Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

SAT-solverek gyorsítása best-first search algoritmussal

Bartók Dávid
II. évfolyam, mérnök informatikus szak

Konzulens: Dr. Mann Zoltán Ádám
Számítástudományi és Információelméleti Tanszék

2013. október 25.

Tartalomjegyzék

Ábrák jegyzéke

Összefoglalás

A mérnöki gyakorlatban a legfontosabb problémák jelentős részére igaz, hogy jelenleg nem létezik minden példányukat gyorsan (polinomidőben) megoldani képes algoritmus. Ezeket összefoglaló néven NP-teljes problémáknak nevezzük. Az első ismert NP-teljes probléma a Boolean Satisfiability (SAT), ahol egy logikai formula változóinak próbálunk úgy értéket találni, hogy a kifejezés igazra értékelődjön ki. Az ipari alkalmazásokban gyakran fordulnak elő olyan problémák, melyeket könnyen megfogalmazhatunk SAT-problémaként. Ezek nehézsége a mai számítógépek számítási kapacitását is keményen próbára teszi. A kutatása éppen emiatt népszerű, olyannyira, hogy évente SAT-versenyt is rendeznek. Itt saját készítésű algoritmusokkal lehet részt venni, és a cél problémapéldányok minél gyorsabb megoldása.

A jelenlegi SAT-megoldó algoritmusok (solverek) legnagyobb problémája az, hogy bár sok esetben gyorsan oldják meg a problémákat, néha a futási idő a szokásosnál nagyságrendekkel hosszabbra is nyúlhat. Ennek csökkentésére a jelenlegi leghatékonyabb módszer az, hogy a keresést bizonyos időközönként újraindítjuk. Valószínűsíthető, hogy ekkor hamarabb eredményre jutunk, mintha az algoritmust beavatkozás nélkül futtatnánk. Ez a megoldás azonban közel sem tökéletes: a keresés elért részeredményei az újraindításnál elvesznek, valamint semmi sem garantálja, hogy az új futtatásnál valóban gyorsan eredményt kapunk.

A TDK dolgozatomban az újraindítás helyett egy másik módszert alkalmazok a futásidők szórásának csökkentésére. Ez a „legjobbat először” (best-first search) algoritmus, amely a futása során gyűjtött adatok alapján próbálja kitalálni, merre kell keresnünk a megoldás minél hamarabbi eléréséhez. Ezáltal a solver képes lesz arra, hogy változtasson a keresési stratégiáján, de az addigi részeredményei megtartása mellett.

A jelenlegi SAT-solverek nem összeegyeztethetők a hagyományos best-first search kialakításával, ezért az alkalmazáshoz az algoritmus jelentős módosítására volt szükség. Fontos kérdés volt az is, hogy pontosan milyen adatok alapján tudjuk megítélni a megoldás közelségét? A tervezés után a kész algoritmust a MiniSatba, egy gyors, de könnyen bővíthető SAT-solverbe implementáltam. Az eredmények kiértékelését és a finomhangolást egy algoritmusok tesztelését segítő szoftvercsomag, a BCAT könnyítette meg.

A tesztelés során kapott eredmények azt mutatják, hogy ezzel a módszerrel számottevően gyorsítottam a különféle problémapéldányok megoldását, és sikerült csökkentenem a futásidők hatalmas különbségét is.

Abstract

For most of the problems in engineering practice currently exists no algorithm, that can solve all instances quickly (in polynomial time). These problems are called NP-complete. The first known NP-complete problem was Boolean Satisfiability (SAT), in which we have to assign values to the variables of a Boolean formula so that it evaluates to „true". Problems that can easily be interpreted as SAT occur often in industrial applications. The difficulty of these can be challenging even for the computational capacity of today's hardware. Accordingly, the research of this problem is so popular that a SAT competition is held annually. Here the participants apply with self-made algorithms and the goal is to solve many instances of SAT as fast as possible.

The greatest trouble with current SAT algorithms (solvers) is that although they mostly solve the problems fast enough, sometimes running time can stretch to even other orders of magnitude. The current most effective solution to this is restarting search after a given amount of time. By using this strategy we probably receive a result faster than if we ran the algorithm without external intervention. However, this method is far from perfect: partial solutions are lost during restarts and we have no guarantee that the next run truly provides a quick result.

In this paper I propose another solution to reduce the variance of running times. This is the best-first search algorithm, which uses data collected during its run to predict where the result might be found. Thus the solver becomes able to change its search strategy but also keeping the already discovered partial results.

Current SAT solvers are not suited for the usage of best-first search, therefore, significant changes had to be executed on the original algorithm before realization. It was also really important to determine how to approximate the closeness of a solution during search. After the design was complete I implemented the algorithm into MiniSat, which is an effective but easily extensible SAT solver. The evaluation of results and fine-tuning were made easier by BCAT, a software package built for testing of combinatorial algorithms.

The achieved results show that by using best-first search I made the solution of several SAT instances significantly faster and the variance of running times has also been reduced.

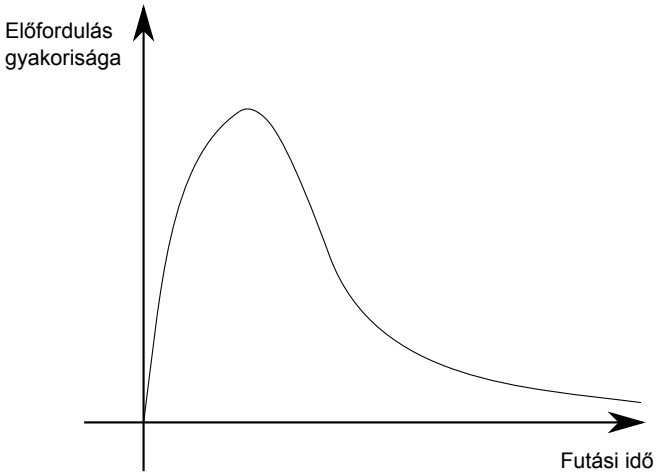
1. fejezet

Bevezetés

1.1. Témaválasztás

A TDK dolgozatomban a best-first search (BFS) algoritmus alkalmazását mutatom be boolean satisfiability (SAT) problémákra. A SAT-problémában egy konjunktív normálformában megadott logikai formuláról kell eldöntenünk, hogy a változóinak tudunk-e úgy értéket adni, hogy a kifejezés igazra értékelődjön ki. Ez volt az első ismert NP-teljes probléma: ez azt jelenti, hogy jelenleg nincsen olyan solver (megoldóprogram), amelyik minden egyes problémapéldányra az input hosszának függvényében polinom futási idejű [12]. Valószínűsíthető, hogy ilyet egyáltalán nem lehet találni, azaz csak a probléma egyes speciális példányaira léteznek hatékony algoritmusok. Mivel a SAT-probléma alkalmazásai szerteágazóak, nagyon sokan foglalkoznak a kutatásával. Évente kerül megrendezésre a nemzetközi SAT verseny, ahol saját készítésű solverekkel lehet nevezni, és a cél különféle problémapéldányok minél gyorsabb megoldása.

A SAT-solvereknél jelenleg a legnagyobb probléma az algoritmusok futási idejének ún. nehéz farkú eloszlása (heavy-tailed runtime distribution). Ez azt jelenti, hogy átlagos esetben rövid idő alatt lefut az algoritmus, azonban nem elhanyagolható azon esetek száma, amikor a futási idő az átlag akár többszörösére is nyúlik, vagy néha nagyságrendekkel nagyobbra is. Ez látható az 1.1. ábrán.



1.1. ábra. A nehéz farkú eloszlás

Ennek jelenlegi leghatékonyabb megoldása az, hogy ha az algoritmus már régen fut, akkor újraindítjuk [9]. Ezt azért tesszük, mert reméljük, hogy következő nekifutásra másképp fog viselkedni, és eredményesebb lesz, azaz az átlagos futási idő alatt, vagy annak környékén végez. Ezzel a megközelítéssel több probléma is felmerül: egyrészt az újraindítással elvesznek az algoritmus eddig elért részeredményei. Másrészt a statisztikán kívül semmi sem garantálja, hogy az ismételt próbálkozás gyorsabb lesz. Azonban a gyakorlat azt mutatja, hogy ez az ún. *restart-módszer* meglepően jól működik, és nagy mértékben gyorsítja a solverek működését [13].

Mi lenne viszont akkor, ha a restart helyett valahogy elérhetnénk azt, hogy az algoritmus valami annyira számottevőt változtasson a keresésben, ami felér ugyan egy újraindítással, de emellett megtartaná az eddigi eredményeit? Kívánatos lenne továbbá az is, hogy a solver ne egy véletlenszerű úton folytassa tovább a keresést, hanem úgy, hogy minél nagyobb valószínűséggel jusson gyors megoldásra. Ezekre a problémákra kerestem a megoldást a kutatásom során, azt várva, hogy így gyorsítani tudom a SAT-solverek működését, leginkább a nehézfarkú eloszlás farokrészének csökkentésén keresztül.

Ez a cél apróbb algoritmikus módosítások segítségével nem érhető el, ezért egy teljesen új megközelítésre volt szükségem. A tapasztalatok azt mutatják, hogy a best-first search, melyet a következőkben részletesen bemutatok, sikeresen alkalmazható ezen célok elérésére a gráfszínezés esetén [5], ami szintén egy NP-teljes probléma. Ez keltette fel a figyelmemet a módszer iránt.

A kutatásom a következő lépésekből állt: először áttanulmányoztam a SAT problémához és a best-first searchhöz kapcsolódó elméleti alapokat. Ezután megismerkedtem a MiniSattal, egy nagyon sikeres SAT-solverrel [7]. Sikerének az oka többek között átláthatóságában és bővíthetőségében rejlik. Ezután kitaláltam, hogyan lehetne ebbe beépíteni a BFS algoritmust, majd kiegészítettem a MiniSat forráskódját ennek megfelelően. Az algoritmus tesztelésének megkönnyítéséhez implementáltam azt a BCAT keretrendszerbe, ami egy algoritmusok célirányos tesztelésére szolgáló szoftvercsomag [6]. Az utolsó lépés a tesztelés és az eredmények értékelése volt.

1.2. SAT gyakorlati alkalmazásai

A SAT-probléma nagyon fontos szerepet tölt be a mérnöki gyakorlatban. Többféle tervezési és tesztelési feladatban használnak SAT-solvereket, valamint számtalan matematikai és informatikai kérdés fogalmazható meg SAT-problémaként [15]. Ezek közül mutatok be néhányat.

1.2.1. Elektronikában

A kombinációs ekvivalencia ellenőrzésénél (combinational equivalence checking) azt ellenőrizzük, hogy két kombinációs hálózat azonosan működik-e. A gyakorlatban ez például akkor fordulhat elő, hogyha van egy hardverünk, amit már teszteltünk, és tökéletesen működik, de valamilyen okból apróbb módosításokat kell rajta végrehajtanunk. A működésen nem szeretnénk változtatni, ezért összevetjük azt az eredeti változatéval, mivel arról tudjuk, hogy a kívánt kimenetet állítja elő.

A megoldást úgy kapjuk, hogy az összes lehetséges bemenetre összehasonlítjuk a hálózatok kimenetét. Ha minden egyes alkalommal ugyanazt az eredményt adják, akkor ekvivalensek, ha viszont létezik olyan input, hogy az outputok különbözőek, akkor nem azonos a működés.

