kiválasztásában. A keletkező klasztercsoportok validálása segít meghatározni a használt eljárás „jósági tényezőjét”, de az ideális kritérium függvény meghatározása ennek ellenére sok fejtörést okozhat.

## *2.5 A Fuzzy logikáról*

A dolgozat által javasolt algoritmus az események leírására a Fuzzy logikából vett elméleti definíciókat használja fel, ezért ebben a fejezetben röviden ismertetem az „elmosódott halmazok elméletének” alapjait.

### 2.5.1 Fuzzy logikáról általánosan

A fuzzy logika a merev Boole-algebrai struktúrával szemben a természetes emberi gondolkodás rugalmasabb modellezését teszi lehetővé, ezért sok esetben alkalmasabb bonyolult műszaki feladatok megvalósítására. Ez a fajta logika a többértékű logikai szemantikák egyike. Számos alkalommal szembesültek azzal a matematikusok és a filozófusok, hogy természetes fogalmaink igazságtartományának határai nem mindig jelölhetőek ki egyértelműen. Klasszikus példa ennek szemléltetésére a homokkupac paradoxon, mely a következőképpen írható le: adott egy homokkupac. A homokszemeket egyesével elvéve a halomból, hány homokszem elvétele után jelenthetjük ki, hogy a homokszemek összességét már nem tekinthetjük homokkupacnak. A példa paradoxon jellege a homokkupac pontatlan definíciójából ered. A precíz definíció hiánya igen gyakori a mindennapi életben használt fogalmainknál, vagyis a legritkább esetben adhatók meg egzakt matematikai módszerrel. A precíz meghatározásokat használó matematikánk ezért nem alkalmas a mindennapi életben használt pontatlan fogalomdefiníciók formális kezelésére, ezek határai ugyanis „elmosódottak”.

Az európi „nyugati” tudományosság a formális logikát már az ókortól kezdve az igaz és a hamis értékpár világába próbálta belekényszeríteni. Ez a gondolkodás már Arisztotelésznél (görög filozófus, i.e. 384–322) jól megfigyelhető. Olyan logikai–filozófiai alapelvek, mint az ellentmondás törvénye és a kizárt harmadik törvénye egészen az ő korági nyúlnak vissza. Ezek értelmében nem lehet valami egyszerre $A$ és $\overline{A}$, illetve valamelyik a kettő közül igaz kell hogy legyen. Az előbb bemutatott homokkupac-paradoxon azonban jól szemlélteti, hogy ezek az elvek nem mindig teljesülnek. Az arisztotelészi logikát a XIX. században G. Boole (angol matematikus 1815–1864) foglalta axiomatikus rendszerbe. Közismert, hogy

a Boole-algebra, azaz a kétértékű matematikai logika és a halmazalgebra struktúráját, tovább általánosítják az olyan absztrakt algebrák, mint például a háló, melynek részletes vizsgálata G. D. Birkhoff (1884–1944) nevéhez kapcsolódik. forrás[..]

Mindezek ellenére már az ókortól kezdve felmerült a többértékű (kezdetben a háromértékű) logikák formalizálásának igénye. A különféle háromértékű logikákat dolgoztak ki például: Lukasiewicz, Bochvar, Klenee, Heyting, Reichenbach. Ezek megszületése után már könnyű volt ezek n-értékű irányba történő módosítása. Az n-értékű logika területén a legjelentősebb eredmények Lukasiewicz nevéhez fűződnek.

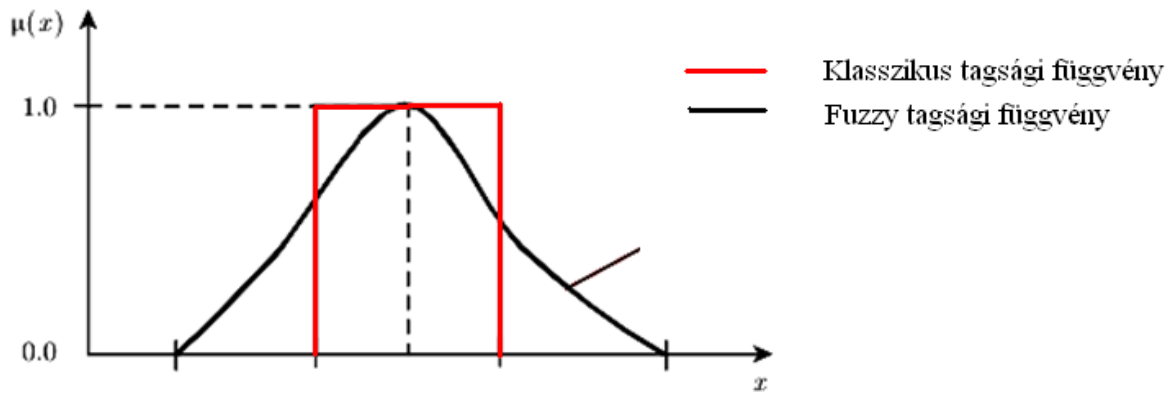
A nyelvi fogalmaink elmosódott határai valamint a többértékű logikai változók lehetséges állapotainak végtelenre való kiterjesztésének igénye okán születhetett meg az ötlete a „részben igaz" állításokat megengedő fuzzy logikának. Matematikailag először Lotfi A. Zadeh, a Berkeley egyetem számítástechnika professzora vizsgálta a fogalmaink igazságtartományának elmosódott határait 1965-ben [5].

Az elmúlt évtizedben, a Lofti Zadeh által 1965-ben bemutatott fuzzy logika a műszaki feladatok széles körében talált alkalmazásra a legkülönfélébb problémák megoldására, mint például a folyamatirányítás, kép-feldolgozás, mintafelismerés és osztályozás, döntéshozatal. Fuzzy rendszereket használnak mosógépek automatizálásánál, videokamerák fókuszának kezelésénél, a TV színeink hangolásánál, automatikus sebességváltóknál a személygépkocsikban, vagy a metróüzemeltetésnél. Mindezek mellett nagy léptékű fejlődést figyelhetünk meg a neurális hálózatoknál is. A fuzzy rendszereket és a neurális hálózatokat sokszor kombinálva alkalmazzák, például a fuzzy következtető rendszereket gyakran alakítják át úgynevezett neuro-fuzzy rendszerekké, ami ötvözi a neurális hálózatok és a fuzzy rendszerek legjobb tulajdonságait. Mára a világon számos neuro-fuzzy rendszert fejlesztettek ki, ezek rendelkeznek a fuzzy rendszerek azon tulajdonságával, hogy a köznapi pontatlan meghatározásokkal rendelkező fogalmainkat képesek kezelni, illetve rendelkeznek a neurális hálózatok tanuló képességével is.

Az emberi agy pontatlan és feldolgozatlan szenzoros információkat kap az érzékelő sejtek összességétől, vagyis az idegrendszertől. A fuzzy elmélet egy szisztematikus eljárást biztosít az ilyen információk formális értelmezéséhez, és ezek számszerű adattá alakításához az úgynevezett tagsági függvények segítségével. Továbbá a „ha-akkor” típusú szabályok képezik az alapját a fuzzy következtető rendszernek (FIS – fuzzy inference system), ami bizonyos helyzetekben rendkívül hatékony módon képes modellezni az emberi viselkedést és döntéshozatalt. Noha a fuzzy logika következtető rendszere strukturált tudásbázissal rendelkezik a szabályok kiértékeléséhez, nem képes alkalmazkodni a külső környezeti feltételek változásához; ezért szokták gyakran egybefoglalni a neurális hálózatok adaptív képességével. Az ezen elgondolás alapján megvalósított rendszereket nevezzük neuro-fuzzy rendszereknek.

### 2.5.2 Alapfogalmak

A hagyományos halmazelmélettel ellentétben, ahol egy adott univerzumbeli elemről egyértelműen eldönthető, hogy tagja-e egy adott halmaznak vagy sem, a fuzzy elmélet által definiált halmazok határai “elmosódottak”, vagyis a gyakorlatban minden az adott univerzumban létező elemhez egy 0 és 1 közötti értéket rendelünk, annak függvényében, hogy az adott elem mennyire tartozik a vizsgált fuzzy halmazhoz. (1- teljesen a halmazhoz tartozik, 0-nem tartozik az adott halmazhoz.) Ezt a függvényt, amely elvégzi ezt a hozzárendelést, tagsági függvénynek nevezzük.




**10. ábra:** Klasszikus és fuzzy tagsági függvények

A 3. ábrán látható egy hagyományos és egy lehetséges fuzzy tagsági függvény. A fuzzy tagsági függvények jelölésére általában a görög μ betűt használják, és alakjukra nincsenek megkötések, gyakran előfordulnak trapéz vagy háromszög alakúak is lehetnek.

A három hagyományos halmazműveletet; azaz a komplementerképzés (negáció), a metszet (konjunkció), és az unió (diszjunkció); végtelenül sokféleképpen lehet kiterjeszteni fuzzy halmazokra. Az X alaphalmazon értelmezett A(x) fuzzy halmaz Zadeh-féle komplemense az a $\overline{A}(x)$ halmaz, melyet az alábbi egyenlőség határoz meg minden x értékre:

$$\overline{A}(x) = 1 - A(x) \tag{2.1}$$

Tetszőleges C függvényt tekinthetünk komplementer függvénynek, ha teljesíti a következő feltételeket:

$$C : [0,1] \to [0,1] \tag{2.2}$$

$$C(0) = 1,\ C(1) = 0 \tag{2.3}$$

$$a_1 \le a_2 \Rightarrow C(a_1) \ge C(a_2) \tag{2.4}$$

Az alaphalmaz azon elemeit, melyekre $\overline{A}(x) = A(x)$ egyensúlyi pontoknak nevezzük. (Mint az a fenti definícióból könnyen leolvasható, a Zadeh-féle komplementer-képzésnek x=0.5 az egyensúlyi pontja.)

Adott A(x) és B(x) fuzzy halmazok esetén, a két halmaz Zadeh-féle metszete a következőképpen áll elő:

$$A(x) \cap B(x) = \min[A(x), B(x)] \tag{2.5}$$

Míg ugyanezen két halmaz Zadeh-féle uniója:

$$A(x) \cup B(x) = \max[A(x), B(x)] \tag{2.6}$$

A minimum- és a maximumképzés asszociatívak, ezért ezek a definíciók kiterjeszthetőek tetszőleges számú elemre.

## *2.6 Az algoritmus elméleti definíciói*

Ebben a fejezetben bemutatom az esemény-előrejelző módszerhez kapcsolódó elemi fogalmakat, melyek alapján az algoritmus kezeli és tárolja a mintavételezések során előálló mérési eredményeket, illetve ezen mérési eredmények összehasonlítására kerülnek. Három alapvető fogalomról lesz szó: az esemény, a TSS, és a TSS távolság.

A módszer modellje szerint a szenzorok elszórtan helyezkednek el a jósolni kívánt esemény (cél esemény) valószínűsíthető felbukkanási helyének környezetében, és folyamatos mintavételezést végezve adatokat szolgáltatnak a cél esemény környezetében felbukkanó eseményekről. A mért értékek úgynevezett *TSS-ekben (Time-Space fuzzy event Signature*) kerülnek tárolásra. Minden szenzor rendelkezik egy TSS adatbázissal, mely a szenzor által mért saját, lokális eseményeket és az azokat megelőző események sorozatát tartalmazzák.[6]

Legyen E az események fuzzy halmaza:

$$E = \{(f, \mu_E(.), ID_f, t_f) \mid f \in F\} \quad \textbf{(2.7)}$$

ahol:

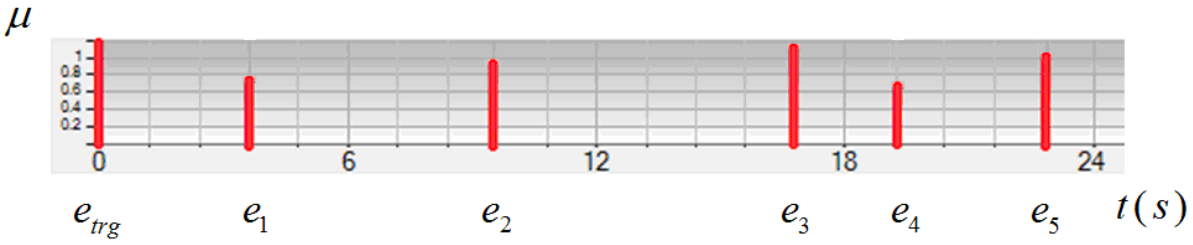
- F: a tulajdonság (feature) tér, vagyis a vizsgált környezeti paraméterek valamilyen tulajdonságát leíró értékek tere;
- f: egy elem a tulajdonság térben;
- $\mu_E(.)$ : a fuzzy halmazt meghatározó tagsági függvény;
- $ID_f$ : annak a szenzornak az azonosítója (ID-je), ahol f bekövetkezett;
- $t_f$ : f detektálásának ideje.

A TSS esemény-szekvenciát a következőképpen definiáljuk:

$$TSS_{ID,i} = \{e_{trg}, e_1, e_2, ..., e_n \mid e \in E\} \quad \textbf{(2.8)}$$

A TSS tulajdonképpen az i. cél eseményt ( $e_{trg}$ ) megelőző események sorát jelenti. A TSS-ben tárolt események többségét a környező szenzorok szolgáltatják. Az $e_1, e_2, ..., e_n$ események rendezve vannak, elsősorban a bekövetkezési idejük ( $t_f$ ) csökkenő sorrendje, másodsorban pedig a szenzorazonosítóik ( $ID_f$ ) szerint.

Ahhoz, hogy az egyes szenzorokon tárolt TSS-eket képesek legyünk összehasonlítani, azokat előbb egységes formára (normál alakra) kell hozni, hogy a későbbiek során ismertetésre kerülő TSS távolság függvény képes legyen az egyes TSS-ek egymáshoz képesti távolságának meghatározására. Egy normál alakra hozott TSS a 4. ábrán látható.




**11. ábra** Egy idő szerint normalizált TSS

Az ábrán látszik, hogy a jósolni kívánt célesemény a 0. időpillanatba lett eltolva, míg az ezt az eseményt megelőző események sorozata pozitív idővel, növekvő sorrendben szerepel, mégpedig oly módon, hogy a céleseményt közvetlenül megelőző esemény szerepel a legkisebb idővel, míg a céleseményhez képest legkorábban történt esemény található legtávolabb az időtengelyen. Ez a fajta normál alak használata a metódus működése szempontjából lényegtelen, bármilyen normál alak alkalmazható mely az egyes TSS-ekben tárolt események között eltelt időintervallumokat összehasonlíthatóvá teszi. Ennek a formának a használata azért előnyös, mert így a normál alakra hozott TSS első elemének (eseményének) szenzor azonosítójából meg lehet állapítani, hogy melyik szenzoron keletkezett az adott TSS.

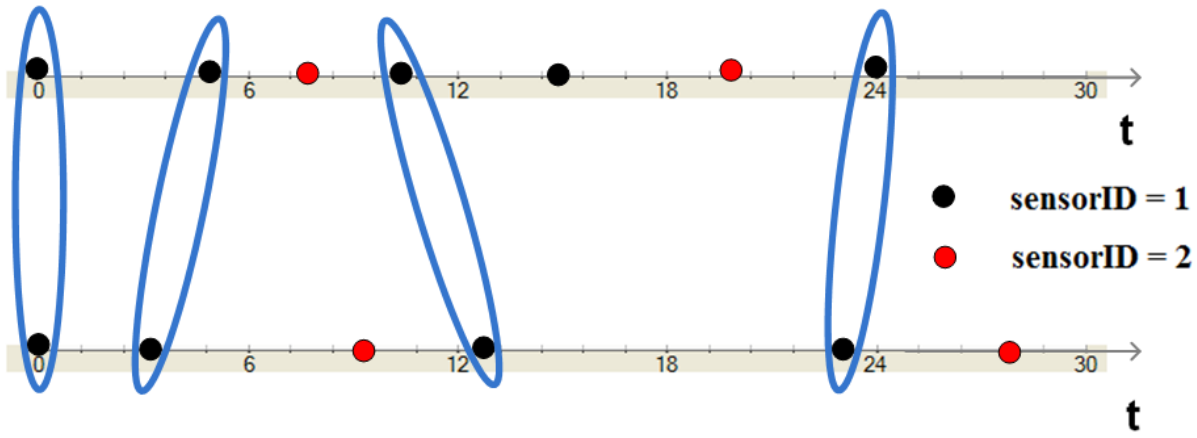
# 3 Az esemény-előrejelző algoritmus működése

A most következő fejezetben bemutatom az esemény-előrejelző metódus működési elvét. Definiálom a TSS távolság függvényt, megmutatom hogyan szerveződnek klaszterekbe az

összehasonlításra került TSS-ek, majd leírom, hogy ezen klaszterekből hogyan kerülnek kinyerésre a „tiszta szekvenciák”.