Azt, hogy az outputok egyenlőek-e, legkönnyebben egy kizáró vagy (XOR) kapcsolattal tudjuk jellemezni. Legyen a két hálózatot leíró függvény f illetve F. Ha a két hálózat *nem* ekvivalens, legalább egy x inputra igaz a következő formula:
$f(x) \oplus F(x) = 1$

A logikai áramkörök konjunktív normálformává való konvertálása ismert [3], ezt alkalmazva a fenti formulára egy SAT-problémával találjuk szemben magunkat.

Az integrált áramkörökben különböző hibák léphetnek fel, amelyek kiszűrése nagyon fontos. Erre a leggyakrabban használt módszer az automatikus tesztminta-generálás (automatic test-pattern

generation - ATPG). Az áramkörben található hibákat a *stuck-at* modell segítségével képzelhetjük el: a hardverben egy összeköttetés „beragadt" valamilyen logikai állapotba (0 vagy 1). Az ATPG olyan inputot generál, amivel egyértelműen kideríthető, hogy egy feltételezett hiba jelen van-e a hálózatban.

Ez a probléma értelmezhető egy ekvivalencia-ellenőrzésként is, ahol a két összehasonlítandó hálózat a jól és a rosszul működő változatok. Amelyik bemenetre a két hálózat különböző kimenetet ad, az alkalmazható a hiba tesztelésére. Ha nincsen ilyen bemenet, akkor a hiba ilyen módon nem mutatható ki. A gyakorlatban ezt a módszert még különféle trükkökkel gyorsítják, pl. a hibásan működő hálózatot nem önmagában, hanem az eredetitől való különbségeivel reprezentálják, azonban ez a SAT-problémává konvertálhatóságot nem befolyásolja.

Egy rendszer ellenőrzésénél nagyon fontos szempont, hogy az megfelel-e a specifikációnak. Erre szolgál a modellellenőrzés (model checking), aminek a kombinációs ekvivalencia ellenőrzése tulajdonképpen egy speciális esete, hiszen abban az esetben a specifikáció maga is egy hálózat. A modellellenőrzésnél a specifikációt egy idő-logika segítségével adjuk meg, például a következőképpen:

A hűtőben a lámpa nem kapcsol fel, amíg az ajtó zárva van.
NOT *lampa_bekapcsol* **UNTIL** *ajto_zarva*

Ezt a specifikációt hasonlítjuk össze az állapotgépként megadott implementációnkkal. Gyakran definiálunk egy ellenőrző függvényt, amelynek az állapotgép minden állapotában igaznak kell lennie. Korlátozott modellellenőrzésnek (bounded model checking) hívjuk azt, amikor azt vizsgáljuk, hogy a kezdőállapotból k lépésszámmal milyen állapotokba lehet eljutni, és ezekben az állapotokban teljesül-e ez a függvény. A teszteléshez célszerű ezt 0-tól egy elég nagy k számig végignézni. A modellellenőrzés ezen változata konvertálható SAT-problémává, és a gyakorlatban ez elég hatékonynak bizonyul [2].

1.2.2. Genetikában

A *haplotípusok* az ember génkészletében bizonyos genetikai mutációkat tárolnak, így a meghatározásuk hasznos lehet különböző betegségek kiszűrésére. A haplotípusokat azonban direkt módon nem tudjuk vizsgálni, ezért az ember genetikai felépítéséből *(genotípus)* határozzuk meg őket statisztikai adatok alapján.

A probléma matematikai leírása [14]:

Adottak a genotípusok: n darab karakterlánc a {0, 1, 2} számokból, amelyek mindegyike m karakterből áll.

A haplotípusok szintén m hosszú karakterláncok, de csak a {0, 1} számokból állhatnak. Két haplotípus (h_1, h_2) *megmagyaráz* egy g genotípust, ha minden $0 < j \leq m$ helyiértékre teljesül a következő:

Ha $g_j = 0$, akkor $h_{1j} = h_{2j} = 0$

Ha $g_j = 1$, akkor $h_{1j} = h_{2j} = 1$

Ha $g_j = 2$, akkor $h_{1j} \neq h_{2j}$

Például a 0011 genotípust megmagyarázzák a {0011 és 0011} haplotípusok, a 0122 genotípust a {0101 és 0110} vagy a {0100 és 0111} haplotípusok.

A feladatunk egy olyan minimális számosságú haplotípus-halmazt találni, amelyből minden genotípushoz ki tudunk választani 2 (nem feltétlenül különböző) elemet úgy, hogy a kiválasztott elemek megmagyarázzák a genotípust.

Az 1.1. táblázatban egy 4 darab, 4 karakter hosszú genotípusból álló példán szemléltetem a problémát. A táblázat azt is tartalmazza, hogy az egyes genotípus milyen haplotípus-párokkal magyarázható meg.

Genotípus	Haplotípus 1	Haplotípus 2
0011	**0011**	0011
0101	**0101**	0101
0002	**0000**	**0001**
2012	0010 *0011*	1011 *1010*

1.1. táblázat. Haplotípus meghatározás

A **félkövérrel** jelölt haplotípusokat mindenképpen bele kell választanunk a halmazba, mert az első három genotípust csak velük magyarázhatjuk meg. Így a *0011*, amit már a halmazba választottunk, az *1010*-val együtt megmagyarázza az összes genotípust. Ezzel megtaláltuk a minimális halmazt. Természetesen ez egy nagyon egyszerű problémapéldány, bonyolultabb problémák esetén viszont kifizetődő a SAT-problémává konvertálás és az így történő megoldás.

1.2.3. Egyéb problémák

Nagyon sok feladat megfogalmazható SAT-problémaként, ezek közül csak párat sorolok fel:

- Gráfszínezés: Kiszínezhetőek-e egy gráf csúcsai k darab színnel úgy, hogy a szomszédos csúcsok között ne legyen azonos színű?
- Integer linear programming (ILP): Adott valahány egészértékű változó, és lineáris egyenlőtlenségek, amelyeknek teljesülnie kell. A feladat egy adott, a változókból álló lineáris célfüggvény maximalizálása.
- Klikk probléma: Egy gráfban maximális méretű teljes részgráf keresése.
- Prímtényezőkre bonthatóság: Egy adott számról el kell döntenünk, hogy prím-e.

2. fejezet

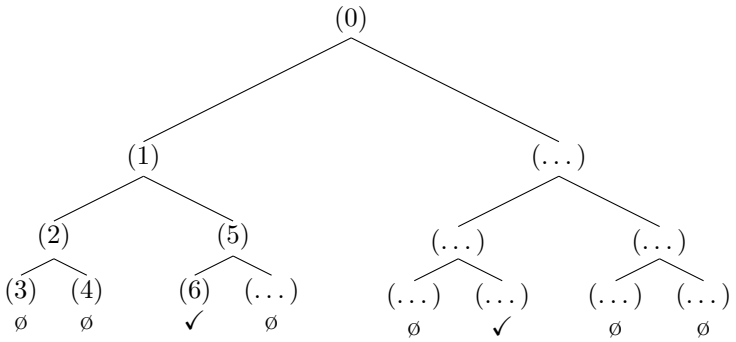
Elméleti háttér

2.1. Keresési fa

Keresési fának egy olyan fát nevezünk, amely rendelkezik egy kitüntetett csúccsal (gyökér), amiből elindulunk a keresés során, illetve a fa levelei lehetséges megoldásokat reprezentálnak, amelyek közül tetszőleges mennyiségű lehet valóban megoldás. Természetesen az is előfordulhat, hogy egyetlen megoldás sem létezik. A bejárás során a kiindulási csúcsból a levelek felé haladunk, azonban ez többféleképpen is megvalósítható az elágazások miatt. A keresés során tehát minden csúcsban döntéseket kell hoznunk arról, hogy merre kívánunk továbbhaladni. Ezeket a döntési lehetőségeket jelentik a csúcsokat összekötő élek.

A fa csúcsai a gyökértől való távolság szerint szinteken helyezkednek el: a fa tetejéből indulunk a keresésnél, és a legalsó szinten helyezkednek el a levelek. A fa szintjeit azzal azonosítjuk, hogy hány döntés után jutunk el rájuk. Ez alapján a fa gyökere a 0. szintnek tekinthető.

A legegyszerűbb algoritmus ilyen problémák megoldására az, ha elindulunk lefelé a fában, véletlenszerűen elágazva minden egyes szinten. Ha egy olyan csúcshoz jutunk el, ami ellentmondásra vezet, akkor visszalépünk a legutolsó olyan pontba, ahol még egyéb irányba is elágazhatunk, majd erre folytatjuk a keresést. Ez a *visszalépéses keresés* (backtrack search), ami tulajdonképpen a mélységi kereséshez hasonlóan járja be a fát.



2.1. ábra. Visszalépéses keresés

A 2.1. ábrán látható az algoritmus működése. A számok azt jelentik, hogy a backtrack milyen sorrendben látogatja meg a fa egyes csúcspontjait, az alsó szint alatti áthúzott kör azt, hogy az adott csúcs nem megoldás, a pipa pedig megoldást jelez. Az első konfliktus a 3. lépés után történt, ahonnan az algoritmus visszaugrott a 2. csúcsra. A 4. lépésben ismét konfliktus történt, ezért az 1. csúcsba tértünk vissza. Itt végül az 5. csúcson keresztül a 6. lépéssel megtaláltuk a megoldást.

A ...-tal jelölt csúcsokat az algoritmus nem látogatta meg. Mivel a backtrack az első megoldás megtalálása után kilép, a másik megoldást nem fedeztük fel.

A visszalépéses keresés egy egzakt algoritmus, ami azt jelenti, hogy garantáltan helyes választ ad a problémára. A véletlenszerű elágazások miatt természetesen ugyanazon a problémán is előfordulhatnak eltérések a futásidőben többszöri futtatás esetén. Az algoritmus rekurzív megvalósítását az alábbi pszeudokód írja le:

```
1  BTSearch(Problem P, Assignment A)
2  {
3          for(d=firstDecision; d<=lastDecision; d++)
4                  A=addDecision(A, d);
5                  if(NOT(contradiction(P, A))
6                          if(solved(P, A))
7                                  return true;
8                          else
9                                  if (BTSearch(P, A) == true)
10                                         return true;
11                 else
12                         A=undoDecision(A, d);
13         return false;
14 }
```

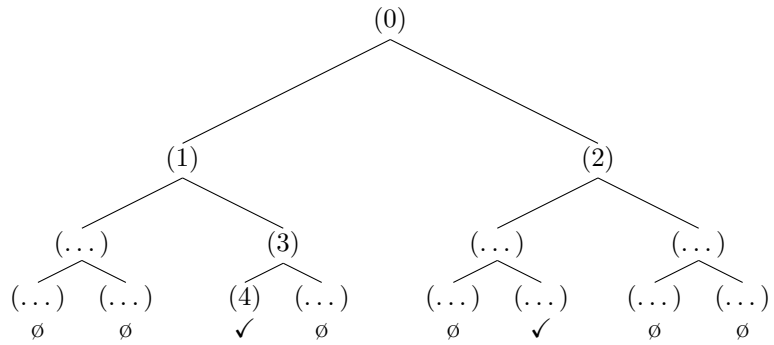
- Problem P: Az adott probléma.
- Assignment A: Az eddig meglevő döntések listája, a legelső függényhíváskor ez üres.
- firstDecision, lastDecision: A döntéseket egész számokként elképzelve az első és utolsó lehetséges értékek.
- addDecision(): Egy adott döntést hozzáad az aktuális döntések listájához.
- contradiction(): Igaz értéket ad vissza, ha talált ellentmondást az adott döntési lista alapján.
- solved(): Igaz értéket ad vissza, ha az adott döntések megoldást adnak a problémára.
- undoDecision(): Visszavonja a döntési lista legutolsó döntését, azaz egy szinttel feljebb kerülünk a keresési fában.