## *3.1 TSS távolság függvény*

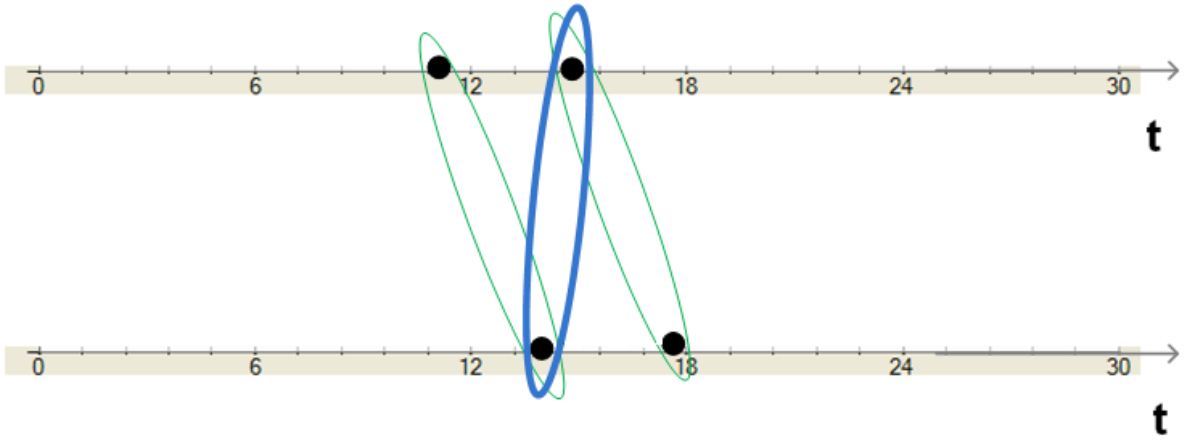
A TSS távolság függvény két korábban normalizált TSS-t hasonlít össze, annak érdekében, hogy egy 0 és 1 közötti értékkel jellemezze az ezekben tárolt esemény-szekvenciák hasonlóságát. A TSS távolság meghatározása alapvetően két részfeladatra osztható fel: első lépésként megpróbáljuk a TSS-ek által tárolt eseményeket párba rendezni, majd a talált párok bizonyos különbözőségeit számszerűsíteni. Az események párba rendezésének egyik fő szempontja, hogy csak az azonos szenzorazonosítóval rendelkező események között keresünk párokat. Az 5. ábrán két TSS-t látunk, melyek két szenzor által érzékelt események időbeli szekvenciáját mutatják: a fekete körök az 1. szenzor, a piros körök a 2. szenzor által érzékelt eseményeket jelölnek.




12**. ábra: Két** TSS eseményeinek egy lehetséges összerendelése

Látható, ahogy a két TSS-ben megjelenő, 1-es szenzorazonosítóval rendelkező eseményeket egy lehetséges módon összerendeltük. Az első TSS ötödik eseményének nem találtunk párt, noha azt összerendelhettük volna a második TSS negyedik eseményével, viszont akkor a jelenlegi párosításban szereplő első TSS negyedik eseménye maradt volna pár nélkül. Felvetődik a kérdés, hogy milyen szempontok alapján célszerű párosítani?

A megfelelő párosítás megtalálása azért fontos, mert a zajtényezők nagy része kiszűrhetővé válna ez által, és azok nem befolyásolnák a két TSS közti távolságot. Az ismétlődő esemény-szekvenciák eseményei közt eltelt időintervallumok kisebb eltérésekkel megegyeznek, így ha megtaláljuk a jó párosítást, akkor nagy valószínűséggel a zajként értelmezhető eseményeknek nem lesz párja, vagy azok időkülönbsége olyan nagy lesz, hogy ez alapján leszünk képesek kiszűrni ezeket. Megállapíthatjuk tehát, hogy az egyik lényeges szempontja a párosításnak az, hogy a talált párok időkülönbségei minimálisak legyenek. A 6. ábra szemléltet egy olyan esetet, amelyben ha csak a fenti szempontot vesszük figyelembe, akkor nem az optimális megoldást kapjuk eredményül (kék színű párosítás).




13. **ábra:** Példa arra, amikor nem ad optimális megoldást a lépésenkénti legkedvezőbb eset választása

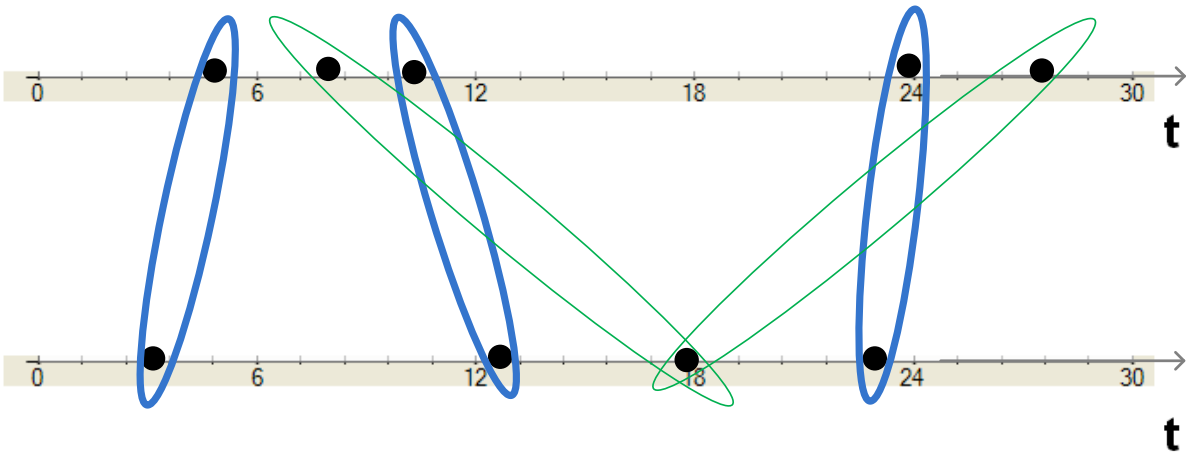
Ha ugyanis az ismétlődő sorozat része az ugyanattól a szenzortól származó, egymást rövid időn belül követő két esemény, és az egyik esetben a sorozat elemei időben kissé elcsúsznak, akkor nem az optimális megoldást találtuk meg. Ebben az esetben az a párosítási szempont volna előnyös, hogy a talált párok időkülönbségeinek összege a lehető legkisebb legyen (zöld színű párosítás). Ez a probléma egyébként megegyezik azzal a gráfelméleti problémával, amikor egy súlyozott élű páros gráfban keressük a maximális összsúlyú párosítást. Ennek a gráfelméleti problémának létezik polinom idejű megoldása, melyet Harold Kuhn publikált 1955-ben, és a „Magyar módszer” nevet adta neki, mivel az algoritmus két magyar matematikus, Kőnig Dénes és Egerváry Jenő, korábbi munkáin alapult. (Az algoritmus fő

lépései a független élek felvétele, illetve javító utak keresése valamint ezek mentén a párosítás növelése, amíg lehetséges.)

Ebből adódik az ötlet, hogy tekinthetnénk a két TSS-beli eseményeket két független ponthalmaznak, mely ponthalmazokban minden elem össze van kötve minden másik ponthalmazbeli elemmel, és ezekhez az élekhez súlyok vannak rendelve a pontpárok időkülönbségeivel arányban. Ebben az esetben feladatunk a maximális összsúlyú párosítás megtalálása volna. A feladat egyediségéből fakadó néhány észrevétel csökkentheti a párosításhoz szükséges számításigényt.

Az első ilyen észrevételről már volt szó, vagyis arról, hogy a párokat csak az ugyanattól a szenzortól származó események között keressük. Ez a gyakorlatban azt jelenti, hogy annyi részgráfban keressük a maximális súlyú párosításokat, ahány szenzortól származó események szerepelnek a két összehasonlításra váró TSS-ben. Ha tehát n darab eseményünk van a két TSS-ben összesen, azok 3 különböző szenzortól származnak, és mind a három szenzortól n/3 darab esemény származik, akkor a párosítást kereső „f függvény” bemeneti ponthalmazának számossága n helyett csak n/3 lesz, és a függvény háromszor fog lefutni.

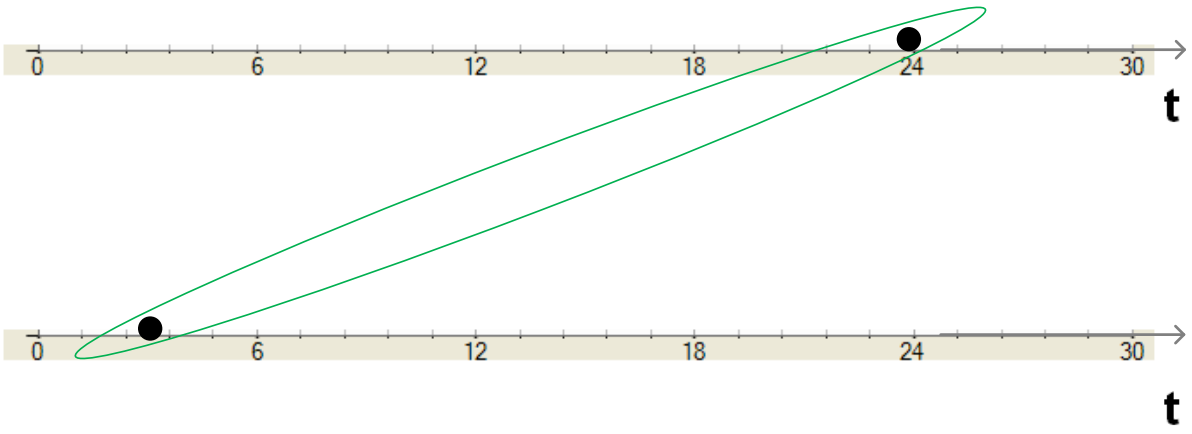
Egy másik ilyen észrevétel szerint azt feltételezzük, hogy egy adott szenzortól származó esemény-szekvenciák sorrendje kötött, nem engedünk meg tehát keresztpárosításokat az azonos azonosítóval rendelkező események között. (Különböző érzékelőtől származó események között megengedett a keresztpárosítás.) Mivel az élekhez a súlyokat az időkülönbségek alapján rendeljük, így ilyen keresztpárosítás akkor fordulhatna elő, ha valamely pontnak nincs párja a közeli környezetében, de az előtte lévő pontnak vagy az utána lévő pontnak van ilyen, és az adott pontnak egy időben messzebbi pontot tudunk párnak rendelni, úgy hogy az összerendelést reprezentáló köztük futó él a már összerendelt párok közti élet keresztezné.




14**. ábra** Párosításban a kereszteződések nem megengedettek

Ezen eseményeknek, ellentétben a szomszédaikkal, valószínűsíthetően azért nincs párjuk a közvetlen környezetükben, mert véletlen zaj jelenségekként kerültek a TSS-ekbe.

A harmadik és utolsó észrevétel, hogy minden eseménynek csak egy rögzített időintervallumon belül keressük a párját; ha azon kívül találnánk párt neki, akkor a párosítás nem jön létre (nincs él a két pont között). Az egyes pontokból tehát csak azon másik ponthalmazbeli pontokba tart él, melyekre teljesül ez a feltétel.




**15. ábra:** Az időben távol eső események között nem lehetséges a párosítás

Ezek után, ezen feltételeket szem előtt tartva, két azonos szenzorazonosítóval rendelkező és különböző TSS-ben megtalálható pont (esemény) között futó él súlya legyen a következő:

$$w = \frac{1 - |\Delta\mu|}{\Delta t^2} \tag{3.1}$$

,ahol w a két pont között futó él súlya, $\Delta t$ a normalizált TSS-ben az adott két esemény időkülönbségének értéke, valamint $\Delta\mu$ a két esemény tagsági függvényértékének különbsége.

A képletből látszik, hogy minél közelebb van időben egymáshoz a két esemény illetve az események tagsági függvényértéke, a köztük futó él súlya annál nagyobb lesz. Az így számított éleknek szükséges meghatározni egy minimális és egy maximális értékét. Egyrészt azért, hogy elkerüljük a képlet szerinti esetleges nullával való osztást. Másrészt, amikor a talált párosításban szereplő élek súlyát összegezzük, kell, legyen egy maximális érték, amihez képest képesek vagyunk viszonyítani a talált párosítás jóságát.

Ezen a módon a két TSS-hez rendelt gráfban kell megkeresni a maximális súlyú párosítást, szem előtt tartva azt, hogy ez a gráf annyi részgráfra osztható, ahány szenzorazonosító szerepel a TSS-ekben. A részgráfokban található maximális súlyú párosítások összege adja a teljes gráfra az eredményt.

Miután megtaláltuk a maximális összsúlyú párosítást, az abban szereplő éleken a következő műveletet hajtjuk végre.

$$c = \frac{\sum w}{TSS_{hossz,min} * w_{max}} \tag{3.2}$$

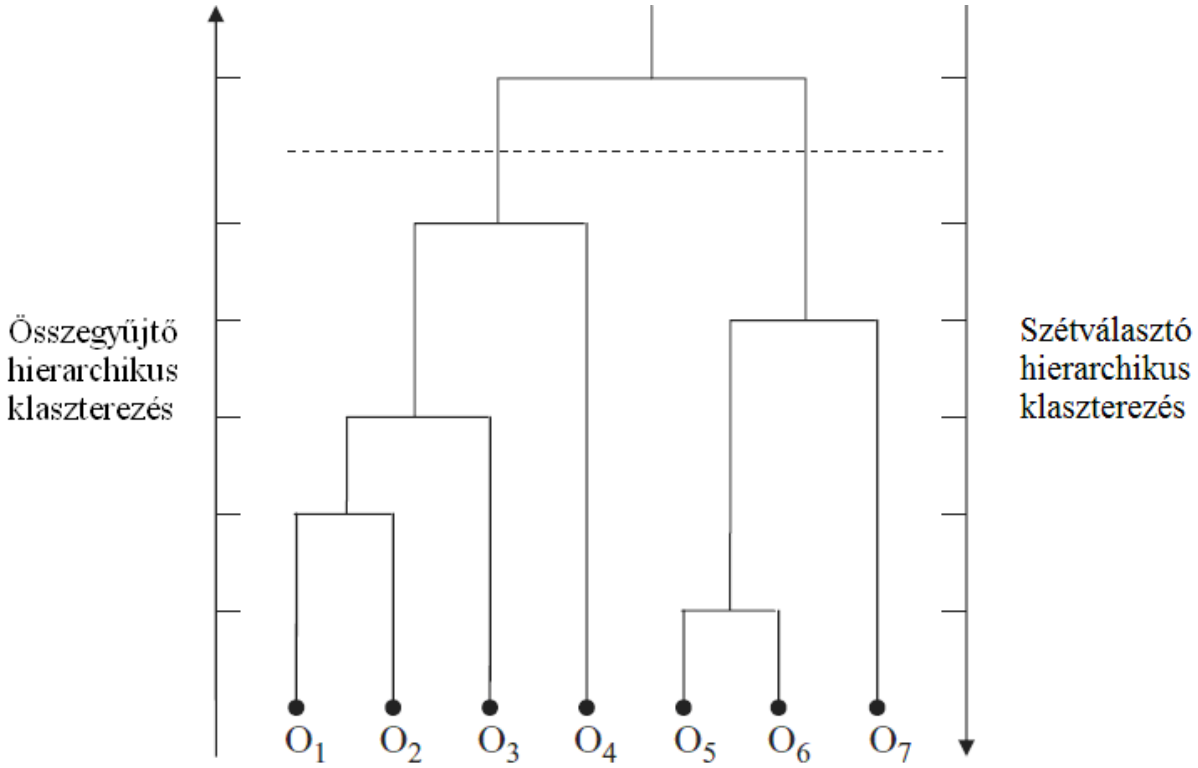
Összegezzük tehát a kapott párosításban az élek súlyait, majd elosztjuk azt a rövidebb TSS elemszám és a lehetséges maximális él-súly szorzatával. Ezzel a művelettel a párosítás jóságát jellemző c paraméter értékét számíthatjuk ki, mely 0 és 1 között változhat. A c értéke 1 ha a rövidebb TSS minden eseményének pontosan megtaláltuk a párját, illetve 0 ha egyik eseménynek sem találtunk párt. Ezek alapján kijelenthető, hogy c a TSS-k „közelségét” jellemzi, így ha távolságot (dist) szeretnénk meghatározni, akkor a következő formulát kell használnunk:

$$dist = 1 - c \tag{3.3}$$

A TSS távolság függvény meghatározásával rendelkezésünkre áll egy olyan módszer, mellyel képesek vagyunk a TSS-ekben foglalt esemény-szekvenciák hasonlóságának jellemzésére. Ennek a módszernek a használatával csoportosítani tudjuk a rendelkezésünkre álló TSS-ek halmazát annak érdekében, hogy az azonos jelenségeket reprezentáló esemény-szekvenciákat (TSS-eket) összerendeljük.