2.2. A best-first search

A *best-first search* a visszalépéses kereséssel ellentétben nem a mélységi keresésre épül, hanem mindig a feladat szempontjából legígéretesebbnek tűnő csomópontot fejti ki. Erre egy kiértékelő heurisztikus függvényt használ, ami a csomópontokhoz egy számot rendel azok tulajdonságai alapján. Ha ez a függvény azt próbálja közelíteni, hogy a keresési fában mennyire vagyunk közel a megoldáshoz, mohó best-first search algoritmusról beszélünk.

A 2.2. ábrán azt láthatjuk, amint az algoritmus az első lépés után úgy ítéli meg, hogy a lentebb lépés helyett kedvezőbb, ha inkább a másik azonos szinten levő döntést fejti ki. Ezután azonban mégis ígéretesebbnek tűnik az első döntésből továbblépés (3. csúcs). A következő, 4. lépésben meg is találjuk a megoldást.

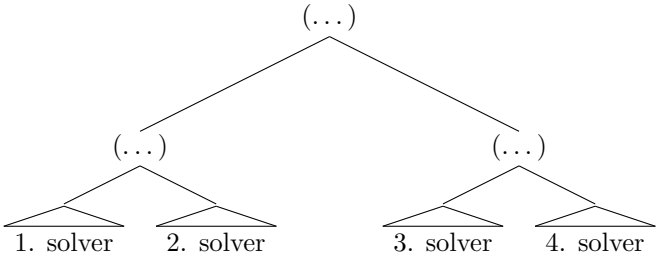
Ezen az egyszerű példán is láthatjuk az algoritmus főbb jellegzetességét: ahelyett, hogy csak ellentmondás esetén lépne vissza, minden egyes döntés után mérlegeli, hogy hol érdemes folytatni a keresést. A 2. lépés természetesen nem a legjobb döntés, hiszen egyből is megtalálhattuk volna a megoldást. Annak az oka, hogy mégis itt folytattuk a keresést az, hogy a csomópontokat kiértékelő



2.2. ábra. Best-first search

függvény csupán megbecsülni tudja a megoldás közelségét. A visszalépéses kereséshez hasonlóan ez az algoritmus is egzakt, a kiértékelő függvény csak az algoritmus futásidejét befolyásolja, de a bejárás ugyanúgy mindig jó eredményt fog adni.

A best-first search a gyakorlatban úgy is megvalósítható, hogy a keresési fát diszjunkt részekre osztjuk, és ezek mindegyikében létrehozunk egy-egy solverpéldányt. A keresés során ezek között váltogatunk az alapján, hogy melyik tűnik a legsikeresebbnek. Ebben a változatban már nem feltétlenül mérlegeljük minden lépés után, hogy melyik solvert futtassuk, hanem csak bizonyos időközönként kerül sor a kiértékelő függvény használatára. Így jelentősen csökkenthető a heurisztikával kapcsolatos járulékos erőforrásköltség (overhead).



2.3. ábra. Solverpéldányok a fában

A 2.3. ábrán látható ennek az alapötlete: úgy hozhatunk létre a legegyszerűbben diszjunkt részeket a fában, hogy a solverek kiindulópontjának egy, az eredetihez képest lentebbi szintet választunk. Az ezen a szinten levő minden egyes csúcsból el kell ilyenkor indítanunk egy solvert, különben nem fedjük le az egész keresési teret. A fenti példában négy részre osztottuk a fát, ennek megfelelően a 2. szintről kezdjük a keresést. Az alábbi pszeudokód a best-first search megvalósítását írja le:

```
1  BFSSearch(List L)
2  {
3          Solver currentSolver=firstOfList(L);
4          while(true)
5                  currentSolver.assignment=addDecision(currentSolver);
6                  markNode(currentSolver.assignment);
7                  if(solved(currentSolver))
8                          return true;
9                  if(contradiction(currentSolver))
10                         if(exhausted(currentSolver))
11                                 removeFromList(L, currentSolver)
12                                 if(isEmpty(L))
13                                         return false;
14                         else
15                                 undoDecision(currentSolver);
16                 if("limit for running the current solver is reached"
17                         OR "current solver was exhausted")
18                                 currentSolver.heuristic=setHeuristic(currentSolver);
19                                 currentSolver=findMaxHeuristic(L);
20
21 }
```

- List L: A keresésre használt solverek listája.
- Solver currentSolver: Az éppen futtatott solver, tagváltozói:

  assignment: Aktuális döntési lista.

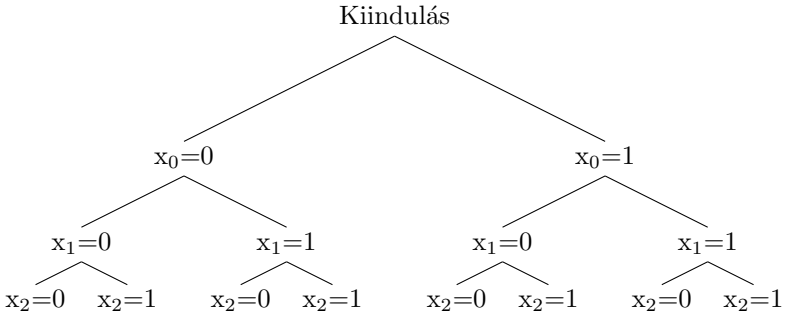
  heuristic: Mennyire jár közel a solver a megoldáshoz.
- firstOfList(): Visszaadja egy lista első elemét.
- addDecision(): Megvizsgálja a problémát és a döntések meglevő listáját, majd egy új döntést hoz létre.
- markNode(): Rögzíti, hogy az adott csúcsát meglátogattuk a fának.
- exhausted(): Igaz értéket ad vissza, ha a solver már teljes egészében bejárta a keresési tér neki szánt részét.
- removeFromList(): Eltávolítja a solvert a listából.
- isEmpty(): Igaz értéket ad vissza, ha üres a lista, ez azt jelenti, hogy a probléma megoldhatatlan.
- setHeuristic(): Kiértékelő függvény, amely az adott probléma és a döntések alapján próbálja meghatározni, hogy mennyire vagyunk közel a megoldáshoz.
- findMaxHeuristic(): A solverek listájából kiválasztja azt, amelyiket legközelebb érdemes futtatnunk.

2.3. SAT-probléma

2.3.1. Áttekintés

Adott egy logikai formula (F) konjunktív normálforma alakban. A probléma a következő: tudunk-e a változóknak úgy értéket adni, hogy F igaz lesz?
A probléma megértéséhez szükségünk van néhány definícióra [8]:
Literál: Egy változó vagy annak negáltja.
Klóz: Literálok diszjunkciója.
Konjunktív normálforma: Klózok konjunkciója.
Interpretáció: Egy formula minden egyes változójához értéket rendelünk. Egy n változóból álló formulának pontosan 2^n darab különböző interpretációja van.

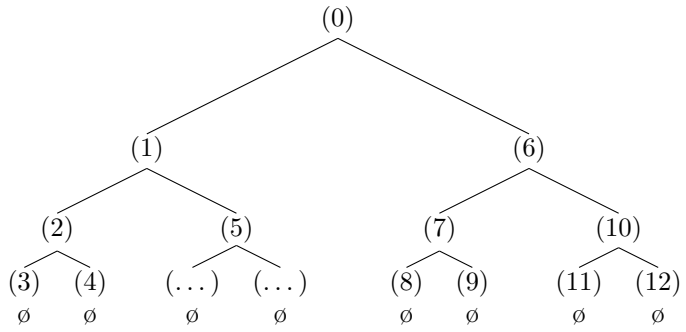
A formula interpretációit ábrázolhatjuk szemantikus fa formájában (2.4. ábra). Itt tehát láthatjuk, hogyan is néz ki a SAT-probléma esetében a keresési fa: az elágazások az egyes változóknak adott értékek alapján jönnek létre.



2.4. ábra. Szemantikus fa

Ezt a keresési fát kell bejárnunk a solver segítségével. Ha találunk egy olyan levelet, ahol az adott interpretációt kiértékelve a formula igaz lesz, megtaláltuk a megoldást *(satisfiable)*. A solverek hatékonyságát mutatja az, hogy mennyire hamar találják meg a fa ezen pontját.

Ha a levelek közül egyik sem ad megoldást, akkor a probléma megoldhatatlan *(unsatisfiable)*. Ennek ellenőrzéséhez legrosszabb esetben mind a 2^n levelet be kell járnunk, ami exponenciálisan sok lépést jelent. Legtöbbször azonban nincsen szükség az összes lépés végrehajtására: ha már a fa legalsó szintje előtt ellentmondásra akadunk, azzal egész területeket zárhatunk ki a keresésből. Ennek magyarázata, hogy ha már néhány változó adott értékei ellentmondásra vezetnek, ez nem oldható fel újabb változóknak történő értékadással.



2.5. ábra. Egy megoldhatatlan probléma keresési fája

Az 2.5. ábrán például már az 5. lépésben észrevettük a keresésnél, hogy a csúcs alatt levő levelek közül egyik sem megoldás.

2.3.2. DPLL

A SAT-probléma kapcsán a klózok leírására a következő definíciókat használjuk:
Satisfied klóz: Olyan klóz, amelynek legalább egy literálja igaz értéket kapott.
Unsatisfied klóz: Olyan klóz, amelynek már összes literálja hamis értéket kapott.
Unit klóz: Olyan klóz, amelyben már egy kivételével minden literálnak van értéke, de ezek mindegyike hamis.
Unresolved klóz: Amikor a klózra a fenti állapotok közül egyik sem teljesül.

Az egyszerűbb felépítésű SAT-solverek a DPLL (Davis–Putnam–Logemann–Loveland) [4] algoritmusra épülnek. Ez tulajdonképpen a fentebb részletezett backtrack egy kiegészített változata. A legfontosabb kiegészítés a *unit propagation*: minden értékadás után észleli a unit klózokat, és a következő értékadás ez alapján történik. Ennek oka, hogy a unit klóz utolsó literáljának igaz értéket kell adnunk, hogy a klóz satisfied legyen.

Néhány DPLL algoritmus tartalmazza az ún. *pure literal rule*-t is: Ha egy változó vagy csak ponáltan, vagy csak negáltan szerepel az egész kifejezésben, a keresés legelején megkapja első esetben az igaz, második esetben a hamis értéket, hiszen így az összes ezt tartalmazó klóz satisfied lesz. Ezeket a klózokat a későbbi keresés során nem kell figyelembe venni. A DPLL legegyszerűbb megvalósítása a backtrackhez hasonlóan rekurzívan történik, ezt szemlélteti a következő pszeudokód is:

```
1  Solve(Problem P)
2  {
3          A=assignPureLiterals(P);
4          return DPLL(P, A);
5  }
6
7  DPLL(Problem P, Assignment A)
8  {
9          unitPropagate(P, A);
10         if(contradiction(P,A))
11                 return false;
12         if(allVariablesAssigned(P,A))
13                 return true;
14         variable V=chooseVariable(P, A);
15         newAssignment1=addDecision(A, V);
16         newAssignment2=addDecision(A, NOT(V));
17         return( DPLL(P, newAssignment1) OR DPLL(P, newAssigment2));
18 }
```

- unitPropagate(): A unit klózokban fennmaradó változóknak értéket ad. Ha ennek során új unit klóz keletkezik, akkor a folyamat tovább folytatódik.
- assignPureLiterals(): A csak egyféle polaritással szereplő változóknak rögtön értéket ad.
- allVariablesAssigned(): Ellenőrzi, hogy minden változónak van-e már értéke. Mivel a konfliktusokat az ez előtti függvényhívással már kiszűrtük, a visszaadott igaz érték egy jó megoldást jelez.

- chooseVariable(): A probléma és az eddigi döntések alapján kiválasztja a következő változót, aminek értéket adunk.

2.3.3. CDCL

A CDCL (conflict-driven clause learning) a modern solverek által leggyakrabban használt algoritmus. Az előző fejezetben ismertetett DPLL-re épül, de azt jelentős módosításokkal használja. A legfontosabb különbség, hogy minden konfliktus után egy tanult klóz kerül hozzáadásra a formulához. A tanult klóz a konfliktushoz vezető értékadások, vagy ezek egy részének negáltját tartalmazza. Ennek a célja az, hogy elkerüljük ugyanazon konfliktus kétszeri előfordulását.

A keresés során az értékadásokhoz mindig kapcsolunk egy döntési szintet. Ez megadja, hogy a keresés alatt hanyadik döntéssel kapott értéket a változó. A unit propagation során értéket kapott változók nem rendelkeznek saját döntési szinttel, hanem a legutóbbi döntés szintjét kapják meg.

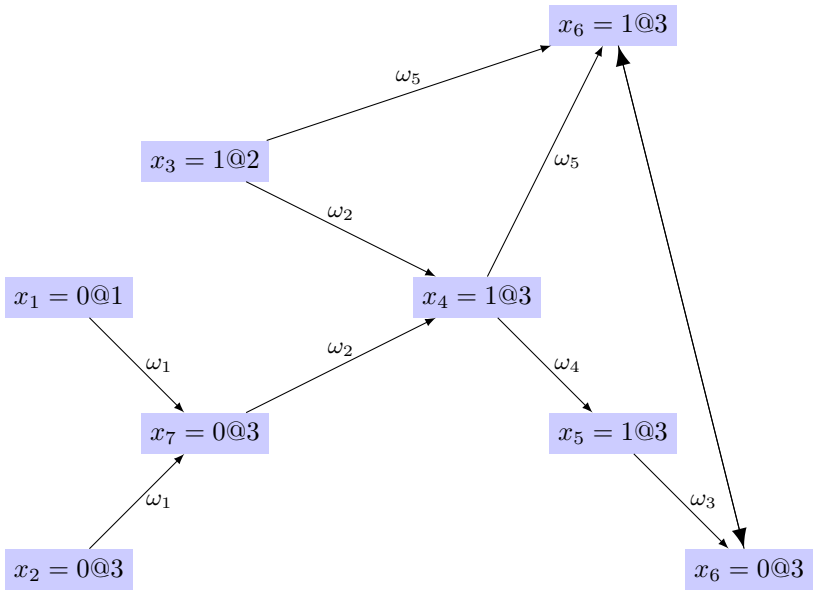
$x_i = v@d$
x_i: az értéket kapó változó
v: a kapott érték (1: igaz, 0: hamis)
d: az aktuális döntési szint

Lássuk az előbb bevezetett jelölések használatát és a klóztanulás megvalósítását egy konkrét példán keresztül:
$\omega_1 = (x_1 \vee x_2 \vee \neg x_7)$
$\omega_2 = (\neg x_3 \vee x_4 \vee x_7)$
$\omega_3 = (\neg x_5 \vee \neg x_6)$
$\omega_4 = (\neg x_4 \vee x_5)$
$\omega_5 = (\neg x_3 \vee \neg x_4 \vee x_6)$

Az eddigi döntések:
$x_1 = 0@1$
$x_3 = 1@2$

A jelenlegi döntés:
$x_2 = 0@3$

Ezek alapján felépítjük az implikációs gráfot (2.6. ábra), melynek csúcsai a meglevő értékadások. Az élek az alapján jönnek létre, hogy melyik értékadásból mi következik; feliratuk azt jelzi, hogy melyik unit klóz felelt a végpontnál szereplő értékadásért. Például x_7 azért kapott 0 értéket, mert a ω_1 klózban szereplő x_1 és x_2 literálok 0 értékéből adódóan a klóz unittá vált.



2.6. ábra. Az implikációs gráf

A gráfban felfedezhető a konfliktus: Az értékadásokból következik, hogy x_6-nak egyszerre kellene 0 és 1 értékkel rendelkeznie. Ez nyilvánvalóan nem lehetséges, ezért az eddig született döntések mellett a probléma megoldhatatlan. Tehát a három változó közül legalább az egyiknek a példában szereplőtől eltérő értéket kell kapnia, hogy ne jussunk ellentmondásra. Ez alapján a következő klózt tanulhatjuk meg: $\omega_l = (x_1 \vee x_2 \vee \neg x_3)$.

A tanult klóz tehát megakadályozza azt, hogy kétszer ugyanaz a konfliktus forduljon elő. Ez lehetővé teszi, hogy a konfliktus után ne közvetlenül a legutolsó döntés elé lépjünk vissza, hanem egy feljebb lévő döntési szintre (non-chronological backtrack).

A CDCL a klóztanulást a valóságban nem feltétlenül az itt bemutatt módszer szerint végzi. Léteznek kifinomultabb megoldások [4], amellyel a folyamatot optimalizálni lehet, azonban ezek ismertetésétől most eltekintek, hiszen a best-first search alkalmazásához ezekre nincsen szükség.

Fontos megemlíteni, hogy a CDCL algoritmusban nem tároljuk explicit módon az összes klózban szereplő mindegyik literál értékét. Ennek (leegyszerűsítve) az az oka, hogy a klózban mindig csak két literál értékét figyeljük, és ezek alapján következtetünk a klóz állapotára. Ez az ún. *watched literal scheme* [17]. Ez az adatstruktúra jelentősen gyorsítja az algoritmust, ugyanakkor, mint később látni fogjuk, számunkra korlátozást jelent a keresés során elérhető információk tekintetében.

A CDCL algoritmusban előszeretettel alkalmaznak restartokat is, akár a tanult klózok megtartása mellett. Minden egyes újraindításnál nő a limit, amit a solvernek a probléma megoldására adunk, hiszen egyébként előállhatna egy olyan helyzet, hogy soha nem jutunk eredményre. Nagy méretű problémák esetén a tanult klózok száma hatalmas lehet, ilyenkor célszerű a memóriaigényre való tekintettel alkalmanként törölnünk ezekből. Az algoritmus főbb lépéseit az alábbi iteratív kódrészlet szemlélteti:

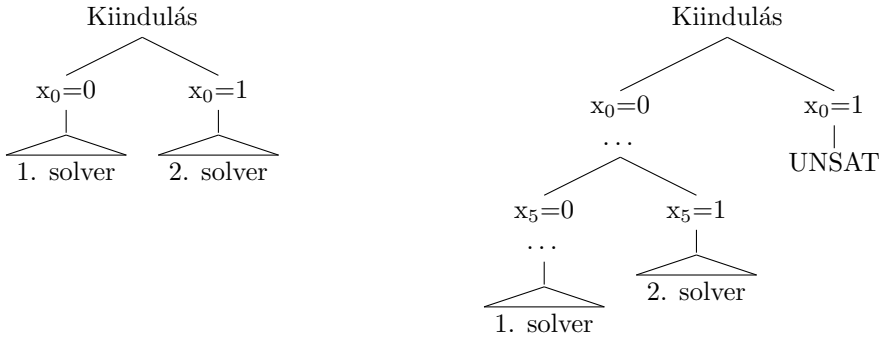
```
1  CDCL(Problem P)
2  {
3          Assignment A=empty;
4          while(true)
5                  if (unitPropagation(P, A) == CONFLICT)
6                          newDecisionLevel=analyzeConflict(P, A)
7                          if(newDecisionLevel<0)
8                                  return false;
9                          else backtrack(P, A, newDecisionLevel);
10                 if (allVariablesAssigned == true)
11                         return true;
12                 (Variable, Polarity)=branchingHeuristic(P, A);
13                 addDecision(A, Variable, Polarity);
14
15 }
```

- unitPropagation(): Itt a konfliktusok kiszűrésére is szolgál, ha egy változó két unit klóz utolsó elemeként is szerepel, méghozzá különböző polaritással, akkor ellentmondásra jutottunk.
- analyzeConflict(): Az adott konfliktust elemezve visszaad egy döntési szintet, ahova optimális backtrackelni, valamint hozzáad egy tanult klózt a problémához. Ha a visszaadott döntési szint kisebb, mint 0, a probléma unsatisfiable, hiszen a kiinduló csúcsnál feljebb nem tudunk visszalépni a keresésben.
- backtrack(): Visszatér a keresési fa egy adott szintjére, visszavonva az az utáni döntéseket.
- branchingHeuristic(): Megadja, hogy mely változónak és milyen értéket célszerű adni.

2.3.4. Kitekintés: Parallel SAT

A többmagos processzorok elterjedésével felmerült az ötlet, hogy a SAT-problémák megoldása úgy is gyorsítható, ha párhuzamosan több solvert is futtatunk. Erre két főbb megoldás létezik [16]:

Az első módszer a keresési tér szétosztása: a módszer lényege, hogy a keresési teret diszjunkt részekre osztjuk, és ezeken futtatjuk a solvereket. Ez a *guiding paths* (vezető utak) módszer segítségével történik: itt a változóknak nem csak az adott értékét mentjük el, hanem azt is, hogy mindkét lehetséges értékük ki lett-e próbálva a keresés során. Minden egyes olyan változó, amely még csak egyféle értékkel rendelkezett a keresés során, alkalmas arra, hogy a meglevő keresési teret két részre osszuk. Ezeket a változókat *nyitott*nak nevezzük.



2.7. ábra. A vezető utak

A solverek száma legtöbbször adott, a 2.7. ábrán például 2. Amikor az egyik solver befejezte a saját területén a keresést (tehát unsatisfiable értékkel tért vissza), akkor a keresési fában legfeljebb található nyitott változónál elágazik a keresés, és az eddig bejáratlan területet megkapja az éppen tétlen solver. A tanult klózok minden konfliktus után átadódnak a többi solvernek, azonban a nagy memóriaigény miatt ez csak korlátozott számban lehetséges.

A másik módszer a portfóliók használata. Itt úgy használjuk ki a párhuzamosítás adta lehetőségeket, hogy egyszerre futtatunk több solvert a problémán, de különböző paraméterekkel. Eltérőek lehetnek többek között az újraindítás gyakoriságában, a klóztanulás módszerében, vagy az értékadási heurisztikában. Ezek a solverek részben megosztják egymással a tanult klózokat, így segítve egymás munkáját. Ez elképzelhető ún. *master-slave* felépítésben is. A masterek végzik a tényleges keresést, és mindegyiknek van egy saját slave-je, amik megkapják a mastertől a konfliktusok listáját, és kifejezetten csak azokon a részeken keresnek, amelyek közelében a masternek konfliktusa volt. A master és a hozzá tartozó slave csak egymással osztják meg a tanult klózokat, és mivel hasonló területet járnak be, így ezek egymás számára relevánsabbak, mint véletlenszerű bejárások esetén [11].

Ezek az eredmények számomra is fontosak voltak a munkám során, annak ellenére, hogy nem alkalmazok direkt módon párhuzamosítást. A fenti módszerek közül én is alkalmazom a keresési tér szétosztását, hiszen ez több solver létrehozásánál lényeges kérdés. Az én algoritmusom azonban nem futtat egyszerre több solvert, hanem azt próbálja meghatározni, hogy az egyetlen futó solver éppen melyik legyen.