## *3.2 Hierarchikus klaszterezés*

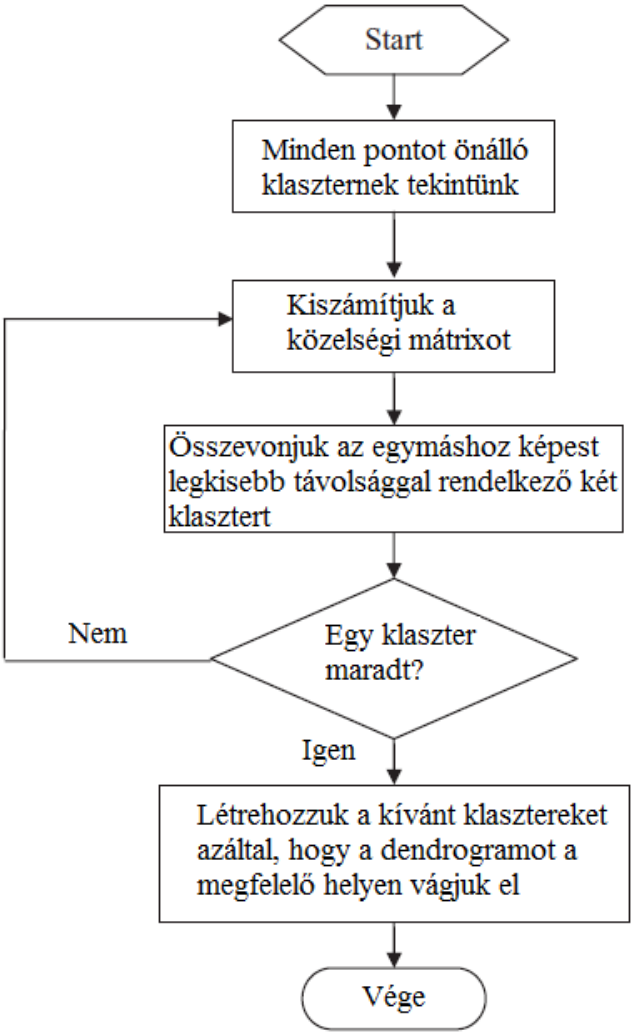
A klaszterező eljárásokat alapvetően kétféle csoportra oszthatjuk, ahogy azt korábban bemutattuk: „hard-partitional" klaszterező eljárásokra és hierarchikusan klaszterező eljárásokra, attól függően, hogy a képzett klaszterek milyen tulajdonságokkal rendelkeznek. A partícionálisan klaszterező eljárások az adott teret alkotó adatobjektumokat közvetlenül, előre meghatározott számú klaszterbe sorolják, mindenfajta hierarchikus rendszer nélkül. A hierarchikusan klaszterező eljárások ezzel szemben egymásba ágyazott klaszterek sorozatába csoportosítják az adatobjektumokat. Az eljárás kezdeti szakaszában lehetnek egyelemű klaszterek, melyeket végül egy minden elemet magába foglaló klaszterbe szervezünk, vagy pont fordítva, a minden elemet magába foglaló klasztert osztjuk fel oly módon, hogy végül csak egyelemű klaszterek maradjanak. Az előbbit összegyűjtő hierarchikus klaszterezési módnak (*agglomerative hierarchical clustering*), míg az utóbbit szétválasztó hierarchikus klaszterezési módnak (*divisive hierarchical clustering*) nevezzük. Mindkét módszer az adatobjektumok közelségi mátrixát használja fel. A hierarchikus klaszterezés eredménye ábrázolható bináris fa vagy dendrogram segítségével. Ez látható a 16. ábran, ahol a nyíl iránya jelzi a klaszterezési folyamat lehetséges haladási irányát.




16**. ábra:** Hierarchikus klaszterezés

A gyökér elem reprezentálja az adott térben előforduló elemeket, és minden egyes levél egy különálló elemet jelent. A közbülső csomópontok leírják az egyes objektumok közelségének mértékét, így a dendrogram szintjeinek magassága megadja, hogy az egyes objektumok vagy klaszterek milyen távol vannak egymástól. A kívánt részletességű klaszterezési eredmények megkaphatóak, ha a megfelelő szinten vágjuk el a dendrogramot. Ez a fajta ábrázolás nagyon informatív képet fest a lehetséges klaszterterezési csoportokról, különösen akkor, ha valódi hierarchikus kapcsolat van az egyes adattagok között, mint például az evolúciós folyamatokat leíró adatoknál.

Az esemény-előrejelző algoritmus az összegyűjtő hierarchikus klaszterezési módszert használja fel, hogy a különböző jelenségeket leíró esemény-szekvenciákat elkülönítse egymástól. A következőkben ismertetem ennek a klaszterezési módnak az egyes lépéseit. Ennek szemléltetésére szolgál a következő ábra:




17. **ábra** Az összegyűjtő hierarchikus klaszterezés folyamatábrája

Kezdetben minden egyes rendelkezésünkre álló adatobjektumot önálló, független klaszternek tekintünk. Kiszámítjuk az így képzett klaszterekre jellemző közelségi mátrixot, majd összevonjuk azt a két klasztert, amelyek egymáshoz képesti távolsága a legkisebb. Megvizsgáljuk a klaszterek számát, és amennyiben az nem egyenlő eggyel, akkor újra számoltatjuk a közelségi mátrixot és megint összevonunk két klasztert. Ha már minden klasztert összevontunk egy klaszterré, akkor nincs más dolgunk, mint a megfelelő helyen elvágni a dendrogramot, hogy megkapjuk az egymástól megfelelően távol eső rész-dendrogrammokat (klasztereket).

A klaszterek összevonása illetve egy új klaszter létrehozása nagyban függ a távolság definíciótól, illetve attól, hogy hogyan adjuk meg az objektum távolság függvényében az objektum-klaszter illetve a klaszter-klaszter közötti távolságot. Amikor a klaszterezés eredményeképpen létrejövő klaszterekről azt valószínűsítjük, hogy azok nagymértékben összefüggőek (kompaktak) lesznek, akkor érdemes a következő képletet alkalmazni:

$$D(C_l,(C_i,C_j)) = \max(D(C_l,C_i),D(C_l,C_j)) \quad \textbf{(3.4)}$$

,ahol $D(\cdot,\cdot)$ a távolság függvény, $C_l$ egy klaszter, és $(C_i,C_j)$ két klaszter összevonása eredményeképpen létrejött klaszter

Amikor összehasonlítunk tehát két klasztert, akkor azok távolságának az egyes elemek közti lehetséges maximális távolságértéket vesszük figyelembe. Ezáltal nem fordulhat elő az a jelenség, ami például a minimumképzéses távolság definícióknál igen gyakori, hogy a klaszterek „elkenődnek”; ebben az esetben a klaszteren belüli elemek egymáshoz képesti távolságai igen nagyok lehetnek.

Térjünk vissza az esemény-előrejelző algoritmushoz. Adott a TSS távolság definíciója, illetve adott az imént tárgyalt klaszterezési módszer. Ezek segítségével hierarchikus klaszterfába rendezzük a rendelkezésünkre álló TSS-eket, majd a megfelelő helyeken elvágva ezt a dendrogramot, a keletkező részdendrogramokból nyerjük ki a lehetséges tiszta (zaj nélküli) szekvenciákat. A megfelelő helyek megtalálása úgy történik, hogy bejárjuk az egész dendrogramot egy „bináris fa bejárás algoritmussal”, és ahol az egyes közbülső elemek alatt közvetlenül elhelyezkedő szomszédos elemek távolságának, és ezen szomszédos elemek alatti átlagos távolságoknak a különbsége meghalad egy bizonyos paraméter értéket, ott elvágjuk a klaszterfát. Tehát:

$$\text{param} \le \text{node(dist)} - \frac{\text{node}_{\text{jobb}}(\text{average\_dist}) + \text{node}_{\text{bal}}(\text{average\_dist})}{2} \quad \textbf{(3.5)}$$

,ahol $\text{node}$: egy tetszőleges közbülső elem;

$\text{node}_{\text{jobb}}$: az adott közbülső elem jobb oldali részfájának gyökér eleme;

$\text{node}_{\text{bal}}$: az közbülső elem bal oldali részfájának gyökér eleme;

$node(dist)$: az adott elem alatt közvetlenül elhelyezkedő $node_{jobb}$ és $node_{bal}$ távolsága;

$node_{bal}(average\_dist)$: az adott elem baloldali részfája ágainak átlagos távolsága;

$node_{jobb}(average\_dist)$: az adott elem jobb oldali részfája ágainak átlagos távolsága.

Ezek után, az így keletkező klaszter-csoportokban lefuttatjuk a szekvencia kinyerő függvényt, mely az összerendelt TSS-ekben keresi az azonos esemény-szekvenciákat. A talált esemény-szekvenciák valószínűsíthetően a keresett tiszta szekvenciák lesznek.

## *3.3 Tiszta szekvenciák kinyerése*

A hierarchikus klaszterezési eljárások hátrányai közé tartozik, hogy kevésbé robosztusak, valamint érzékenyek a zajra illetve a klasztercsoportoktól távol eső magában álló adat-objektumokra. Amennyiben egy adat-objektumot egy klasztercsoportba soroltunk, a későbbiek folyamán nem lesz önálló objektumként figyelembe véve, ami azt jelenti, hogy a hierarchikus klaszterezés nem képes korrigálni a korábbi téves osztályozási lépéseit. Másik nagy hátránya ennek a klaszterezési eljárásnak a magas számításigénye, ami legalább O ($n^2$) komplexitású, így a hierarchikus klaszterezés nem képes megfelelő hatékonysággal kezelni nagy számosságú adathalmazokat. Mindezek ellenére rendelkezik egy rendkívül előnyös tulajdonsággal, nevezetesen hogy az ideális csoportosítás érdekében a klaszterezési folyamat kezdetekor nem kell a felhasználónak tudatában lenni a valós klasztercsoportok számának, és azt bemeneti paraméterként megadnia, ugyanis az könnyen becsülhető a klaszterezés eredményét reprezentáló dendogramból.

Abban az esetben, ha az adathalmazban ténylegesen jelenlévő klasztercsoportok száma egy szűk tartományon belül változhat, akkor esetleg célszerűbb lehet másfajta klaszterezési eljárást alkalmazni.

### 3.3.1 Partícionáló klaszterezés

A partícionálisan klaszterező eljárások az adott teret alkotó adatobjektumokat közvetlenül, előre meghatározott számú klaszterekbe sorolják, mindenfajta hierarchikus rendszer nélkül. Ez a folyamat általában együtt jár egy kritérium függvény-optimalizálással. Részletesebben,

adott egy $x_i \in R^d$, $i = 1,...,N$ adathalmaz, a klaszterező eljárás megkísérli K darab klasztercsoportba $\{C_1,...,C_K\}$ sorolni ezeket, oly módon hogy egy előre meghatározott J kritérium függvényt minimalizálni vagy maximalizálni igyekszik. A kritérium függvény értéke teszi lehetővé az egyes csoportosítások „jóságának" összehasonlítását, vagyis hogy mennyire esnek közel az ideális megoldáshoz.

A K-mean klaszterezési eljárás az egyik legismertebb és legnépszerűbb partícionáló módszerek egyike. Adott objektum-halmazban keresi az ideális partícionálást iteratív optimalizáló lépéseket alkalmazva, mégpedig oly módon, hogy minimalizálni igyekszik a következő kritérium függvényt („négyzetes hibák összege"):

$$J_s(\Gamma, M) = \sum_{i=1}^{K} \sum_{j=1}^{N} \gamma_{ij} \| x_j - m_j \|^2$$

ahol:

$\Gamma = \{\gamma\}_{ij}$ az úgynevezett partíció mátrix, amiben: $\gamma_{ij} = 1$ ha $x_j \in C_i$, egyébként 0, ($\sum_{i=1}^{K} \gamma_j = 1 \ \forall j$)

$M = [m_1,....m_k]$ klasztercsoportok prototípus vagy súlypont mátrixa, $m_i = \frac{1}{N_i} \sum_{j=1}^{N} \gamma_{ij} x_j$

Tehát a fenti kifejezés összegzi minden pont esetében azt a távolságot, ami az adott pont és a pont klasztercsoportjának súlypontja között mérhető. Az a partícionálás, amelynél fenti kritérium-függvény a legkisebb értéket veszi fel, optimálisnak tekinthető és legkisebb szórású partíciónak (*minimum variance partition*) is nevezik. A „négyzetes hibák összege" kritérium függvény megfelelő megoldást adhat, ha a felderítendő klasztercsoportok kompaktak és kellően elkülönülnek egymástól.

18. ábra Kép szegmentálása k-mean klaszterezéssel 2,4,8 szegmensre

A K-mean klaszterezést gyakran használják a digitális képfeldolgozás során a pixel pontok szegmentálásához. A szegmentálás eredményeképpen adódó új képben könnyebb a képen szereplő tárgyak, élőlények határvonalainak azonosítása, ezáltal azok azonosítása. Gyakran az RGB pixel alapszínek által kifeszített térben az Euklidészi távolságot használják a pixel pontok elkülönítésére.

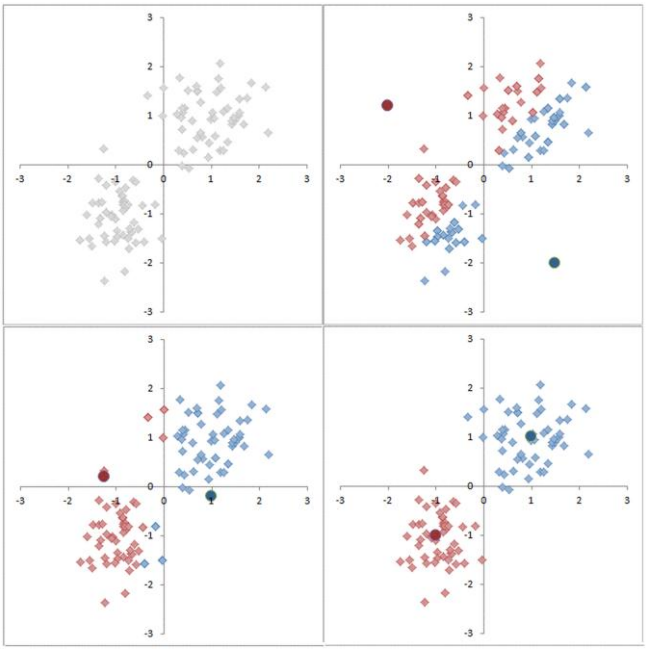
A K-mean klaszterezés lépései a következőek:

1. K darab partíció inicializálása véletlenszerűen vagy valamiféle heurisztika alapján. A klasztercsoportok súlypont mátrixának kiszámítása $M = [m_1, ....m_K]$
2. Minden egyes adatobjektumot hozzárendeljük a legközelebbi $C_l$ klasztercsoporthoz, a következőek szerint: $x_j \in C_l$, ha $\| x_j - m_l \| < \| x_j - m_i \|$

   $j = 1,..., N$, $j \neq l$ és $i = 1,..., K$
3. Újra számoltatjuk a klasztercsoportok súlypont mátrixát a 2.-es pontbeli hozzárendelések alapján

$$m_i = \frac{1}{N_i} \sum_{x_j \in C_i} x_j$$

4. A 2-es és 3-as lépeseket ismételtetjük, amíg már nincs változás az egyes csoportokban.



19. ábra A K-mean klaszterezés lépéseinek szemléltetése

A fentebb leírt lépéseket szemlélteti a 19. ábra. A bemeneti kétdimenziós adathalmazhoz tetszőlegesen felveszünk két klaszter középpontot (súlypontot). Az egyes elemeket hozzárendeljük valamelyik csoporthoz, attól függően, hogy az előre meghatározott távolság definíció szerint melyik középpontjához van közelebb az adott pont. Miután minden pontot besoroltunk valamelyik csoportba, újra számoljuk a klaszter középpontokat a besorolás alapján. Az utóbbi két lépést ismételjük, míg a középpontok helyzete már nem változik.