3. fejezet

Használt programok

3.1. MiniSat

A MiniSat [7] egy nyílt forráskódú, C++-ban írt SAT-solver, amelyet ketten készítettek el 2003-ban. Az első változat programkódja csupán 600 soros volt, mégis tartalmazta a CDCL solverek összes jellegzetességét. Célirányosan arra fejlesztették ki, hogy segítse a szakterületen a kutatásokat és a fejlesztéseket. Ebből adódóan jól dokumentált, könnyen átlátható szerkezetű és egyszerűen módosítható. Az újabb verziók már hosszabb kóddal rendelkeznek, de a solver lényegi része még mindig egy kicsivel kevesebb, mint 1 000 sorból áll. A MiniSat 2005-ben a SAT versenyen 4 kategóriában ért el első helyet, nagy fölénnyel legyőzve versenytársait. Számos kutatás eszközeként bizonyított már, ezért esett az én választásom is rá.

A MiniSat eredetileg Linux alatti használatra készült, de a futtatása Windows alatt is megoldható, azonban ehhez néhány Linux specifikus programrész eltávolítása szükséges. Ezek a programkód-részletek nélkülözhetőek a solver működéséhez, hiszen csak CPU és memória erőforráskorlátozásokra szolgálnak.

Innentől nincs más teendőnk, mint futtatni a MiniSatot. A program parancssori argumentumok formájában kapja meg a bemeneti problémát cnf formátumban és a kimeneti txt fájlt, amibe a megoldás kerül. Az argumentumok formátuma a következő: **minisat.exe** [**bemeneti fájl neve**] [**kimeneti fájl neve**] [**esetleges opciók**]

A MiniSat az objektumorientált elveket követve készült, tehát egy Solver nevű osztály szolgál a problémák megoldására. Ebben vannak eltárolva a cnf fájlból beolvasott adatok, és a keresés elindítása is egy metódus meghívásával történik. Ez a tulajdonság a kutatásom során jelentős szerepet játszott, hiszen így egyszerűen lehet több solvert párhuzamosan létrehozni.

A program támogatja az úgynevezett *assumption*ök használatát. Ez azt jeleneti, hogy megadhatjuk a solvernek, hogy néhány változó adott értékű legyen végig a keresés során, az pedig ezen feltételek mellett próbál megoldást találni. Fontos megjegyezni, hogy ha a solver unsatisfiable eredményre jut, az csupán azt jelenti, hogy az adott assumptionök mellett nem létezik megoldás. Ez a funkció a keresési tér szétosztásánál fog fontos szerepet játszani.

3.1.1. A cnf formátum

A SAT-problémák tárolása legtöbbször az előző fejezetben is említett cnf formátumban történik, melynek felépítése a következő:

```
c Minta
p cnf 5 3
1 −5 0
2 3 −1 0
−4 1 2 0
```

A c-vel kezdődő sorok kommentként tekintendők, a fájl feldolgozása közben lényegi jelentőségük nincs. Az első valójában feldolgozandó sor tartalma mindig adott: **p cnf [változók száma] [klózok száma]**

Ezután következnek a klózok: a változókat számokkal jelöljük, 1-től n-ig, ahol a probléma n változót tartalmaz. A negált literálok negatív előjellel szerepelnek. A literálokat egy-egy szóköz választja el, a klóz végét pedig egy 0 karakter jelzi, és általában egy sorban egy klóz található.

Ezek ismeretében a fenti minta fájl az alábbi problémát kódolja:
$(x_1 \vee \neg x_5) \wedge (x_2 \vee x_3 \vee \neg x_1) \wedge (\neg x_4 \vee x_1 \vee x_2)$

3.2. SAT-probléma generátorok

A solverek teszteléséhez nagy mennyiségű SAT-problémára van szükségünk, ezek előállítása azonban nem egyszerű feladat.

Az interneten található cnf fájlok szinte mindegyike túl könnyű (tizedmásodpercek alatt megoldható), vagy annyira nehéz, hogy percekig tartó futás után sem jut eredményre a solver. Ezért célszerű SAT-probléma generátort használni, amelytől alapvető elvárás, hogy tudja szabályozni a problémák nehézségét. Ugyancsak fontos, hogy tegye lehetővé külön megoldható, illetve nem megoldható problémák generálását.

A SAT-problémák generálására a legegyszerűbb módszer, ha adott számú változóból véletlenszerűen elkezdünk klózokat gyártani. Az így keletkező példányokról azonban nem tudjuk előre, hogy megoldhatóak lesznek-e. Ezt a megoldást tehát nem alkalmazhatjuk, ha csak megoldható problémákon szeretnénk tesztelni.

Egy másik módszer az, hogy előre kitalálunk egy megoldást, és a generálásnál csak azokat a klózokat tartjuk meg, amelyeket ez a megoldás igazzá tesz. Az így keletkező problémák mindenképpen megoldhatóak lesznek, azonban a gyakorlat azt mutatja [1], hogy ezek a példányok egyszerűbbnek bizonyulnak a hasonló méretű, de véletlen generáltakkal szemben.

Az lsencode [1] egy kifejezetten megoldható SAT-problémák generálására készült, C++-ban íródott szoftver. A működése legegyszerűbben latin négyzetek segítségével szemléltethető [10].

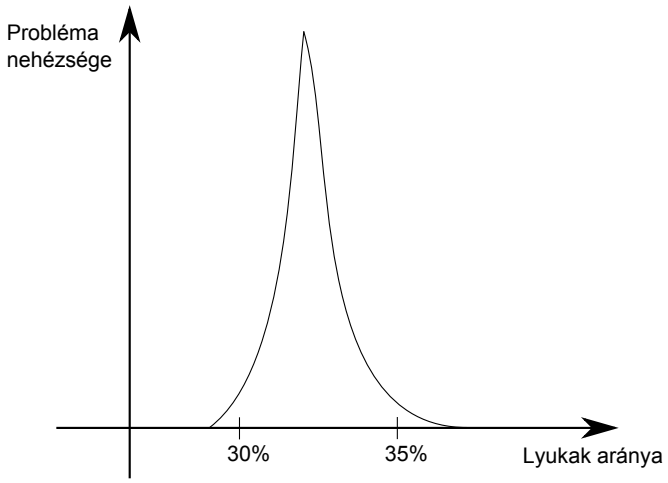
$$\begin{vmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 1 & 3 \\ 3 & 1 & 4 & 2 \\ 4 & 3 & 2 & 1 \end{vmatrix}$$

3.1. ábra. Egy latin négyzet

Ezek olyan $N \times N$ méretű táblázatok, amelyek N számmal vannak feltöltve úgy, hogy ugyanabban a sorban vagy oszlopban nem szerepelhet többször ugyanaz a szám. N-et a latin négyzet rendjének nevezzük, tehát például a 3.1. ábrán szereplő táblázat rendje 4. Az lsencode először egy teljesen kitöltött négyzetet generál, majd ebből kitöröl néhány elemet, így lyukakat képezve. A feladat a szabályok szerint kitölteni ezeket, akárcsak az ismert sudoku játékban. Az így generált problémáknak mindenképpen lesz legalább egy megoldása: az a teljes táblázat, amit éppen mi generáltunk. Ahogy

az 1.2-es fejezetben említettem, nagyon sok matematikai probléma értelmezhető SAT-ként, például a latin négyzetek is. Az lsencode ennek megfelelően a generált problémákat képes SAT-tá konvertálni.

Nyilvánvaló, hogy a probléma nehézsége a négyzet rendjének függvényében szigorúan monoton nő. A lyukak arányának a nehézséggel való kapcsolata azonban egészen másként alakul.



3.2. ábra. A probléma nehézségének alakulása

A 3.2. ábrán láthatjuk, hogy a probléma akkor a legnehezebb, hogyha a lyukak aránya a mezőkhöz képest 30% és 35% között van. Ennek az az oka, hogy ha nagyon kevés a lyuk, akkor a megoldás elég egyértelmű, ha pedig nagyon sok az üres hely, akkor sokféle jó kitöltés lehetséges. Viszonylag szűk az a terület, ahol az igazán nehéz problémapéldányok helyezkednek el.

A program használata parancssori argumentumok segítségével történik a következő módon: először generálunk egy adott méretű, kitöltött négyzetet, majd egy másik paranccsal lyukakat teszünk bele. Az lsencode pls formátumban tárolja a problémákat, azonban beépített konverterével átalakíthatjuk ezeket cnf fájlokká.

Az általam alkalmazott másik generátor a ToughSat [18]. Ez egy online használható, vagy akár le is tölthető program, ami rengeteg opcióval rendelkezik a generált példányok tulajdonságait illetően. Támogatja a már korábbiakban említett prímtényezőkre bontás problémák készítését is, amelyeket a teszteléskor én is előszeretettel használtam.

3.3. BCAT

A BCAT (Budapest Complexity Analysis Toolkit) [6] egy algoritmusok tesztelésére C++-ban kifejlesztett szoftvercsomag. 4 fő összetevőből áll: problémák (problems), algoritmusok (algorithms), elemzők (analyzers), és konverterek (converters). A probléma bármilyen jellegű lehet, támogatott a fájlból történő beolvasás és a generelás is. Az algoritmusok lehetnek teljes egészében a BCAT-be implementálva mint függvények, de akár egy külső exe fájlként is meghívhatók. A BCAT rögzíti az algoritmusok futási idejét, és az általuk talált megoldást is. Az elemzők szintén problémákra használhatóak, de nem megoldják azokat, hanem a problémák valamilyen tulajdonságáról gyűjtenek információt. Ez az algoritmusok hatékonyságának vizsgálatánál hasznos lehet az összefüggések kereséséhez. A konverterek egy problémát egy másik fajta problémává tudnak átalakítani, például ILP-t SAT-tá. Ezáltal egy adott solvert többféle problémára is tudunk használni.

A BCAT-nek egy konfigurációs fájlon keresztül mondhatjuk meg, hogy mely problémákra mely algoritmust, elemzőt vagy konvertert futtassa. Ez egy, a C++-hoz nagyon hasonló, leíró nyelven

történik. Fontos, hogy a generált problémák és az algoritmusok paraméterezhetők, és a BCAT a paraméterek összes lehetséges kombinációjára, az összes problémára lefuttatja az algoritmust. A tesztelés eredményét a BCAT egy csv (comma-separated values) fájlban tárolja, amelyből például Excel segítségével könnyedén készíthető grafikon.

A BCAT számos beépített problématípust és algoritmust tartalmaz, de természetesen saját algoritmusok tesztelésére is alkalmas. Ehhez nem kell mást tennünk, mint az UI (user interface) segítségével hozzáadni a kívánt algoritmust. Az algoritmusunknak megfelelően kell kezelnie a BCAT által számára átadott paramétereket (a konfigurációs fájlból), és meg kell oldanunk a BCAT-tel való kommunikációját is.

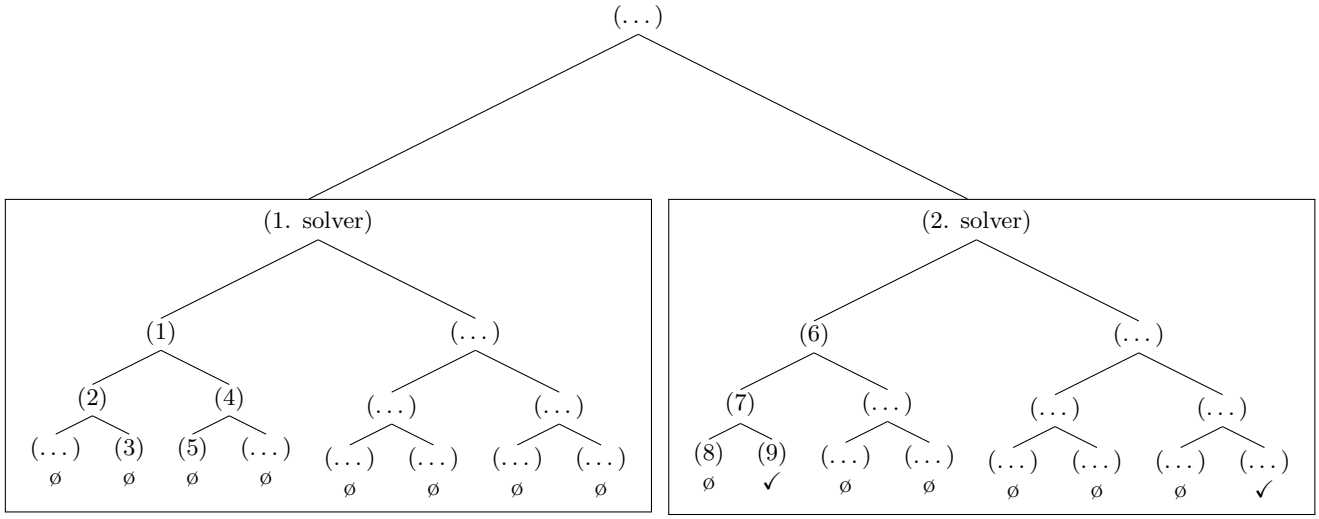
4. fejezet

Az algoritmus

4.1. Logikai szint

Ebben a fejezetben összefoglalom a best-first search segítségével gyorsított SAT-solver működését és előnyeit.

4.1.1. Az alapötlet



4.1. ábra. Best-first search 2 solverrel

Az algoritmus a következőképpen gyorsíthatja fel a SAT-solver működését: felbontjuk a keresési fát valahány részre, amelyek közül mindegyikben egy külön solver végzi a keresést. A 4.1. ábrán ezt 2 solverrel szemléltetem: az egyik elindítja a keresést, majd két ellentmondásra is akad (3. és 5. csúcs). Ezután a 2. solver fog futni, majd a 9. lépésben megoldást is talál. Nagyobb problémák esetén nyilvánvalóan többször is válthatunk solvert.

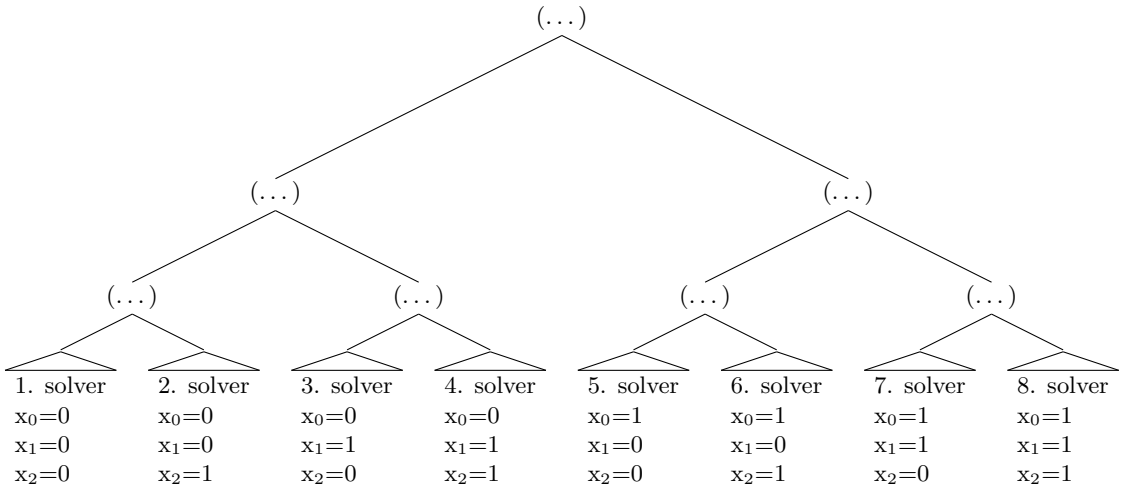
Vegyük észre, hogy a példában az első solver részfájában egyáltalán nincsen megoldás. Akár az is lehetséges, hogy egy több ezer változóból álló probléma esetén ugyanez fordul elő: az első értékadás hatására egy olyan részfába kerülünk, ahol nincsen megoldás, de a fa másik felében akár több is található. Ebben az esetben a fa egyik felét teljesen feleslegesen járjuk be, mielőtt átkerülünk a

másik részfára. Éppen emiatt találták ki a restartokat: a hosszú eredménytelen keresés arra utal, hogy a solver egy ilyen részfán tartózkodik. Hogyha az újraindítás után az első értékadás máshogy történik, valószínűleg egy kedvezőbb részfára kerülünk, ezáltal hamarabb találunk megoldást.

Ugyanerre a problémára nyújt megoldást a best-first search is, azonban több előnye is van. Egyrészt a solverváltás után nem vesznek el az addigi eredmények, tehát ha például szeretnénk visszatérni az 1. részfába, akkor nem a legelejéről kell elkezdenünk a keresést. Egy nagyobb probléma esetén, ahol sokszor oda-vissza váltogatunk a solverek között, ez sok időt takaríthat meg. Másrészt ez az algoritmus a restarttal ellentétben garantálja azt, hogy kikerülünk az eddig vizsgált részfából, és teljesen másik területet fogunk vizsgálni. Harmadrészt pedig, a solverekhez rendelt heurisztikus pontszám miatt mindig azon a részen fogunk keresni, ahol a legnagyobb valószínűséggel található megoldás.

4.1.2. A keresési tér szétosztása

A keresési tér szétosztásához el kell érnünk azt, hogy mindegyik solver csak a neki kijelölt részfán végezze a keresést. Ahogy a 3.1-es fejezetben említettem, ezt a MiniSat által biztosított assumptionök segítségével oldjuk meg. Ha ez a funkció nem létezne, egy saját készítésű függvénnyel kellene vezérelnünk az első néhány értékadást, hogy a solverek a megadott helyükre kerüljenek a fában. Ez esetben arra is kellene figyelnünk, hogy a solver a backtrack során ne ugorjon ki a neki szánt részfából. Az assumptionök könnyebbé teszik a fa részekre osztását, hiszen az említett feladatokat elvégzik helyettünk.



4.2. ábra. 8 solver megvalósítása assumptionökkel

A 4.2. ábra 8 solverre mutatja be azt, hogy hogyan lehetséges a keresési tér szétosztása assumptionök használatával: vesszük k változó összes interpretációját, ami a 2.3.1-es fejezetnek megfelelően éppen 2^k. Ezeket az interpretációkat elképzelhetjük úgy is, hogy k döntés után ennyi féle különböző csúcson lehetünk éppen a fában. Ha ezen csúcsok mindegyike egy solver számára jelenti a kiindulási pontot a keresésben, akkor sikerült a keresési teret 2^k részre osztanunk. Látható, hogy ha ezt a módszert használjuk, akkor a használt solverek száma *(numSolvers)* 2 hatványa kell, hogy legyen. Megoldható lenne tetszőleges számú solver használata is, azonban ez azt jelentené, hogy a solverek különböző döntési szintekről indulnak a keresésben (egy adott döntési szinten mindig 2^k csúcs található). Ez nem feltétlenül jelentene problémát, azonban a heurisztikát szükségtelenül bonyolulttá tenné.

A keresési fa feldarabolásánál a solverek száma egyértelműen meghatározza, hogy hány változót adunk az egyes solvereknek assumptionként. Arról azonban, hogy ezek mely változók legyenek,

szabadon dönthetünk. Ez azt jelenti, hogy azt, hogy mi alapján osztjuk szét a keresési teret, mi döntjük el ezen változók kiválasztásával. Felmerül a kérdés, hogy mely változókat célszerű erre a célra használni. A legegyszerűbb az, hogy 2^k solver esetén ezek legyenek $x_0, x_1, \ldots, x_{k-1}$. Egy másik lehetőség, hogy a legtöbb klózban előforduló változókat használjuk. Ez azért lehet jó ötlet, mert így valószínűleg egymástól nagyon különböző kiindulási alapot kapnak a solverpéldányok, és így talán pontosabban meg tudjuk határozni, hogy melyiket is érdemes futtatni. Arra, hogy ez valóban így van-e, a Mérések fejezetben fogunk választ kapni.

4.1.3. A heurisztika

A best-first searchben fontos elem a solverek sikerességét kiértékelő heurisztikus függvény. Ebben a fejezetben azt ismertetem, hogy milyen adatokat célszerű használni ebben a függvényben. Ehhez azt kell végiggondolnunk, hogy mi jelzi előre azt, ha egy SAT-solver közel jár a megoldáshoz. Azt is figyelembe kell vennünk, hogy ezen tényezők közül némelyikhez esetleg olyan mértékű adatgyűjtés is szükséges lehet, hogy a jelentkező overhead miatt azok alkalmazása nem éri meg.

- Jelenleg hány változónak van értéke? Minél több változó rendelkezik már valamilyen értékkel, annál valószínűbb, hogy néhány további döntés után megoldásra jutunk.
- Mennyi az eddig elért maximális mélység? Ha egy solver már nagyon mélyen is járt a keresési fában, valószínűsíthető, hogy hamar megoldást ad.
- Mennyi az átlagos mélység? A sikerességet szintén jelezheti az, hogy a solver átlagos mélysége a keresés során nagy.
- Milyen hosszúak a tanult klózok? A rövid tanult klózok jobban használhatóak, hiszen egy erősebb feltételt fogalmaznak meg, ezzel leszűkítve a keresést. Leghasznosabbak az egy literálból állók, hiszen egy ilyen az adott változó értékét egyértelműen meghatározza.

Ezen ötletek alapján a kiértékelő függvény a következőképpen épül fel:

$score = f(currentDepth, maxDepth, averageDepth, averageLearntClauseLength)$

Természetesen ez ebben a formában csak egy ötlet, aminek a helyességét a mérések során kell ellenőriznünk. Lehetséges, hogy a fenti feltételezések közül némelyik nem jelzi előre a megoldás közelségét, és azt is meg kell vizsgálnunk, nem szükséges-e ezek valamelyikéhez túl sok adatgyűjtés.

A CDCL leírásáról szóló fejezetben említettem, hogy a futás során nem minden adatot tudunk az egyes klózokról kinyerni a speciális adatszerkezet miatt. Ez azt jelenti, hogy nem tudjuk például megállapítani, hogy *egy klózon belül* hány változónak van értéke. Ezt tehát nem tudjuk alkalmazni a heurisztikában.

4.2. Implementáció

A BFS algoritmus MiniSatba való implementálása során a következő elveket követtem:

- A módosításokat próbáltam a main() függvényre korlátozni, azaz a solver lényegi működésében nem akartam jelentős módosításokat létrehozni.
- A MiniSat alapból meglevő funkcióit semmilyen módon nem szerettem volna korlátozni, csupán a problémák megoldási folyamatát gyorsítani.
- A változtatásokat úgy eszközöltem, hogy azok a MiniSat többi részével azonos adatszerkezeteket használjanak.

4.2.1. Az eredeti MiniSat működése

Ebben a fejezetben röviden ismertetem a MiniSat működését, hiszen ezen ismeretek fontosak ahhoz, hogy megértsük, miben különbözik az eredeti MiniSattól a best-first search.