Érdemes megemlíteni, hogy a módszer által talált megoldás nem biztos, hogy megegyezik a globális optimummal, ezért célszerű lehet többször lefuttatni az algoritmust. (A kezdeti partíciók véletlenszerűen kerülnek inicializálásra.)

### 3.3.2 A szekvencia kinyerő függvény

Adottak a hierarchikus klaszterezés eredményét reprezentáló dendogram szétvágása után kialakuló klasztercsoportok. Minden ilyen csoportban összerendelt TSS-ek találhatóak. A szekvencia kinyerő függvény bemenete egy ilyen klasztercsoport, amely különböző számú normalizált TSS-t tartalmazhat. Az egyes TSS-ekből kiszedjük az azonos szenzor-

azonosítóval rendelkező eseményeket és egy, az adott szenzorazonosítóhoz rendelt tömbbe tesszük ezeket. Ezzel a lépéssel tulajdonképpen a 3 dimenziójú esemény (sensorID, idő, $\mu$) adatstruktúrákat a szerint csoportosítjuk, hogy melyek helyezkednek el azonos síkban a sensorID dimenzió mentén. Ezek után az adott síkokra alkalmazzuk a fent vázolt k-mean klasztrezési eljárást, ahol az x,z koordináták helyett az idő (t) és a tagsági függvény értékek ($\mu$) szerepelnek, a következőek szerint:

Adottak a következő paraméterek (a felhasználó definiálja):

- *number_of_repeats*
- *require_percentage*
- *threshold_variance*

Veszünk egy egész típusú „k” változót, mely az adott iterációban a klasztercsoportok számát jelöli. Kezdeti értéke legyen 2. Az adott iterációban a „numbe_of_repeats” paraméter értékének megfelelő ismétlésszer hívódik meg a k-mean klaszterezés az adott szenzorazonosítóhoz rendelt esemény tömbben. (Ez a paraméter minél nagyobb, annál nagyobb valószínűséggel találjuk meg a globális optimumot.) Minden ilyen ismétléskor összegezzük a kialakuló klasztercsoportok szórását. A legkisebb szórású megoldást elmentjük, míg nem találunk esetleg még jobb megoldást. Ezek után megvizsgáljuk, hogy a kapott klasztercsoportok szórása kisebb e, mint a „threshold_variance” paraméter értéke, ha találunk olyat, melynél ez a feltétel teljesül, akkor azt is megvizsgáljuk, hogy az adott esemény-tömben található események száma hogyan viszonyul az ebbe a klasztercsoportba sorolt események számához („require_percentage”). Ha mindkét feltétel teljesül, akkor az adott klasztercsoport súlypontját a tiszta szekvencia részének tekintjük és elmentjük. Ha valamelyik feltétel nem teljesül, akkor a „k” értékét növeljük eggyel és előlről kezdjük az iterációt. Amennyiben a tömb elemszáma = n, és k értéke eléri a ”n * (1-require_percentage)+1” értéket, akkor kilépünk az iterációból, és másik szenzorazonosítóhoz tartozó tömbre térünk át, míg van ilyen.

# 4 Szenzorhálózat szimulátor

Ebben a fejezetben ismertetem az esemény-előrejelző algoritmus tesztelése céljából implementált szenzorhálózat szimulátort. Mivel a szimulátor C# nyelven és .NET

környezetben íródott, ezért egy alfejezetben röviden bemutatom a programnyelvet és a futási környezetet. Ezt követően a szimulátor felépítéséről, tervezési szempontjairól és működéséről írok, végül a tesztelés során szerzett tapasztalataimat osztom meg az olvasóval.

## *4.1 A C# és a .NET rövid bemutatása*

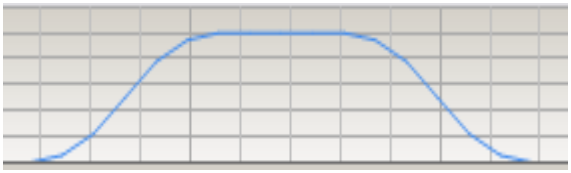
A C# egy tisztán objektumorientált programozási nyelv, melyet C++ és Java alapokon fejlesztettek ki. Támogatja a komponens alapú és a többrétegű alkalmazásfejlesztést is. A C# platform-független, amit a .NET-es futtatókörnyezet biztosít. Egyik fontos tulajdonsága, hogy a nyelv fordítása során egy úgynevezett *kezelt kódot* kapunk. Ez a magasszintű (C#) és az alapszintű (gépi kód) között helyezkedik el. A *közös nyelvű futási idejű környezet* (Common Language Runtime - CLR) futási időben, menet közben fordítja le a kódot a Just-in-Time (JIT) fordító alkalmazásával. Ennek az az előnye, hogy csökken az alkalmazás munkahalmaza, hiszen a közbenső kód memóriaigénye kisebb lesz. Az alkalmazás futása közben a JIT csak a szükséges kódot fordítja le; ha például a kódunk tartalmaz egy olyan függvényt, amit egyszer sem hívunk meg, akkor annak a kódrészletnek a lefordítására nincs szükség. Ezenfelül a CLR képes menet közben optimalizálni a program futását. Összességében elmondható, hogy a JIT technika ötvözi az automatikus helyben futtatást a lefordított kód gyors végrehajthatóságával.

A .NET keretrendszer olyan futtatási környezet, amely lehetővé teszi, hogy a programozók könnyen írhassanak robosztus programokat. Az alkalmazásoknak olyan szolgáltatásokat nyújt, mint például az automatikus memóriakezelés, a rendszerszolgáltatások egyszerűbb elérése, vagy például az egyszerű internet- és adatbázis-hozzáférés.

## *4.2 Szimulációs adatok beolvasása*

A szimulátor tetszőleges számú szenzort képes szimulálni, köszönhetően annak, hogy a szenzorok osztályként vannak reprezentálva. Az egyes szimulációk során a szenzor osztályból annyi példány fog készülni, ahány szenzort megkövetel az adott szimuláció. A szimulációk előtt a szimulátor egy bemeneti fájlból olvassa be az egyes szenzorok által érzékelt jelenségek paramétereit. Ezek a beolvasott adatok rögzítik, hogy az egyes szenzorok a szimuláció folyamán milyen eseményeket érzékelhetnek a mintavételi frekvenciájuktól függően az idő előre haladtával.

Jelenleg az események harang (dupla szigmoid) alakú tagsági függvényekkel vannak reprezentálva:



**20. ábra:** Az események egy lehetséges fuzzy halmazának alakja

Ilyen tagsági függvénnyel jellemezhetünk egy környezeti zajt mintavételező szenzor által érzékelt hanghatásokat. Ha egy autó halad el a szenzor mellett, többé-kevésbé elmosódottan szeretnénk definiálni azt a spektrális tartományt, ami jellemzi a detektálandó motorzajt. Minél távolabb vagyunk a referencia tartománytól, annál kisebb mértékben tartozik bele a detektált zaj a definiált specifikus motorzaj események fuzzy halmazába.

A szimulátor bemenete egy *bemenet.txt* fájl, melynek formátuma a következő:

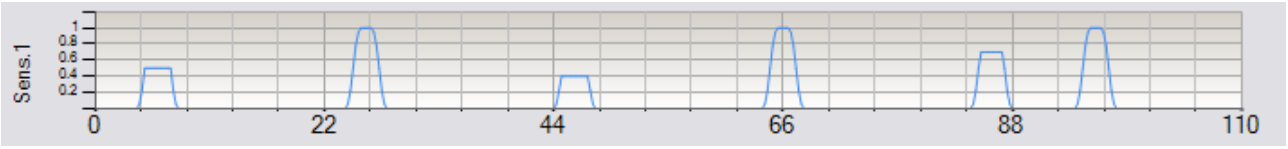
*Patterns{sensorID}=[t1 u1;t2 u2; .....; tn un];*

ahol *SensorID*: az adott szenzor azonosítója;
*t1, t2,...., tn* : az eseményeket szimbolizáló harang alakú tagsági függvények helyzete az időtengelyen;
*u1, u2..., un:* az adott harang alakú tagsági függvényekhez tartozó maximális érték.

A szimulátor következőképpen értelmezi ezt a bemeneti sort:

*Patterns{1}=[6 0.5 ; 26 1 ; 46 0.4 ; 66 1 ; 86 0.7 ; 96 1];*

Először is példányosít egyet a szenzor osztályból, melynek a szenzor azonosítója az 1-es lesz. Majd az alábbi ábrán látható esemény-szekvenciát rendeli ehhez a szenzorhoz:



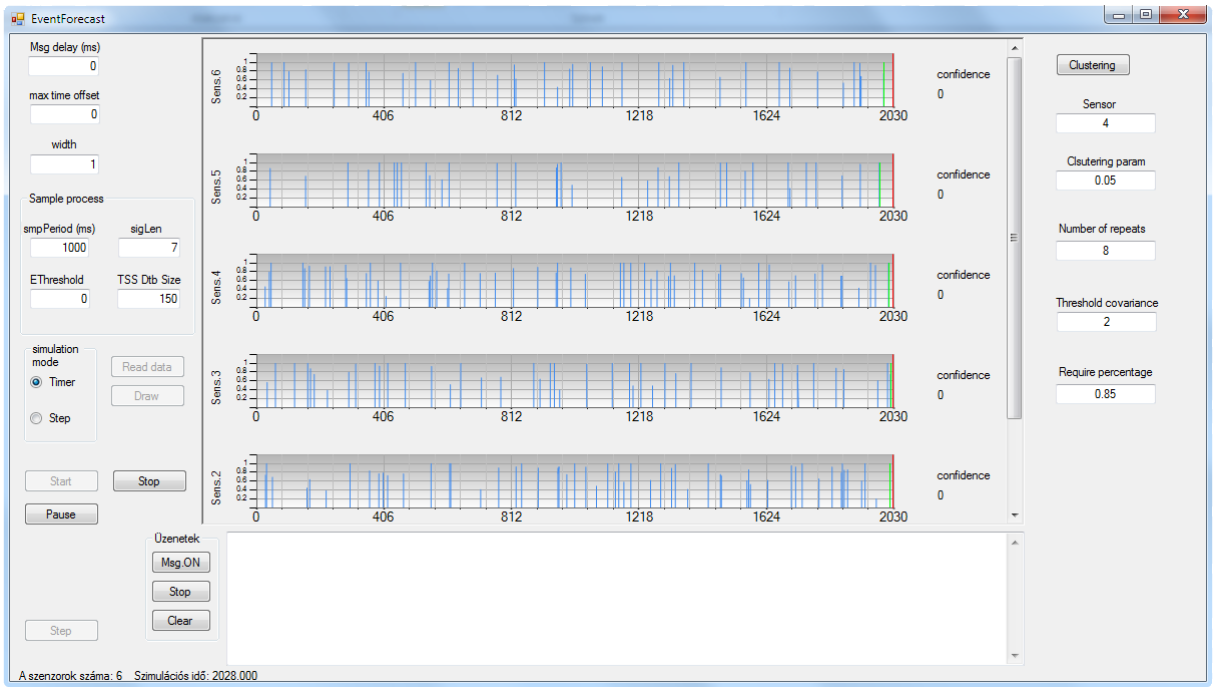

21. **ábra:** Egy szenzor lehetséges esemény-szekvenciája

A 12. ábrán látszik hogy a *t1, t2,....., tn* adatok a tagsági függvények közepét jelölik az idő tengely mentén, míg az *u1, u2,..., un* értékek ezen tagsági függvények maximumát határozzák meg.

Ha az adatok beolvasása során a szimulátor hibás formátumú sorokat talál vagy ismétlődő szenzor azonosítót, akkor az adott sort átugorja. A szimulátor annyi szenzort fog szimulálni futása során, ahány megfelelő formátumú sort sikeresen beolvas.

## *4.3 A felhasználói felület*

A szimulátor működése a felhasználói felületen keresztül követhető figyelemmel, és azon keresztül befolyásolható is. Különböző paraméterek állíthatóak ezen a felületen, mint például a mintavételi frekvencia értéke, a szenzorok egymásnak küldött üzeneteinek a késleltetése, vagy akár a szenzorok időszinkronitás-eltérésének maximális értéke. A felhasználói interfész a 13. ábrán látható.




22. **ábra:** Felhasználói interfész

A felület bal felső részében látható *Msg delay* mezőben beállított érték határozza meg a szenzorok közötti üzenetek késleltetését, milliszekundumban. A *max time offset* mező értéke a szenzorok idejének egymáshoz képesti elcsúszását befolyásolja. A *width* paraméter

segítségével az események harang alakú tagsági függvényének szélessége szabályozható. A *sigLen* szövegdobozban megadott érték lesz a TSS-ek és a szenzorok FIFO-jának maximális hossza, az *Ethreshold* mezőben közölt érték a mintavételezés során a küszöbértéket állítja be, míg az *smPeriod* a mintavételezési időközt adja meg milliszekundumban. A *Read data* gombra kattintva kerülnek beolvasásra a *bement.txt* fájlban lévő adatok, majd a *Draw* gombot használva kirajzolódnak az egyes szenzorok esemény-szekvenciái. A szimulátornak két fajta működési módja van. *Timer* üzemmódban a szimuláció magától fut, a felhasználó csak az interfész alsó részében található szövegdobozban megjelenő rövid üzentekből értesül az éppen zajló eseményekről. Ezzel szemben *Step* üzemmódban a felhasználó maga lépteti eseményről eseményre a szimulátor ütemezőjét. A szimuláció minden esetben a *Start* gomb lenyomásával indul, és a *Stop* gomb lenyomásával ér véget. A működés szüneteltethető a *Pause* gomb segítségével. Abban az esetben, ha a szimuláció elérte az előre kijelölt időtartam végét, az ütemező működése leáll.

## *4.4 A szimulátor működése*

A szimulátor megfelelő működéséért az úgynevezett ütemező felel. Az ütemező célja, hogy a szimulátor elszakadjon a fizikai időtől, és csak a szimulációs időtől függjön. Erre azért van szükség, mert kellően nagyszámú szenzor szimulálása esetén előfordulhatna, hogy a szimulátor egy adott időköz alatt nem lenne képes minden, az adott idő intervallumra kiírt feladatát elvégezni. Működése a következőképpen néz ki:

```
1:  tsim=10*smpPeriod;
2:  actSenosr=null;
3:        for(i=1..n)
4:        if(Sensors[i].Nextevent_time() <=tsim)
5:            if(Sensors[i].Nextevent_time() = tsim && Sensors[i].Nextevent_op>Operation)
6:                actSensor=Sensors[i];
7:                Operation= Sensors[i].Nextevent_op;
8:            else
9:                tsim= Sensors[i ].Nextevent_time();
10:               actSensor=Sensors[i];
11:               Operation= Sensors[i].Nextevent_op;
12:       endfor
13:  perform(actSensor, Operation, tsim);
14:  actSensor.update_nextevnt_time();
```

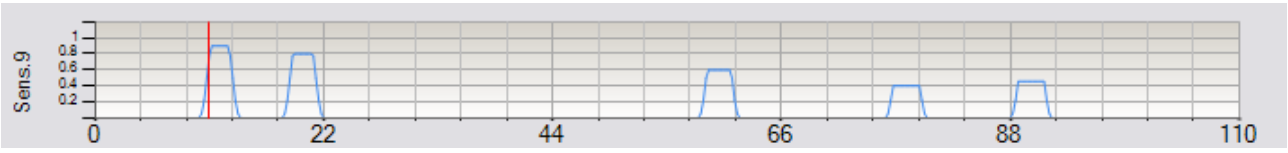
Első lépésként tehát a szimulációs időt a mintavételezési idő tízszeresére állítjuk be, majd a soron következő aktuális szenzort jelölő *actSensor* pointert *null*-ra inicializáljuk. Ezután a

*bemenet.txt* fájl alapján a példányosított *Sensor* osztályú elemeket tartalmazó *Sensors* hash táblán végigmegyünk egyenként, és lekérdezzük az adott szenzortól a következő olyan időpontot, amikor valamilyen feladata van. Ha ez kisebb, mint a jelenlegi szimulációs időpont, akkor a szimulációs időt beállítjuk erre az értékre; az *actSensor* az adott példányosított szenzor osztály lesz, míg a művelet prioritását (*Operation*) egyenlővé tesszük ennek a műveletnek a prioritásával. Abban az esetben, ha a jelenlegi szimulációs időpont megegyezik az éppen vizsgált szenzor következő műveletének idejével, akkor csak a műveletek prioritását vizsgáljuk, és a nagyobb prioritású műveletet állítjuk be az ütemező következő elvégzendő feladataként.