A MiniSat által használt adattípusok a következők:
vec: Template-esített, tömböt megvalósító osztály.
Lit: Literálokat tárol.
CRef: Klózokat tárol.

A MiniSat main() függvényének részletes működése:

Először beállításra kerülnek az argumentumokban kapott opciók, majd, ha a bemeneti fájl elérési útvonala megfelelő, a MiniSat beolvassa egy Solver típusú objektumba a cnf fájlból a klózokat. Ezután hívódik meg a simplify() függvény, amely még a keresés előtt elvégez egy unit propagationt. Ha a solver ezalatt ellentmondásra jut, a probléma unsatisfiable.

Egyéb esetben elkezdődik a probléma megoldása. Ez a solveLimited() függvény segítségével történik, amelynek bemenő paramétere a keresésnél figyelembe veendő assumptionök listája. Ha nem szeretnénk assumptionöket használni, egy üres tömböt kell beadnunk a függvény paramétereként.

A solveLimited() függvény csupán annyit csinál, hogy átmásolja a paraméterében kapott assumption tömböt a Solver assumptions nevű változójába, ezután meghívja a solve_() függvényt. A fentiekből következik, hogy a solve_() meghívásának előkövetelménye, hogy az assumptions belső változó már inicializálva legyen.

A solve_() függvény tulajdonképpen egy while ciklusban futtatja a search() függvényt, ami a tényleges keresést végzi. Erre azért van szükség, mert a search() függvény paraméterként egy egész számot kap, és ennyi darab konfliktus után visszatér, de megoldás nélkül (l_Undef). Egészen addig maradunk a while cikluson belül, amíg a search() függvény vagy l_Unsat (unsatisfiable) vagy l_Sat (satisfiable) értékkel nem tér vissza. Minden egyes, eredményre nem jutó keresés után növelésre kerül a limitként adott konfliktusok száma, és újra meghívódik a search().

```
1  inline lbool Solver::solveLimited (const vec<Lit>& assumps)
2  {
3          assumps.copyTo(assumptions);
4          return solve_();
5  }
6
7  lbool Solver::solve_()
8  {
9          ...
10         while (status == l_Undef)
11         {
12                 ...
13                 status=search(rest_base*restart_first);
14                 ...
15         }
16         ...
17         return status;
18 }
```

Láthatjuk, hogy a seach() függvény a megengedett konfliktusok számát egy szorzat formájában kapja. Ezek közül a *restart_first* egy konstans, a *rest_base* minden visszatérés után nő, így biztosítva azt, hogy a függvény egyre nagyobb limittel rendelkezzen. Az alapbeállítás szerint a *rest_base* exponenciálisan növekszik, de ezt a felhasználó módosíthatja. A keresés befejezése után történik a megoldás output fájlba írása, a lefoglalt memóriaterület felszabadítása, majd a kilépés.

4.2.2. Inicializálás

A BFS implementációjához az első lépés az, hogy több solver legyen jelen egyszerre, és ezek mindegyikének legyen egy pontszáma, ami alapján meg tudjuk mondani, mennyire állnak közel a probléma megoldásához. Ezt a következő struktúra segítségével valósítottam meg:

```
1 struct BestFirst
2 {
3     Solver* Slv;
4     double score;
5 };
```

A BestFirst struktúra elemei tehát: egy Solver típusú pointer, és a solverhez tartozó aktuális pontszám. A programban használt solvereket egy BestFirst típusú pointerekből álló vektorban tárolom:

```
1 vec<BestFirst*> S;
```

A SAT-probléma fájlból történő beolvasása egy for ciklusban történik, méghozzá annyiszor, ahány solvert szeretnénk használni. Ezen természetesen lehetne gyorsítani: a Solver objektumhoz ajánlatos készíteni egy másoló konstruktort. A kutatásom során ettől eltekintettem, mert a méréseim azt mutatták, hogy a probléma beolvasása nagyságrendekkel kevesebb időt vesz igénybe, mint a megoldás tényleges keresése.

```
1 for(int ii=0; ii<numSolvers; ii++)
2 {
3     BestFirst* tmp=new BestFirst;
4     S.push(tmp);
5     S[ii]->score=INFINITE; //initialise score
6     S[ii]->Slv=new Solver;
7         ...
8 }
```

Maga a beolvasás teljesen analóg az eredeti MiniSat megoldásaival, annyi a különbség, hogy az *S[ii]* struktúra *Slv* tagváltozójába kell betöltenünk az adatokat. Az *INFINITE* egy predefiniált konstans, értéke a *score* változóban tárolható legnagyobb érték kell, hogy legyen. A solverek pontszámát azért erre az értékre inicializáljuk, hogy a következő solver kiválasztásnál preferálva legyenek a még nem használt solverek azokkal szemben, amelyek már a heurisztika alapján pontszámmal rendelkeznek. Ennek megfelelően csak akkor választunk solvert a heurisztika alapján, ha már mindegyikről rendelkezünk valódi adatokkal (amiket a futás során gyűjtöttünk).

Annak kiválasztásáról, hogy mely változókat adjuk meg assumptiönként, a későbbiekben lesz szó. Egyelőre tegyük fel, hogy egy occurdata nevű struktúratömb első $\log_2 numSolvers$ helyén álló változók adják meg az assumptionöket, az index nevű tagváltozójukban, a következőképpen: occurdata[0].index, occurdata[1].index stb. Az assumptionök létrehozását a következő kódrészlet szemlélteti:

$$0 = 000 \Rightarrow x0 \wedge x1 \wedge x2$$
$$1 = 001 \Rightarrow x0 \wedge x1 \wedge \neg x2$$
$$2 = 010 \Rightarrow x0 \wedge \neg x1 \wedge x2$$
$$3 = 011 \Rightarrow x0 \wedge \neg x1 \wedge \neg x2$$
$$4 = 100 \Rightarrow \neg x0 \wedge x1 \wedge x2$$
$$5 = 101 \Rightarrow \neg x0 \wedge x1 \wedge \neg x2$$
$$6 = 110 \Rightarrow \neg x0 \wedge \neg x1 \wedge x2$$
$$7 = 111 \Rightarrow \neg x0 \wedge \neg x1 \wedge \neg x2$$

4.3. ábra. A solver száma és a hozzá tartozó assumption-tömb

```
1  int binary;
2  Lit L;
3  int cntr;
4  for(int ii=0; ii<numSolvers; ii++) //every solver has assumptions
5  {
6      cntr=0; //setting assumption counter to zero
7      binary=ii; // assumption in binary form
8      for(int jj=numSolvers; jj>1; jj/=2) //assumption list for a solver
9      {
10         L=mkLit(occurdata[cntr++].index, binary%2); //making of a literal
11         S[ii]->Slv->assumptions.push(L); //add the literal
12         binary/=2; //next position
13     }
14 }
```

A kódrészlet némi magyarázatot igényel. Képzeljük el, hogy *numSolvers* darab solvert szeretnénk futtatni a keresési fán. Tudjuk, hogy az assumptionök száma $\log_2 numSolvers$, és hogy mindegyiknek különbözőnek kell lennie. Az assumptionök tulajdonképpen $\log_2 numSolvers$ hosszú bitsorozatok, amelyek megadják az egyes változók polaritását. Ezekből következik, hogy a solver sorszáma egyértelműen hozzárendel egy adott assumption-tömböt, hogyha a sorszámot $\log_2 numSolvers$ jegyű, kettes számrendszerbeli számként tekintjük. Ez látható a 4.3. ábrán.

A külső for ciklusban a solvereken iterálunk végig. Minden iteráció elején inicializáljuk a cntr változót, ami azt mutatja meg, hogy hanyadik assumptiont adjuk be az adott solvernek; továbbá a binary változóban tárolt assumption-kódot. A belső ciklusban történik maga az assumptions tömb felépítése. Ezzel a viszonylag egyszerű, rövid kódrészlettel történik tehát a keresési fa szétosztása.

4.2.3. Az assumptionökben szereplő változók meghatározása

Ebben a fejezetben röviden ismertetem, hogyan lehet egy cnf fájlból kinyerni azt, hogy melyik változó hányszor szerepel benne. Erre a 4.1.2-es fejezetben említettek miatt van szükség: meg szeretnénk vizsgálni, hogy érdemes-e a leggyakrabban előforduló változókat használni az assumptionökben.

A MiniSat cnf fájlból történő beolvasási folyamatát úgy módosítottam, hogy elmentse azt, hogy az egyes változók hányszor szerepelnek a kifejezésben összesen. A kódban a szürkével kiemelt részek az én módosításaim ennek a mérésére.

A megvalósításra a következő struktúra szolgált:

```
1  struct var_occur
2  {
3      int count; //how many times the variable occured
4      int index; //index of the variable
5  };
6
7  var_occur* occurdata;
```

```
1  parse_DIMACS_main(InputStream in, Solver S)
2  {
3          vec<Lit> lits;
4          while(in != EOF)
5          {
6                  if(in == 'p')
7                  {
8                          vars=parseInt(in);
9                          occurdata=new var_occur[vars];
10                         for(int ii=0; ii<vars; ii++)
11                         {
12                                 occurdata[ii].count=0;
13                                 occurdata[ii].index=ii;
14                         }
15                         clauses=parseInt(in);
16                 }
17                 else if (in == 'c' || in == 'p')
18                         skipLine(in);
19                 else
20                 {
21                         readClause(in, S, lits);
22                         S.addClause_(lits);
23                 }
24         }
25 }
26
27 static void readClause(InputStream in, Solver S, vec<Lit> lits)
28 {
29     while (true)
30     {
31         parsed_lit=parseInt(in);
32         if (parsed_lit == 0)
33                 break;
34         increaseCount(occurdata, var);
35         addLiteral(lits, parsed_lit);
36     }
37 }
```

- parseInt(): Beolvas egy egész számot egy fájlból.
- skipLine(): Átugorja a kommentként szolgáló sort.

- addClause(): Hozzáad egy literáltömböt a már meglevő klózok listájához.
- addLiteral(): Hozzáad egy literált a paraméterként kapott literáltömbhöz.
- increaseCount(): megnöveli eggyel az adott változó előfordulásának számát a nyilvántartásra szolgáló tömbben

Miután rendelkezésünkre állnak a szükséges adatok, az occurdata tömböt egyszerűen *count* szerint növekvő sorrendbe kell rendeznünk, a tömb elején így a leggyakoribb változók lesznek (az *index* tagváltozókban).