Az ütemező aktuális feladatáról a felhasználó interfész alsó részében elhelyezkedő szövegdobozban kiírt rövid szöveges üzenetekből értesülhetünk. Információt kapunk az aktuális feladat idejéről és mivoltáról is.


23. **ábra:** Az ütemező feladatáról informáló szövegdoboz

Egy adott szenzor mintavételezését az esemény-szekvenciáját jelölő grafikonon léptetett, piros színű kurzor is jelzi. Ez látszik a 15. ábrán.




24. **ábra:** A szenzorok esemény-sorozatát jelölő grafikon mintavételezéskor

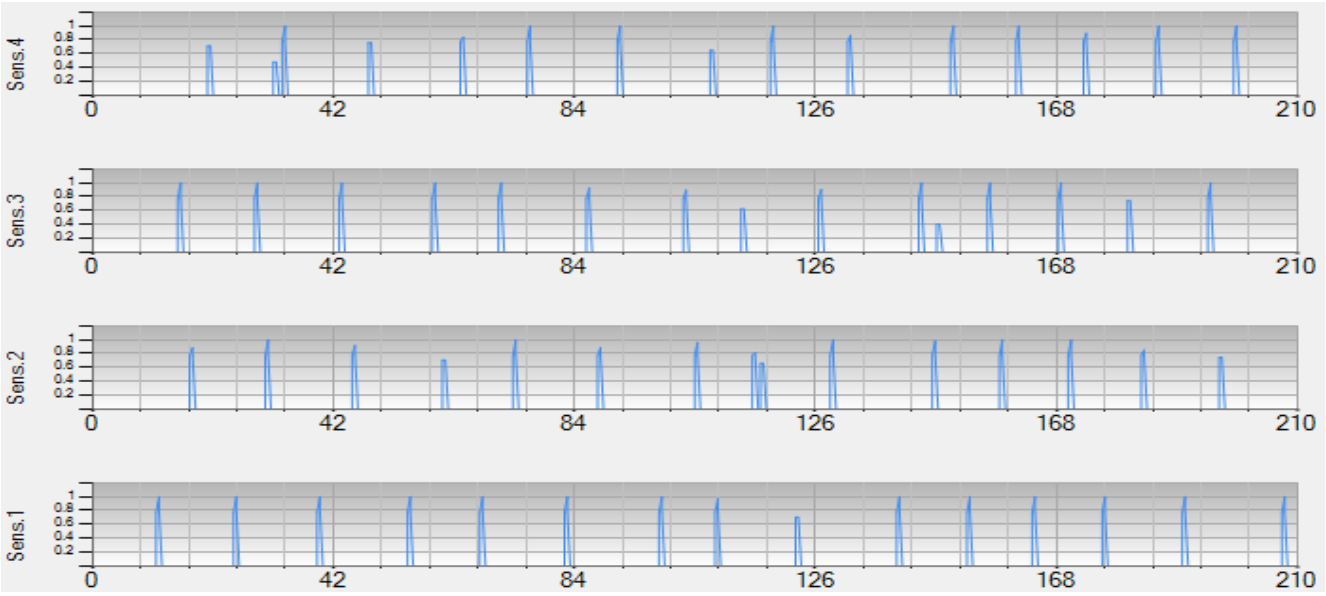
A piros kurzor pillanatnyi helyzetének léptetése azt is jelzi, hogy az egyes szenzorok az általuk érzékelhető jelenségeket miként látják, tehát az egyes jelenségek harang alakú tagsági függvényeit hányszor és mely pillanatban mintavételezik.

## 4.5 *Az algoritmus tesztelése a szimulátorban*

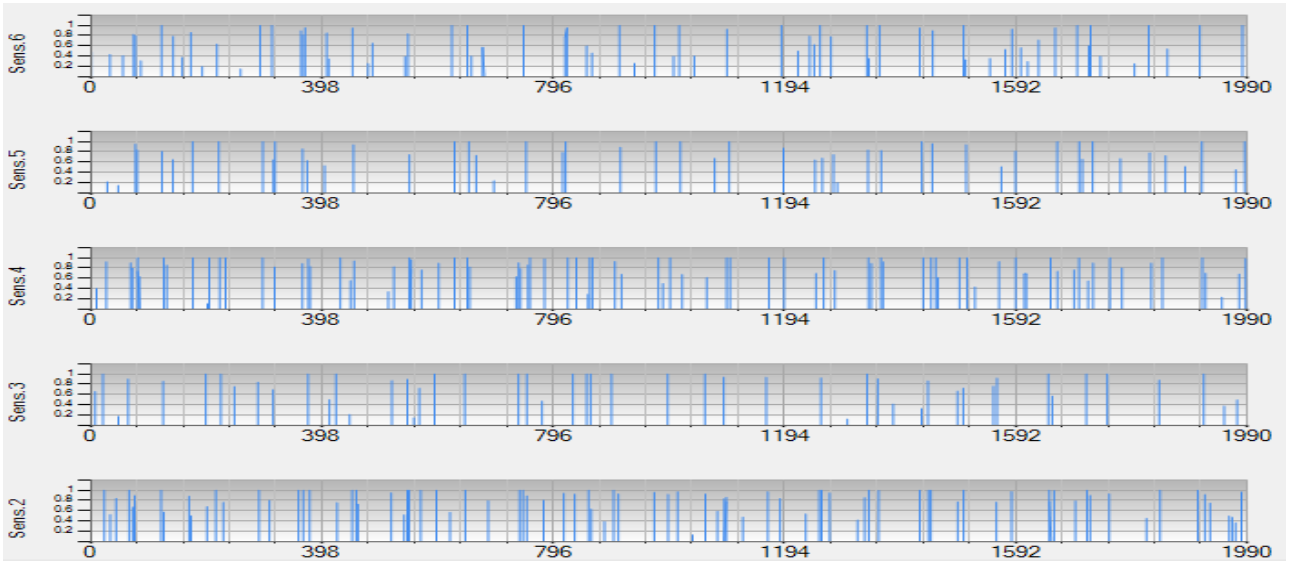
Az algoritmus hatékonyságának teszteléséhez különböző paraméterekkel rendelkező bemeneti fájlokat használtam. A teszt fájlok generálása a következők szerint történt: elsőként meg kellett határozni az ismétlődő esemény-szekvenciát (tiszta szekvencia). Ezt követően a meghatározott esemény-szekvenciát sorozatosan újra generáltuk az időben előre haladva oly módon, hogy az egyes szekvencia-kezdések között egyenletes eloszlású véletlen időtartam teljen el. Ezt addig folytattuk, míg el nem értük a kívánt szimulációs idő végét. Az egyes szekvenciákhoz még egy normál eloszlású idő-jittert és egy $\mu$-jitttert is hozzáadtunk. Ezek után következett a zaj események hozzáadása a szekvenciákhoz. A zaj események $\lambda$ paraméterű exponenciális eloszlás szerint követték egymást.

A szimulátorban futó algoritmus feladata az volt, hogy az imént felvázolt szisztéma szerint képzett esemény-sorozatokból (bemeneti file-ból) kiszűrje a tiszta szekvenciákat. A keresett szekvenciák gyakran átlapolódtak egymással, olykor önmagukkal is.

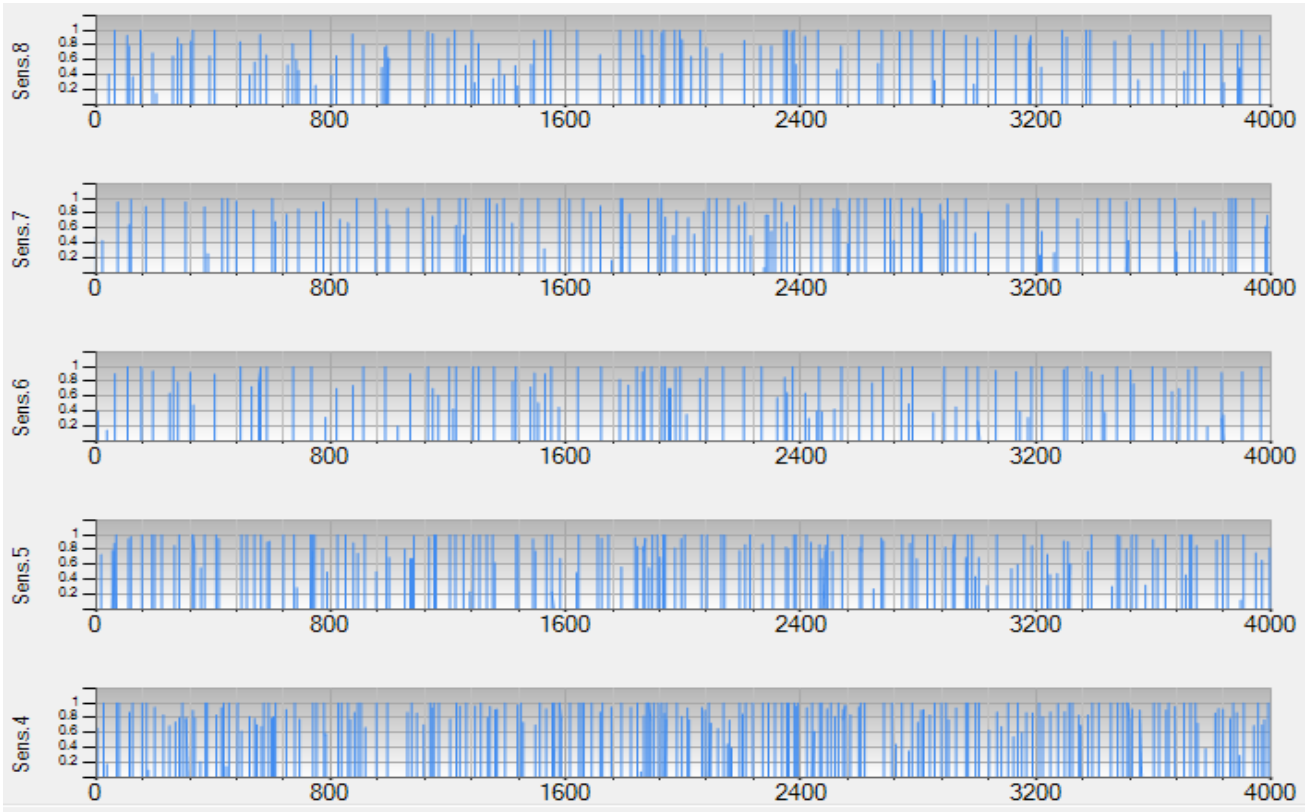
A következő ábrákon szeretném szemléltetni a különböző mértékben zajos bemeneti tesztfájlokat.




25**. ábra: A** legkevésbé zajos szimuláció




26**. ábra:** Közepesen zajos szimuláció




27**. ábra: A** leginkább zajos szimuláció

Három szempont szerint kívánom bemutatni az algoritmus hatékonyságának vizsgálatát. Először a tiszta szekvenciákhoz hozzáadott zaj események sűrűségének, majd a maximális TSS hossz változásának, végül a tiszta szekvenciák eseményeihez hozzáadott idő-jitter függvényében fogom a megtalált eseményeket kiértékelni.

Két tiszta szekvenciát tartalmaztak a tesztfájlok, ezek minden 100 másodpercben egyenletes eloszlás szerint kezdődtek újra. Az első szekvencia eseményei között átlagosan 1.5 s volt, míg a második eseményei között 4s. A szimulációs idő 2000 másodperc volt minden esetben. A tiszta szekvenciák idő-jitterének szórása $Jt_\sigma = 0.3$, a $\mu$-jitter szórása $J\mu_\sigma = 0.2$, ezek várható értéke pedig 0 volt.

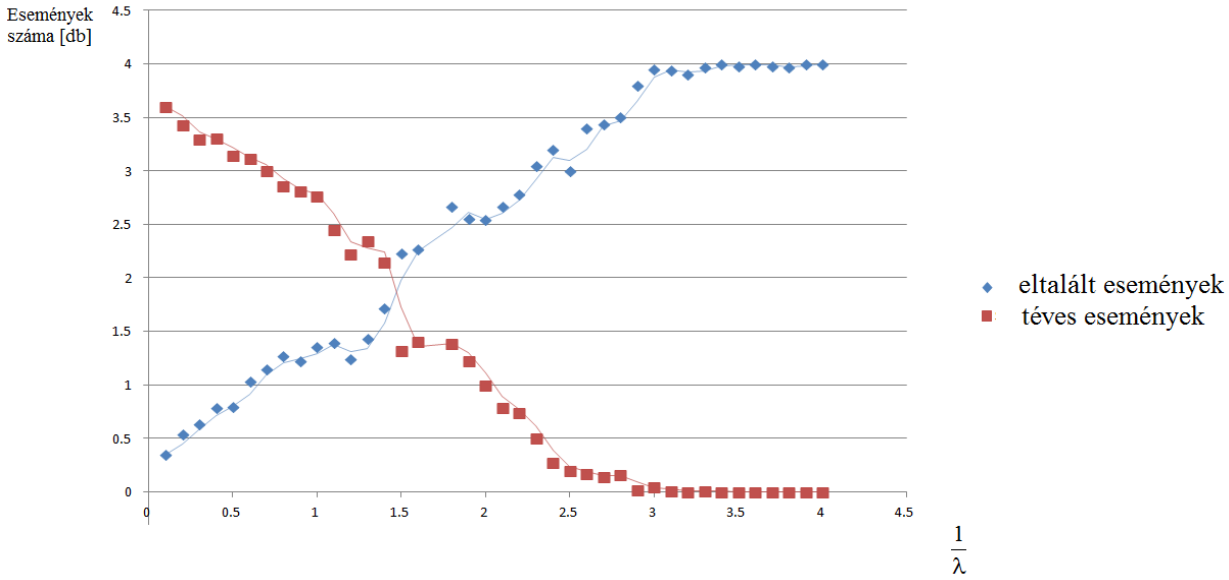
$\lambda$-paraméterű exponenciális eloszlás szerint követték egymást a zaj események, tehát két zaj esemény közötti időtartam várható értéke $\frac{1}{\lambda}$ volt. A zaj események $\mu$-beli értékeit normális eloszlás szerint vettem fel, melynek várható értéke $m_\mu = 0.5$, illetve szórása $\sigma_\mu = 0.2$ volt.

Az algoritmus paraméterei a szimulációk során:

- TSS hossz=10
- mintavételi frekvencia = 1000 ms
- EThreshold = 0
- clustering parameter = 0.05
- number_of_repeats = 8
- threshold_variance = 2
- require_percentage = 0.85

Az algoritmus érzékenysége a zaj események időbeli sűrűségére a 28. ábrán látható. A képen látható pontok mindegyik 10 szimuláció és azok eredményének átlagát reprezentálják. A kék pontok jelölik normálva, hogy az eredményül kapott események közül hány darab egyezett

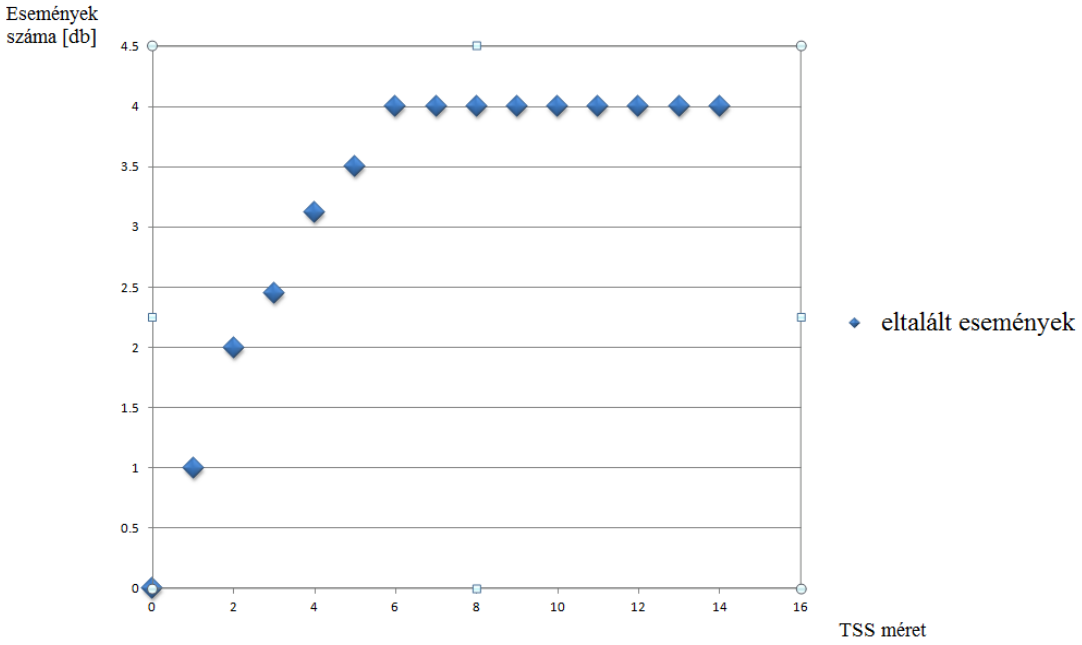
meg a 4 elemű tiszta szekvenciákkal. Tehát szimulációk végén vettük az eltalált események számát, és elosztottuk az összes eredményül kapott esemény számával. Ezzel szemben a piros pontok jelölik a tévesen a tiszta szekvenciák közé vett események számát normálva.




28. ábra Az algoritmus érzékenysége a zaj események időbeli sűrűségére

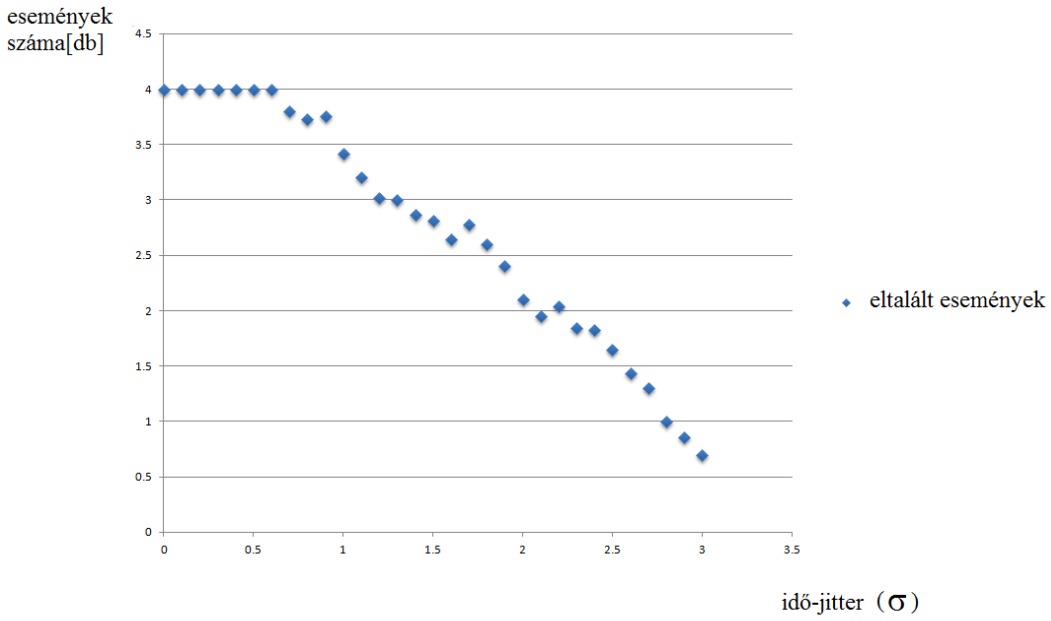
Azt mondhatjuk az ábra alapján, hogy az algoritmus jól működött, míg a zaj események közötti idő várható érték le nem csökkent 3 alá, vagyis körülbelül a tiszta szekvenciák eseményei közötti átlagos időértékre.

A vizsgálódás következő tárgya az volt, hogy a szenzorok által tárolt események-szekvenciák maximális száma (maximális TSS méret), hogyan befolyásolja az algoritmus hatékonyságát. A 29. ábra ezt szemlélteti. Az eredmények azt mutatják, hogy adott zajtényező mellett az algoritmus a 6 vagy annál nagyobb TSS méreteknél működött hibátlanul.




29. ábra Az algoritmus hatékonyságának függése a TSS mérettől

Utolsó vizsgálódási pontunk az eltalált események számának változása az idő-jitter változásának függvényében. Ez látható a 30. ábrán.




30. ábra Az eltalált események az idő-jitter függvényében

Tapasztalatként itt azt mondhatjuk el, hogy ha az idő-jitter szórása kisebb vagy egyenlő volt az előzetesen inicializált szekvencia kinyerő függvény threshold_variance paraméter értékével, akkor az algoritmus megfelelően működött.

# 5 Közlekedési csomópontokban mért adatokon való tesztelés

Az előző fejezetekben bemutatott esemény-előrejelző algoritmust szerettem volna valós körülmények között is tesztelni, ennek érdekében pedig valós adatokon kívántam tesztelni az eljárást. Ennek okán implementálásra került egy TinyOS operációs rendszert futtató, a Crossbow cég által gyártott valós MicaZ szenzorokból álló rendszer, mellyel különféle forgalmú és típusú kereszteződésekben haladó gépjárművek által generált eseményeket rögzítettünk. A mért események a szimulátor bemenetét képezték, így lehetőség nyílt az algoritmus vizsgálatára valós környezetben mért mintákon

A soron következő alfejezetben a teszteléshez használt Crossbow cég által gyártott MICAz mote-okat és a rajtuk futó TinyOS operációs rendszert ismertetem, majd kitérek a méréseket lebonyolító rendszer működésére, a mérések lebonyolítására, végül a mért adatokon való teszelelésre.

## *5.1 A MICAz mote*

*Mote-nak* nevezzük az olyan kisméretű intelligens kommunikációs eszközt, mely a szenzorhálózatok alapvető építőeleme. Ezen eszközök rendelkeznek kis teljesítményű processzorral, saját memóriával és rádiós kommunikációhoz szükséges interfésszel. Ha a mote-hoz egy érzékelőkártya is csatlakozik, akkor szenzornak nevezzük. Egy szenzorhálózat tehát szenzorokból és mote-okból épül fel; a hálózat fontos elemeit képezik tehát azok a mote-ok is, melyek bár nem képesek önmagukban érzékelésre, részt vesznek a többi szenzor által mért adatok továbbításában a bázisállomás felé.

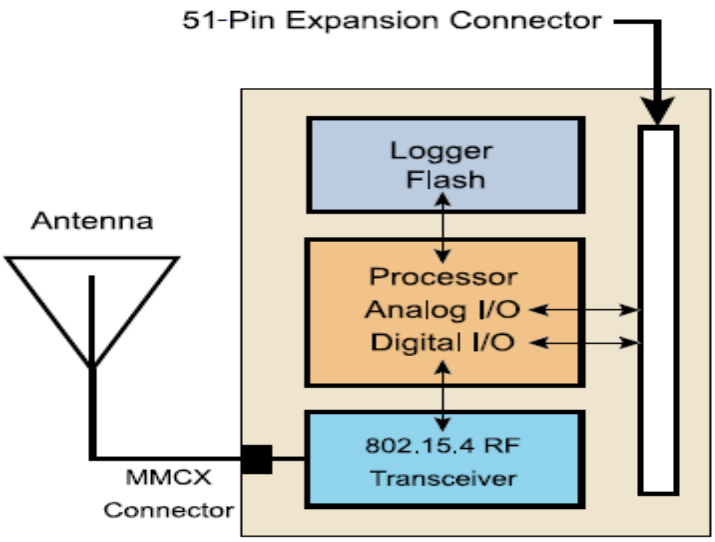
Ebben a fejezetben bemutatom a MICAz *mote* felépítését és tulajdonságait, illetve a használatához elengedhetetlen mib520-as *programozói kártyát* és az úgynevezett *érzékelőkártyákat* (*sensorboard*). Egy ilyen mote látható az 19. ábrán.

31**. ábra:** Egy MICAz mote

A MICAz mote jellemzői:

- Az IEEE 802.15.4–es szabvány szerint kommunikál a 2.4 GHz-es ISM frekvenciasávban;
- A maximális átviteli sebessége 250 kbps;
- A mote-on a TinyOs operációs rendszer fut;
- Külön csatlakoztathatóak hozzá az *érzékelőkártyák (sensorboard),* amelyek fény, hőmérséklet, légnyomás, gyorsulás, szeizmikus mozgások, hang és mágneses terek érzékelésére lehetnek képesek;
- Minden mote képes adattovábbító feladatokat is ellátni.

Egy MICAz mote általános blokkvázlata a 20. ábrán látható.




32**. ábra:** Egy MICAz mote blokkvázlata

A flash memória mérete 512 kbyte, amiben több mint 100 000 mérési eredmény is tárolható hosszútávon. Ez a viszonylag nagy memória méret azt a célt szolgálja, hogy legyen elég tárhely a mérési eredmények tárolására abban az esetben, ha a rádiós interfész foglalt. A TSS-ek tárolása a flash memóriában nem a legjobb megoldás, mivel a flash memória írási és olvasási ideje viszonylag nagy, így a TSS-eken végrehajtandó műveletek futási ideje is jelentősen megnőhet. Megemlíteném, hogy a TinyOS operációs rendszer a flash memóriakezelést interfészen keresztül valósítja meg, ezáltal a felhasználó számára jelentősen leegyszerűsödik a flash memória kezelése és használata.

A rádiós interfész egy integrált adó-vevő áramkör (CC2420), amely a ZigBee/IEEE 802.15.4 szabvány szerint működik a 2.4 GHz-es ISM sávban. A maximális sebesség elméletileg 250kbps, azonban a gyakorlatban ezt a sebességet nem lehet elérni a kommunikációs overhead miatt. A ZigBee szabványt célirányosan a kisméretű helyi hálózatokkal szemben támasztott követelményeknek megfelelően alakították ki. Egyik ilyen szempont a nagy csomópontszámú hálózatok támogatása, amit nagy címtartománnyal tesznek lehetővé. További fontos szempont a kis fogyasztású eszközök kialakításának támogatása, melyet esetünkben a CC2420 integrált rádiós interfész célirányos hardveres moduljaival is segít, mivel így nem szükséges a mikrokontroller erőforrásait bizonyos feladatokra feleslegesen felhasználni. Alapvetően a szabvány két alsó protokoll réteg meghatározására koncentrál, ezek a fizikai réteg és a közegelérési alréteg (MAC – Medium Access Control), amely az adatkapcsolati réteg része. A

szenzorhálózatok esetében a többutas jelterjedés gyakran hibák forrása lehet, ezért ebben az esetben a mote szélesebb frekvenciatartományban sugároz; a különböző frekvenciájú jelek különböző mértékben csillapodhatnak az adott környezetben, és így a hibák könnyebben elkerülhetővé válnak.

A mote tartalmaz egy ATmega128 típusú mikrokontrollert, amely egy általános célú RISC (Reduced Instruction Set Computer – csökkentett utasításkészletű számítógép) elv alapján tervezett 8 bites vezérlőegység. Ez a mikrokontroller a következő főbb tulajdonságokkal rendelkezik:

- 128 kbyte program flash memória;
- 512 kbyte mérési flash memória;
- 4 kbyte statikus RAM;
- 4 kbyte EEPROM (elektronikusan törölhető, programozható ROM);
- szinkron és aszinkron soros interfész;
- analóg komparátor;
- 10 bites A/D átalakító.

A program flash memóriába a program utasításai, a mérési flash memóriába a mérési eredmények, a 4 kbyte-os RAM-ba a program változói, míg a 4 kbyte-os EEPROM-ba a konfigurációs adatok kerülnek.

A mote-ok felprogramozása programozói kártya segítségével történik. Az általam használt mib520 típusú kártya nem csak a mote alkalmazások feltöltésében játszik fontos szerepet; csatlakoztatva hozzá egy szenzort, képes bázisállomásként funkcionálni, illetve megteremti a kapcsolatot a PC és a szenzorhálózat között, az USB csatlakozón keresztül kapcsolódva a számítógéphez.

33**. ábra** Mib520 programozói kártya

A MICAz mote-okhoz *érzékelő kártyák* (sensorboard) csatlakoztathatóak, melyek ténylegesen felelősek a környezeti paraméterek mintavételezéséért. Az érzékelő kártyák az általuk mért analóg jeleket egy 16 bites számmá konvertálják, majd ezt adják át a mote-nak. A mote-ok nincsenek tisztában a kapott adatok jelentésével, vagyis nem tudják, hogy azok mely érzékelőtől származnak; elvégzik a szükséges számításokat, majd továbbítják a feldolgozott adatokat a felhasználói alkalmazásnak. Az általam használt sensorboard az mts300 volt.

34**. ábra** mts300 sensorboard

Ez az érzékelő kártya képes fény, hőmérséklet, hangintenzitás mérésére. A szenzor programozásakor definiálhatóak monitorozó interfészek, és minden ilyen interfész hozzárendelhető egy adott környezeti paraméter mintavételezéséhez. A definiálás után egy „call” paranccsal olvasható le az adott interfész által érzékelt paraméter. Az is megoldható, ha a mintavételezés frekvenciája magas, hogy előre definiált adattömböket „fűzűnk föl” egymás után, amiket folyamatosan tölt fel az alkalmazás adatokkal. Amikor egy tömb megtelt mérési

eredményekkel, akkor ez egy „event”-tel kerül jelzésre, és ezt követően megtörténhet az adatok feldolgozása. Ezután az adott tömböt felfűzhetjük újra a „tömbsor” utolsó helyére, ezzel biztosítva, hogy folyamatosan tárolásra kerüljenek a mérési eredmények.

## *5.2 TinyOS*

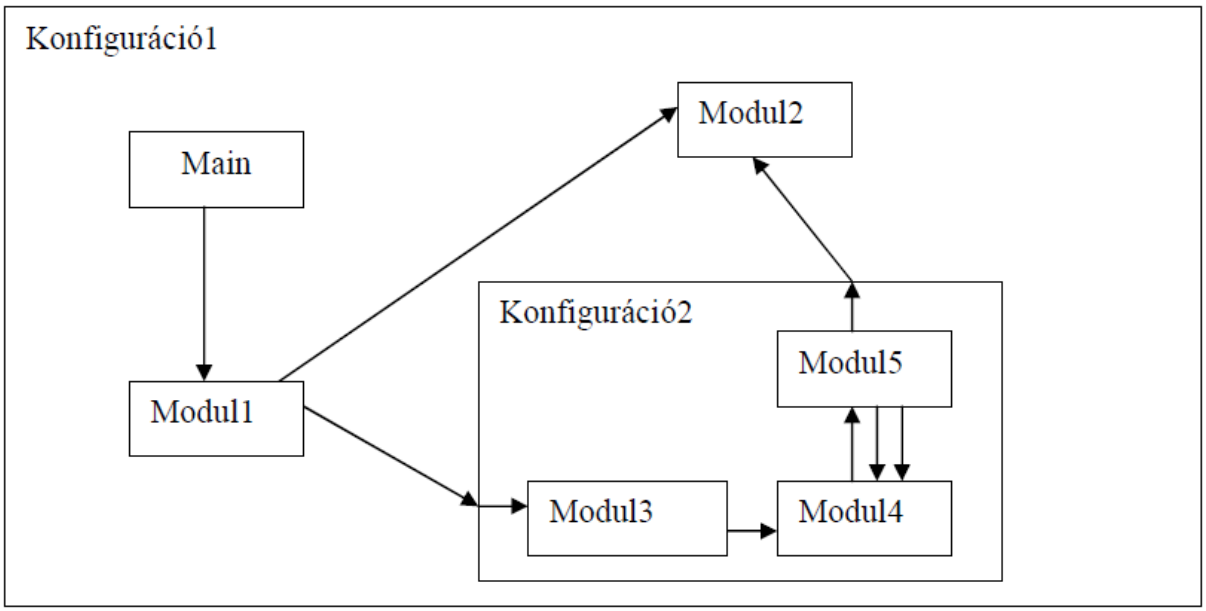
A TinyOS egy nyílt forráskódú és ingyenes operációs rendszer, amit kifejezetten a vezeték nélküli szenzorhálózatokhoz fejlesztettek ki. Felépítésére jellemző az eseményvezéreltség és a komponens alapú működés. Az egyes komponensek interfészeken keresztül kapcsolódhatnak egymáshoz. A komponensek közötti kapcsolat a fordításkor jön létre, utána az már nem módosítható. Az interfészek eseményeket (event) és parancsokat (command) szolgáltatnak, melyeken keresztül képesek kommunikálni a szoftverkomponensek. A parancsok segítségével felhasználhatjuk egy másik komponens szolgáltatásait, míg az események egy bizonyos történés bekövetkeztét jelzik.

Maga a TinyOs *NesC* nyelven íródott, ami egy C szintaxisú programozási nyelv. Egy adott alkalmazás lefordítása, majd telepítése után az operációs rendszer mindig része az alkalmazásnak, vagyis a hardveren nincs permanensen futó mag. A NesC nyelv két fajta komponens implementálását teszi lehetővé, ezek a *modulok* és a *konfigurációk*.

A modulokban implementáljuk a modul által szolgáltatott interfészek parancsait, illetve definiáljuk az események kezelését. A konfigurációban más konfigurációkat és modulokat kötünk össze azonos interfészeken keresztül. Két fajta konfiguráció létezik: *top level konfiguráció* és *újrafelhasználható konfiguráció*. A *top level konfiguráció* magát az elkészült programot jelöli. Ezzel a komponenssel kapcsolatos követelmény, hogy tartalmazza a *Main* modult, mely az *StdControl* interfészen keresztül végzi a program inicializálását, indítását, valamint leállítását. Az *újrafelhasználható konfiguráció* szintén komponensek összekötésére szolgál, de oly módon, hogy a benne szereplő komponensek bizonyos interfészeit kivezetjük, és kívülről hozzáférhetővé tesszük más komponensek számára.

Az eddigiek világossá tétele érdekében figyeljük meg a 23. ábrát, elyen a *Konfiguráció1* alkalmazás felépítése látható. A *Main* modul a *Modul1 StdControl* interfészével van összekötve. A *Konfiguráció2* három modult tartalmaz, és a *Modul5* valamint a *Modul3* komponensek interfészei vannak kivezetve. A *Modul5* a *Modul2* által szolgáltatott interfészt

használja, míg a *Modul3* interfésze elérhetővé van téve a *Modul1* számára. Az ábrán a nyilak az interfészt szolgáltató modul felé mutatnak.




35**. ábra** Komponens diagram

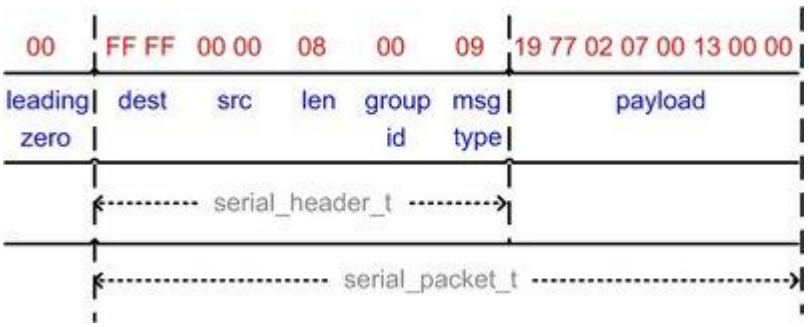
A NesC lehetőséget ad úgynevezett *task*-ok alkalmazására. Ezeket a task-okat a programunk bármely pontjából meghívhatjuk. Több task meghívása azok a meghívásuk sorrendjében futnak le, vagyis egy task nem szakíthat félbe egy másik task-ot, hardver megszakítás viszont igen. Ebből az következik, hogy a task-okat a kisebb prioritású feladatok végrehajtásához használjuk, hiszen végrehajtásukat megszakítások késleltethetik. Mint az eseményvezérelt rendszerekben általában, a TinyOS-ben is meg kell oldani a kölcsönös kizárást az erőforrások használatánál. Ez kétféleképpen történhet. Egyfelől az *atomic* kulcsszó által kijelölt tartományon belüli utasítások megszakítás nélkül futnak le. Másfelől a kölcsönös kizárást elkerülhetjük úgy is, ha a közös erőforrásokat csak task-okon belül hívjuk meg.

### 5.2.1 A TinyOS néhány főbb komponense

A *TimerC* komponens tartalmazza az időzítésekért felelős szoftverelemeket. Egy úgynevezett *Timer* interfészen keresztül csatlakozhatnak hozzá más komponensek. Amikor egy másik szoftverelemben hivatkozunk a *Timer* interfészre, akkor rendelhetünk hozzá egy paramétert, ami az időzítés idejét határozza meg. Ezáltal az is lehetővé válik, hogy az interfész több

szálon kapcsolódjon ugyanahhoz a komponenshez. Az egyes kapcsolatok csak paraméterezésükben térnek el, tehát a *Timer* interfész virtuálisan több időzítendő elemet is képes kiszolgálni egyszerre.

A kommunikációért a *GenericComm* komponens felelős. Ez két fontosabb interfészt definiál: *ReceiveMsg* és *SendMsg*. Ezek segítségével lehet üzeneteket küldeni, illetve eseményt generálni, amikor üzenet érkezik. Az üzenetek csomagokban kerülnek továbbításra. Egy TinyOs általános üzenet struktúrája a 24. ábrán látható.




36**. ábra:** TinyOs üzenetstruktúra

Alapértelmezett esetben egy ilyen üzenetcsomag maximum 29 bájtnyi adatot tartalmazhat. Fejlécezéssel és keretezéssel együtt a maximális hossz 40 bájtra bővül. Ezekben a fejlécekben szerepel a címzett azonosítója, a forrás azonosítója, a csoport azonosítója, és az üzenethossz.

## 5.3 Közlekedési útvonalak mérésére szolgáló szenzorhálózat

A közlekedési útvonalakon és csomópontokban végzett mérések által valós adatokat tartalmazó bemeneti fájlok álltak rendelkezésre a szimulátorhoz. Ebben a fejezetben szeretném röviden bemutatni a mérésekhez felhasznált szenzorhálózat működését.

A mérések a következőek szerint történtek:

- a programozói kártya és a PC-n futó felhasználói interfész segítségével szenzorok felprogramozása;
- a mérni kívánt útvonal vagy kereszteződés megközelítése;

- szenzorok elhelyezése a mérni kívánt pontokon, valamint a „Slave” node-ok bekapcsolása;
- a mérés indítása a „Master” node bekapcsolásával;
- mérés végeztével a node-ok begyűjtése és kikapcsolása;
- a programozói kártya és a PC-n futó felhasználói interfész segítségével az mért adatok kiolvasása és loggolása;
- a node-ok flash memóriájának törlése.

A node-ok felprogramozásakor van lehetősége a felhasználónak tetszőleges paramétereket megadni a mérések lebonyolításához, mint például a mért adatok közötti loggolási időköz, küszöbérték a mérési eredményekhez, stb. A mintavételi-frekvencia minden esetben 4kHz volt, az ezzel a frekvenciával képződő mérési eredmények átlagolódtak ki a loggolási időpillanatokra. A slave node-ok akkor indították el a mintavételezést, amikor a master node-tól megkapták az első szinkronizációs üzenetet. Az adatok kiolvasásakor a nodeok folyamatosan olvasták ki a flash memóriából a mérési logokat, majd tíz ilyen log kiolvasása után továbbították azokat a bázisállomásnak, mely a kapott rádiós csomagot egy az egyben továbbította a soros porton keresztül a PC-n futó felhasználói interfésznek. Az adatok küldésének befejeztével az adatok PC-n való loggolása is megtörténik. Az összes szenzor adatainak beolvasása után lehet képezni a teszt fájlokat, valamint a szenzorok flash memóriáját ki lehet törölni, hogy készen álljanak a következő mérésre.

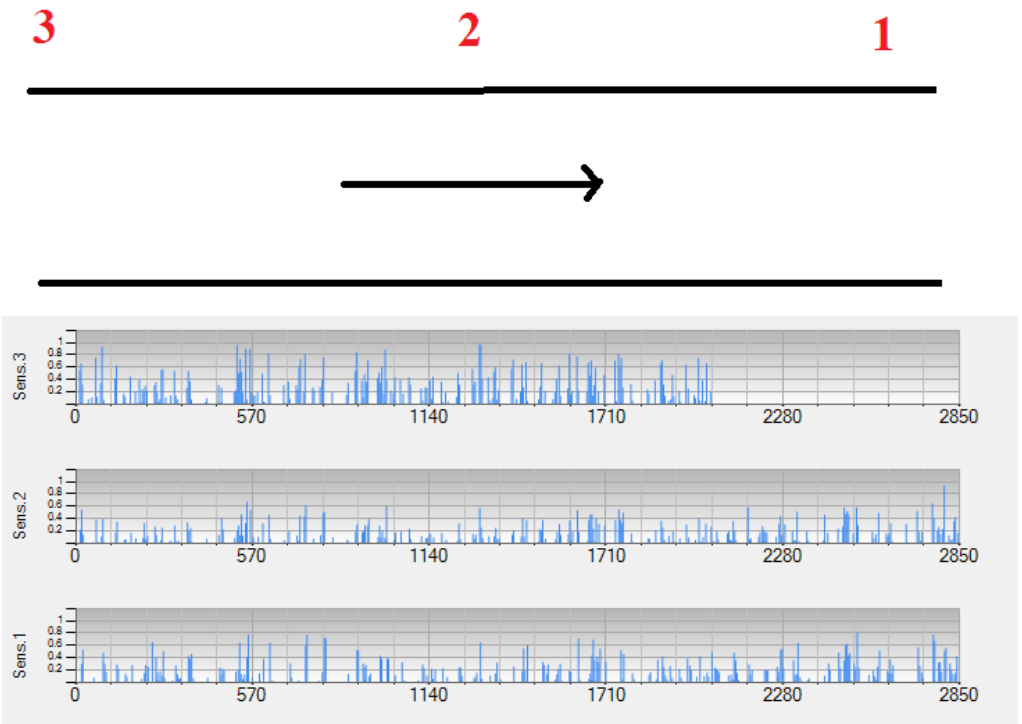
A szenzorok szinkronizációját egy master-slave viszony biztosította, vagyis volt egy master node, amely periodikusan (pár percenként) küldött egy broadcast üzenetet a saját órájának aktuális értékéről. Amikor egy slave node megkap egy ilyen üzenetet, indít egy új timer interfészt, és az órájának értékét beállítja a szinkron üzenetben kapott értékre. Az 5-6 órás tesztelések azt mutatták, hogy a szenzorok szinkronba maradtak, amennyiben a mintavételi-frekvencia 100ms vagy annál nagyobb volt.

## 5.4 Közúti mérések

Ebben a fejezetben bemutatok néhány közúti mérés alapján futtatott szimulációt, és megosztom a szimulációs tapasztalatokat.

### 5.4.1 Egyirányú egyenes szakasz

A legegyszerűbb, és az autók hangintenzitás által keltett esemény-szekvenciák szempontjából talán legkevésbé zajjal terhelt eset az egyirányú egyenes szakasz mentén elhelyezett szenzorok általi mérés. Az esetleges nehézséget ennél az esetnél a különböző sebességgel, különböző hangintenzitással közlekedő autók, valamint az autók sebességének változása okozhat. A 37. ábra szemlélteti a mérési elrendezést, valamint az egyes szenzorok által mért hangintenzitásokat az idő változásával. (A piros számok az adott azonosítóval rendelkező szenzor elhelyezkedését jelölik az út mentén.) A szenzorok között körülbelül 20m-es távolság volt.



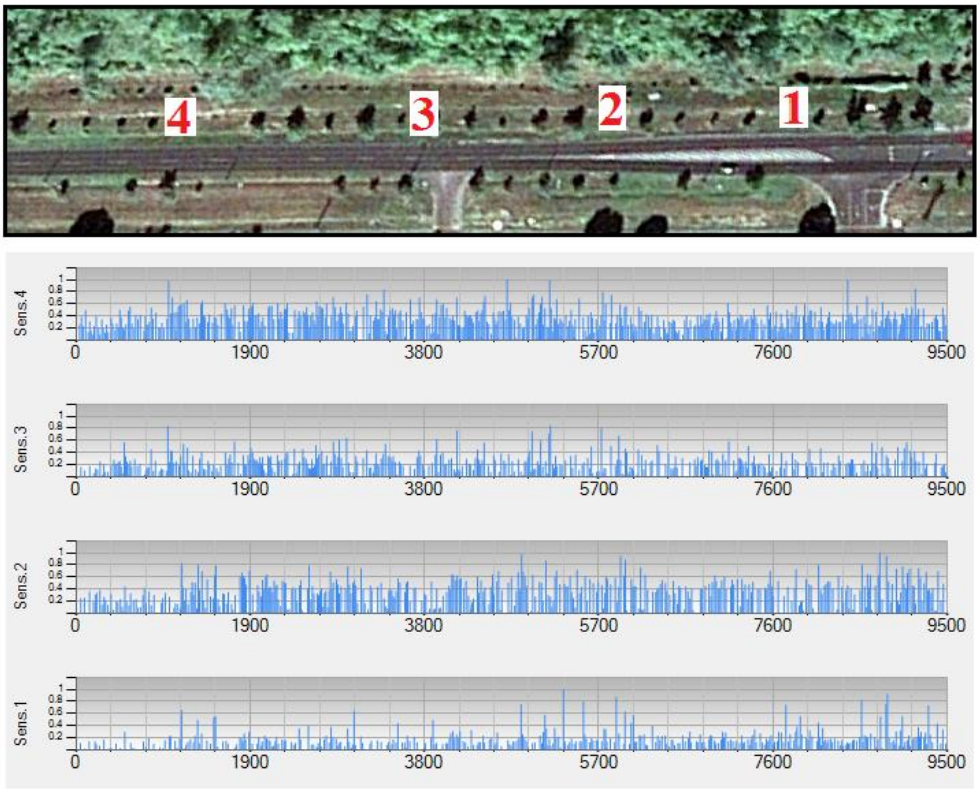

37. **ábra** Egyirányú egyenes szakasz mérési elrendezése és az egyes szenzorok által észlelt események

Az algoritmus egyértelműen azonosította az útszakaszon előforduló esemény-szekvenciákat, tehát az eredményül kapott esemény-szekvenciák időben így követték egymást: 3,2,1. Ezt a

szekvenciát több változatban is eredményül adta, melyek sorrendje azonos volt, de az események között eltelt időtartamok változtak, tehát a különböző sebességgel haladó gépkocsik által keltett esemény-szekvenciák is azonosításra és elkülönítésre kerültek.

### 5.4.2 Kétirányú egyenes szakasz

A vizsgálódás következő tárgya a szintén egyenes, de kétirányú forgalommal rendelkező út vizsgálata volt. A szenzorok az előző esethez hasonlóan az út egyik oldalán sorban, egymás után, körülbelül 20mes távolságra helyezkedtek el egymástól. Az autók sebessége, hangintenzitásának változékonysága mellett a két irányból közlekedő autók által kelttett esemény-szekvenciák átlapolódása is megjelent a mérési eredményekben.




38. ábra Kétirányú egyenes szakasz mérési elrendezése és a szenzorok mérési eredményei

A 38. ábra mutatja a mérési elrendezést, valamint az egyes szenzorok által mért hangintenzitásokat. A képen látható vízszintes útra merőleges utak sorompóval védett lakótelepi bejáratok. tehát a mérési idő jelentékeny hányadában fordultak be és ki onnan gépjárművek. Az egyes szenzorok mérési eredményeiben látható hangintenzitásbeli különbségek az autók sebesség különbségéből adódhattak az adott részein az útnak, valamint

az éppen gyorsuló autók hangintenzitás jelentősen nagyobb, mint az állandó sebességgel, vagy éppen fokozatosan lassulóé. Az út szenzorok felöli oldalán az autók (a képen) jobbról-balra, míg az átellenes oldalon balról-jobbra haladtak. Tehát a keresett esemény-szekvenciák az 1-2-3-4 illetve a 4-3-2-1 voltak.

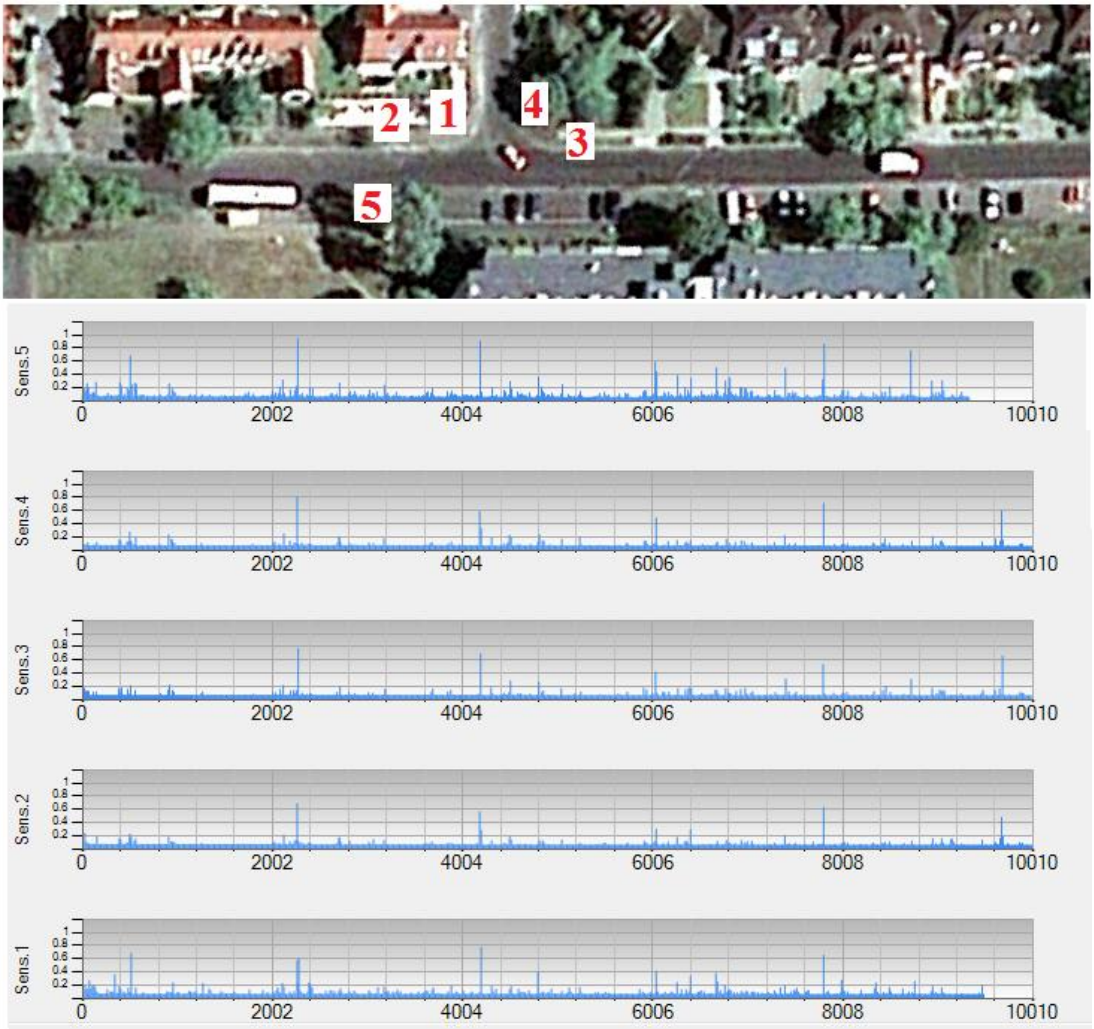
Az algoritmus futtatása során a szenzorokhoz közelebbi sávban közlekedő autók által keltett szekvenciából (1-2-3-4) a 4-es szenzor kiadta a (2-3-4)-es, a 3-as szenzor a (3-2), míg a 2-es szenzor a (2-1) sorozatokat. Annak, hogy a szenzorok nem találták meg egyenként a teljes esemény-szekvenciát, azon kívül, hogy a mérési eredmények átlapolódtak a másik szekvenciával, még oka lehetett a nem ideális TSS méret, valamint az, hogy az 1-es szenzor mérési eredményei értékeinek nagysága kisebb volt a többi szenzor mérési eredményeihez viszonyítva. (Ez a grafikonon is látszik.)

A (4-3-2-1) szekvencia keresése az algoritmussal azt az eredményt adta, hogy az egyes szenzorok jellemzően csak egy „szenzornyit látnak” visszafele, tehát a (4-3),(3-2) es a (2-1) sorozatok adódtak ki. Annak tudatában, hogy a szenzorok a tőlük távolabb eső sávot kevésbé érzékelhették, valamint a közelebbi sáv egy-egy autójának hangját a szenzorok tovább észlelhették (ezzel több loggolásnyi időt elfedve), az esetleges nem megfelelő TSS méret ennek a szekvenciának az azonosításánál fokozottabban érződhetett.

Összességében azt mondhatjuk, hogy ha a szenzorhálózat rendelkezik egy bázisállomással, és az egyes szenzorok elküldik az általuk érzékelt lokális szekvenciákat ennek, akkor a bázisállomással kapcsolatban álló felhasználói interfész, és feldolgozó program könnyedén összeilleszthetti ezeket, ezzel azonosítva a hálózat területén jelentkező globális esemény-szekvenciákat. (Azt mondhatjuk, hogy érvényesült a szenzorhálózatok elosztott módjából adódó fokozottabb zajszűrési tényező.)

### 5.4.3 Kis forgalmú útkereszteződés

Az egyenes útszakaszok után egy kisforgalmú, lakóövezetben található kereszteződés forgalmát mértem. A 39. ábrán látható a kereszteződés felülnézeti képe, valamint a szenzorok elhelyezkedése a mérés során. A lakóövezet miatt az autók átlagsebessége alacsonyabb volt (20-25 km/h), mint az előző méréseknél.




39. ábra Kis forgalmú kereszteződés mérési elrendezése és mérési eredményei

Az 1-es szenzor által talált szekvenciák az (2-5-1) és az (3-5-1) voltak, ezek valószínűsíthetően a védett útról (ábrán a vízszintes út) jobbról és balról a mellékútra behajtó gépkocsik által keltett szekvenciák lehetnek. A 2-es szenzor szekvenciái az (5-2) és (4-2) önmagukban értelmezhetetlenek. A 3-as szenzor által kiadott esemény-szekvenciák a (4-3) és a (5-2-4-3) voltak. Az előbbi a védett útról jobbról a mellékútra behajtó autókat jelölheti, míg

az utóbbi a védett úton egyenesen továbbhaladó gépjárműveket. A 4-es szenzor a (2-5-4)-es szekvenciát adta eredményül, amely szekvencia a védett úton balról érkező, és a mellékutcába beforduló autókat jelölheti. Az 5-ös szenzornál a (3-2-4-5) és az (3-5) szekvenciák adódtak eredményül. Az (3-2-4-5) –es sorozat értelmetlen, míg a (3-5)-ös szekvencia az egyenesen haladó autókat reprezentálhatja.

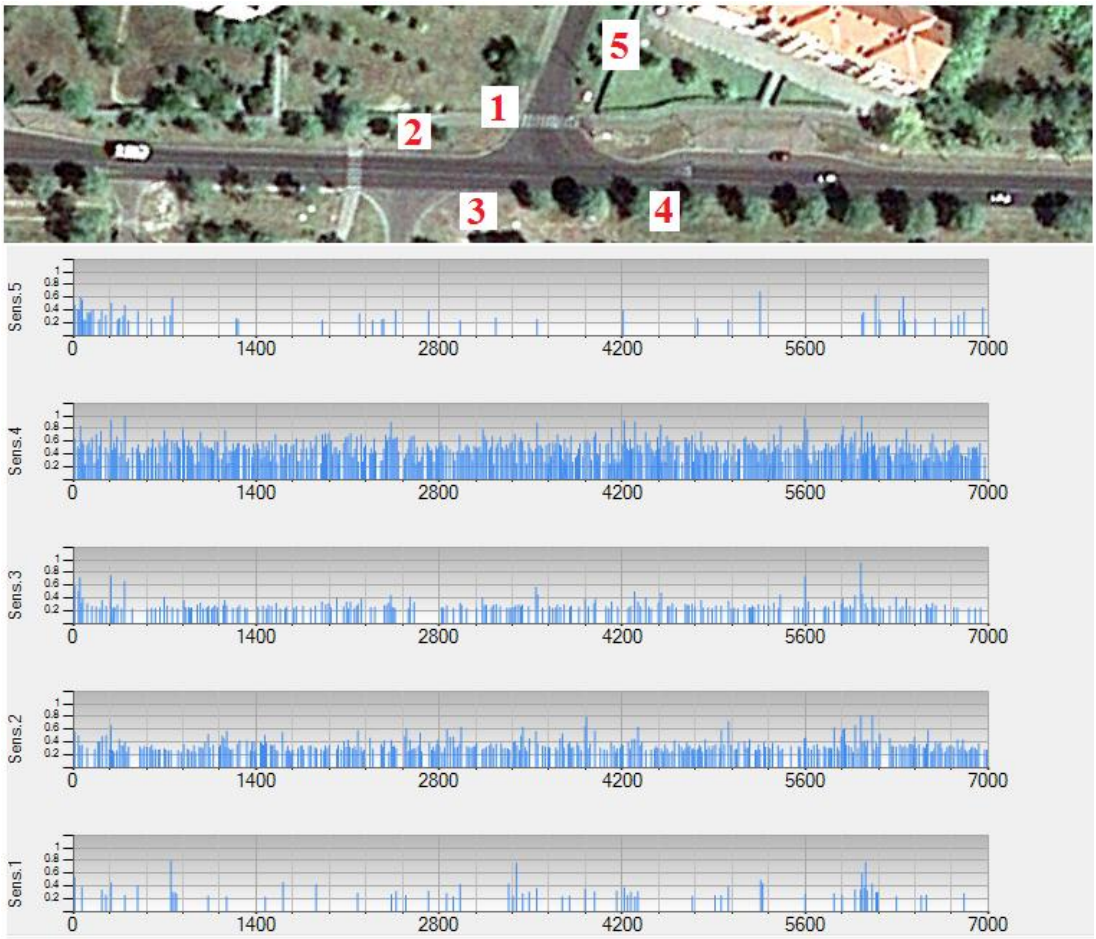
Az eredmények jól jellemzik az adott kereszteződésben előforduló forgalmat, viszont említést érdemel, hogy az algoritmus paramétereit hangolni kellett az értelmezhető eredmények érdekében. (clustering_param, threshold_variance)

### 5.4.4 Nagy forgalmú útkereszteződés

A következő mérés egy hagyományos, viszonylag nagy forgalmú kereszteződésben történt. Egy védett útra (az alábbi ábrán a vízszintes út) két oldalról lehet ráfordulni alacsonyabb rangú utakról. Az adott szakaszon közlekedő autók sebességének átlagértéke is jelentősen magasabb volt (45-50km/h). Megállapítható tehát, hogy a járművek sűrűbb és magasabb intenzitású eseményeket generáltak, mint az előző esetben.

A mérési eredményeket szemügyre véve rögtön szembetűnik, hogy a 4-es szenzor jelentősen nagyobb számú eseményt rögzített, mint a többi szenzor. Ennek az lehetett az oka, hogy az út olyan szakaszán helyezkedett el, amelyen a járművek gyakran gyorsítanak, az éppen gyorsuló gépjármű által kibocsátott hangintenzitás pedig jelentősen magasabb, mint az állandó sebességgel haladóé. Annak érdekében, hogy kiküszöböljük azt az esetet, amikor a TSS-ek a 4-es szenzor eseményeivel vannak megtelve, a szimulátor események $\mu$-értékére vonatkozó beolvasási EThreshold paraméterét magasabbra állítottam.

A talált szekvenciák a következőek voltak: az 1-es szenzor túl kevés eseménnyel rendelkezett, ezért nem adott eredményül esemény-szekvenciát. A 2-es szenzor a (4-3-2)-es esemény-szekvenciát adta eredményül, amely a védett úton jobbról balra közlekedő gépkocsik hangintenzitása által keltett események lehetnek. A 3-as szenzor a (2-3)-as esemény, a 4-es szenzor a (3-4), míg az 5-ös a (4-5) szekvenciát adta eredményül. Az első kettő a védett úton balról jobbra haladó járműveket, amíg az utolsó a védett útról a mellékútra kanyarodó gépjárműveket jelölheti.




40. ábra Nagy forgalmú kereszteződés mérési elrendezése és mérési eredményei

Összességében azt mondhatjuk, hogy a szenzorok kis száma és az események nagy száma miatt gyakran csak 2-3 elemű esemény-szekvenciákat kaptunk eredményül, amennyiben az egyes szenzor csomópontok ezeket a talált szekvenciákat elküldik a bázisállomásnak, az pedig az egyes szekvenciákat illeszti, akkor megfelelő modellt kaphatunk az egyes forgalmi csomópont vagy úttest forgalmi viszonyairól.

# 6 Összegzés

A szenzorhálózatok hasznosságát és a bennük rejlő lehetőségeket az élet minden területén kezdik felismerni; ennek okán számos szenzorhálózatos tanulmány született a közelmúltban és születik a jelenben is. Ezek a hálózatok nagymértékben különböznek a hagyományos érzékelő-rendszerektől. Intelligens szenzorokból állnak, mely szenzorok valamilyen közös mérési feladat végrehajtását elosztott módon valósítják meg. A szenzorok összehangolt mintavételezésével és jelfeldolgozásával a szenzorhálózat magasabb fokú érzékenységet és zajelnyomást képes biztosítani, és a dolgozatban bemutatott esemény-előrejelző algoritmus is a szenzorhálózatok ezen tulajdonságát használja ki

A dolgozatomban bemutattam az esemény-előrejelző algoritmus megvalósításának egy lehetséges formáját. Ez a funkció számos gyakorlati probléma megoldása során alkalmazható volna, például felügyeleti rendszerekben, közlekedési torlódás megelőzésben vagy betegfelügyeleti rendszerekben. Az algoritmus teszteléséhez az általam számítógépre implementált szenzorhálózat szimulátort használtam. A szimulátorban nagyszámú szenzor szimulálható, valamint számos szimulációs paraméter állítható. A valós adatokon való tesztelés céljából implementálásra került egy szenzorhálózat rendszer, mellyel különféle forgalmi utakon valamint kereszteződésekben mértem a gépkocsik által keltett hanghatások. A mért adatok PC-n való beolvastatása után, az algoritmus tesztelhetővé vált valós körülmények között is.

A tesztelési eredmények azt mutatták, hogy a szenzorok elhelyezkedése a mérések során nagyban befolyásolta, hogy adott idő alatt hány eseményt rögzített az adott szenzor. Ennek okán a későbbiek során szükséges lehet egy önkonfigurációs algoritmus kidolgozására, mely például a mérésekhez tartozó küszöbértéket állítana, annak érdekében, hogy nagyságrendileg azonos számú eseményt rögzítsenek az egyes szenzorok, ezzel az esemény-előrejelző algoritmust segítve, hiszen a szimulátorban való tesztelés során az algoritmus hatékonysága a várakozásnak megfelelően alakult. Ezek után lehet az algoritmust a különféle felhasználásokhoz igazítani.

# Irodalomjegyzék

[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, *Wireless Sensor Networks: a Survey*, Computer Networks, Vol. 38, pp. 393-422., 2002.

[2] G. Hoblos, M. Staroswiecki, A. Aitouche, *Optimal design of fault tolerant sensor networks*, IEEE International Conference on Control Applications, Anchorage, AK, September 2000, pp. 467–472.

[3] J.M. Rabaey, M.J. Ammer, J.L. da Silva Jr., D. Patel, S. Roundy: *PicoRadio supports ad hoc ultra-low power wireless networking*, IEEE ComputerMagazine (2000) 42–48.

[4] Rui Xu, Donald C. Wunsch, II: *Clustering*. John Wiley and Sons, New Jersey, 2009

[5] Jys-Shing Roger Jang, Chuen-Tsai Sun, Eiji Mizutani: *Neuro-Fuzzy and Soft* http://www.soukalfi.edu.sk/01_NeuroFuzzyApproach.pdf

[6] Kóczy T. László: Fuzzy rendszerek

http://www.tankonyvtar.hu/informatika/fuzzy-rendszerek-fuzzy-080904

[7] Gergely Öllös, Rolland Vida: *Infrequent Event Forecasting in Wireless Sensor Networks*

[8] Tibor Jordán, András Recski, Dávid Szeszlér*: Rendszeroptimalizálás*, Budapest 2004

[9] Schaffer Péter: *Spontán kooperáció kialakulásának vizsgálata különböző fennhatóság alá tartozó szenzorhálózatok között*, Diplomadolgozat, BME, 2005.

[10] Orosz György, Lajkó László, Sujbert László: *Aktív zajcsökkentő rendszerek megvalósítása szenzorhálózattal*, TDK Dolgozat, BME, 2005.

[11] Völgyesi Péter: *Szenzorhálózatok*

http://volgy.com/pubs/BIR_SensorNetworks.pdf

[12] David S. Platt: *Bemutatkozik a Microsoft .NET*, Szak Kiadó Kft. Bicske, 2001

[13] Trey Nash: C# 2008, *Könnyen is lehet!*, Panem Könyvkiadó, Budapest, 2009,

[14] MICAz datasheet and user's manual

http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.p df

http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf

# Ábrajegyzék