4.2.4. A keresés

A solverek és assumptionök inicializálása után megkezdődhet maga a best-first search. Ez egy ciklusban fut, ahol minden iteráció elején növeljük az adott solverpéldánynak adott konfliktuslimitet. Ezután a megadott számú konfliktus eléréséig keresünk az aktuális solverrel, és a visszatérési értéknek megfelelően halad tovább a program. Ez az érték háromféle lehet:
SAT: a solver egy megoldást talált
UNSAT: a solver befejezte a keresést, a hozzá rendelt részfában nincs megoldás
UNRESOLVED: a solver részfájában lehet megoldás, de még nem találtuk meg azt

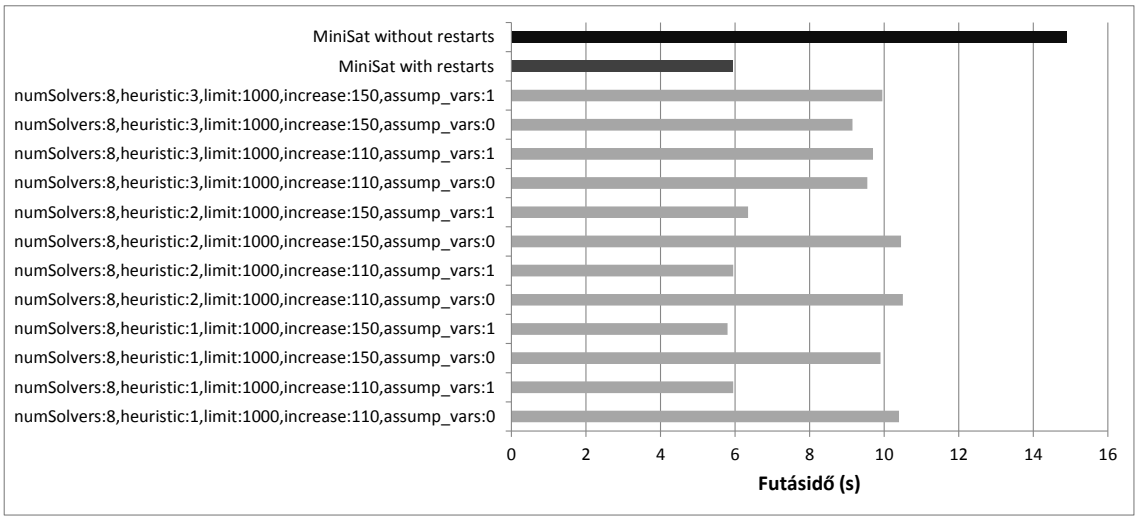
Ha egy solver SAT-tal tér vissza, akkor kiléphetünk a ciklusból, a probléma satisfiable. UNSAT érték esetében két eset lehetséges, attól függően, hogy van-e még működőképes solver. Amennyiben nincs, akkor a probléma unsatisfiable. Ha van még aktív solver, akkor kiválasztjuk, hogy melyik folytatja a keresést. Ha pedig UNRESOLVED a visszatérési érték, akkor a feladatunk pontszámot rendelni az aktuális solverhez és kiválasztani a következőként futtatásra kerülő példányt. A MiniSatba implementált best-first search algoritmus pszeudokódja ennek megfelelően a következőképpen alakul:

```
1  BFS(SolverList L)
2  {
3          Solver currentSolver;
4          boolean satisfied=false; // true, if a solver has returned SAT
5          boolean exists_active_solver=true; // true, if we have an active solver
6          while(NOT(satisfied) AND exists_active_solver)
7
8                  currentSolver=findMaxHeuristic(L);
9                  limit=increaseLimit();
10
11                 retCurr=solve(currentSolver, limit);
12
13                 if(retCurr == UNRESOLVED)
14                         setHeuristic(currentSolver);
15
16                 else if(retCurr == SAT)
17                         gatherStats(currentSolver);
18                         satisfied=true;
19
20                 else if(retCurr == UNSAT)
21                         gatherStats(currentSolver);
22                         deAllocate(currentSolver);
23                         exists_active_solver=activeCheck(L);
24
25         printStats();
26         if(satisfied)
27                 return true;
28         if(NOT(exists_active_solver))
29                 return false;
30 }
```

- Solver currentSolver: Az éppen futtatott solver.
- increaseLimit(): Minden futtatás előtt megnöveli a solvernek adott konfliktuslimitet.
- solve(): Keres a fában, visszatér ha eredményre (SAT vagy UNSAT) jutott, illetve ha eléri a paraméterben megadott konfliktusszámot (UNRESOLVED).
- setHeuristic(): Egy pontszámot rendel a solverhez, a futási adatok alapján
- findMaxHeuristic(): Egy egyszerű maximumkereséssel (pontszám alapján) kiválaszatja a következőként futtatásra kerülő solvert.
- gatherStats(): Elmenti a solver adatait a futás befejezése utáni statisztika készítéséhez.
- deAllocate(): Felszabadítja a solver által használt memóriaterületet.
- activeCheck(): Igaz értéket ad vissza, ha van még működő solverünk.
- printStats(): A futás végén kiírja a solverekkel kapcsolatos statisztikai adatokat.

5. fejezet

Mérések



5.1. ábra. A best-first search változatok eredményei

Az első grafikonon (5.1. ábra) 14 különböző solver változat futási idejének összehasonlítása látható. A solverek között szerepel az eredeti MiniSat restarttal és anélkül, valamint 12 best-first search változat is. Mindegyik solvert ugyanazon a 20 problémán futtattam, és a futásidők átlagát ábrázoltam a grafikonon. A tesztelésre használt problémapéldányokat az lsencode segítségével generáltam, ebből következik, hogy ezek mind megoldhatóak. A példányok mindegyike egy 35×35 méretű, 44% lyukat tartalmazó véletlenszerű latin négyzet SAT-problémaként való megfogalmazása. Az lsencode-ról szóló fejezetben láthattuk, hogy a lyukak száma drasztikusan befolyásolja a probléma nehézségét. Ennek megfelelően én viszonylag nagy lyukarányt használok, és a nehézséget inkább a táblamérettel szabályozom. A generált cnf fájlok kb. 4 000 változót és 40 000 klózt tartalmaznak.

Az előzetes vizsgálatok során számos heurisztikát kipróbáltam, ezek közül csak a 3 legsikeresebb vett részt a végső tesztelésekben.

A best-first searchöt alkalmazó solverek paraméterei a következők:

- numSolvers: a solverpéldányok száma
- heuristic: a heurisztika kiszámításának módja
  - 1: $score = averageDepth/averageLearntClauseLength$
  - 2: $score = currentDepth + maxDepth$
  - 3: $score = currentDepth * averageDepth/averageLearntClauseLength$

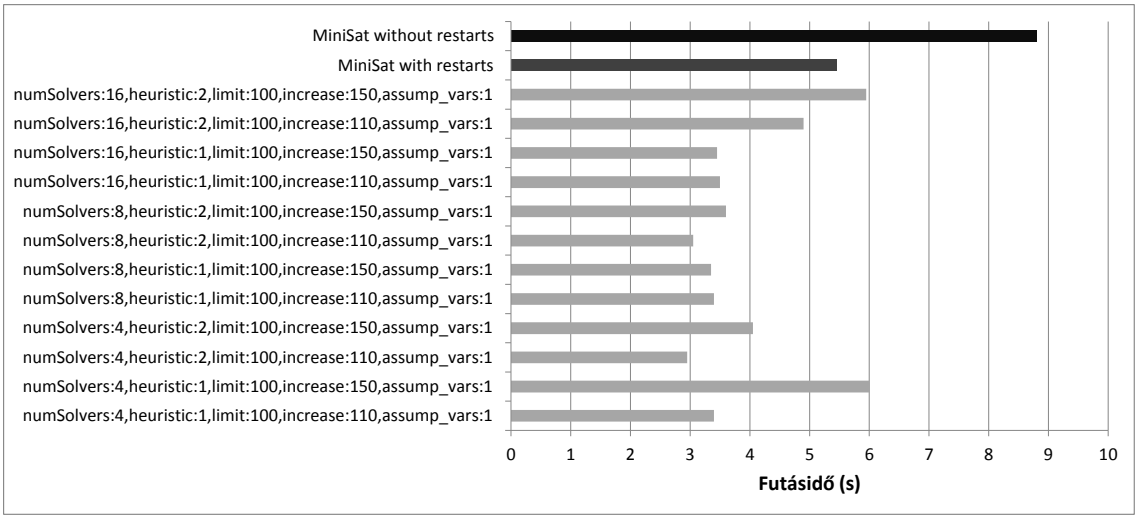
- limit: a solverek először hány konfliktusig futnak
- increase: a limitet mindig ilyen mértékben változtatjuk százalékban értve, pl.: 110 esetén mindig 1,1-szeres lesz a növekedés
- assump_vars: mely változók szerint kapják a solverek az assumptionöket

  0: első k változó

  1: leggyakoribb változók

Ez a teszt több szempontból is rendkívül tanulságos. A best-first search összes változata gyorsabb, mint a restart nélküli MiniSat, de az újraindításokat használó változatot nem igazán sikerült felülmúlni. Nyilvánvaló lett az is, hogy a leggyakrabban szereplő változókat célszerű az assumptionökben használni, és hogy a 3-as heurisztika nem megfelelő.

Láthatjuk tehát, hogy a best-first searchöt valahogyan kombinálni kellene az eredeti MiniSat újraindításaival. A további tesztekben a best-first search egy átalakított változatát használom, ahol a heurisztika kiszámítása után mindig újraindítom az adott solvert. Így valószínűleg mindig a legígéretesebb solvert fogjuk futtatni, azonban a restartokat is alkalmazzuk az algoritmus további gyorsítására.



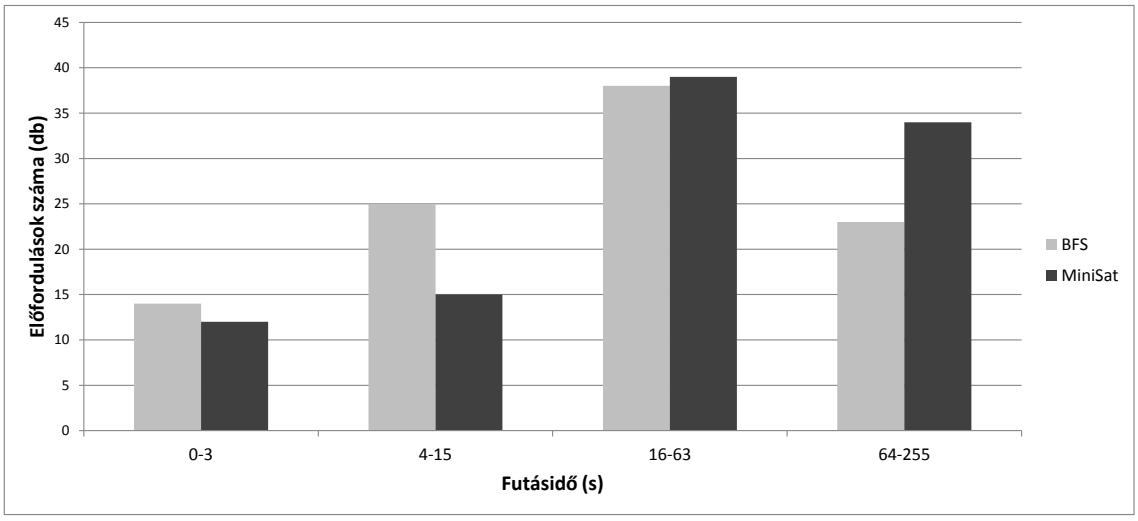
5.2. ábra. Az újraindításokkal gyorsított BFS

A második grafikon (5.2. ábra) az előzővel azonos körülmények között hasonlítja össze a módosított best-first searchöt az eredeti MiniSattal. Itt az előzőekben legsikeresebb 1. és 2. heurisztikákat használom, valamint csak a leggyakoribb változókat adom assumptionként. A tesztben azonban az előzőtől eltérően többféle solverszámot is kipróbálok.

Itt már a restartot alkalmazó MiniSat is alulmarad a legtöbb BFS változattal szemben, köszönhetően annak, hogy azokba is beleépítettem az újraindítást.

A harmadik grafikon (5.3. ábra) a legsikeresebb best-first search konfiguráció teljesítményét mutatja be 100 különböző problémán. Ennek paraméterei:

- numSolvers=4
- heuristic=2 ($score = currentDepth + maxDepth$)
- limit=100



5.3. ábra. A BFS és a MiniSat szórásának összehasonlítása

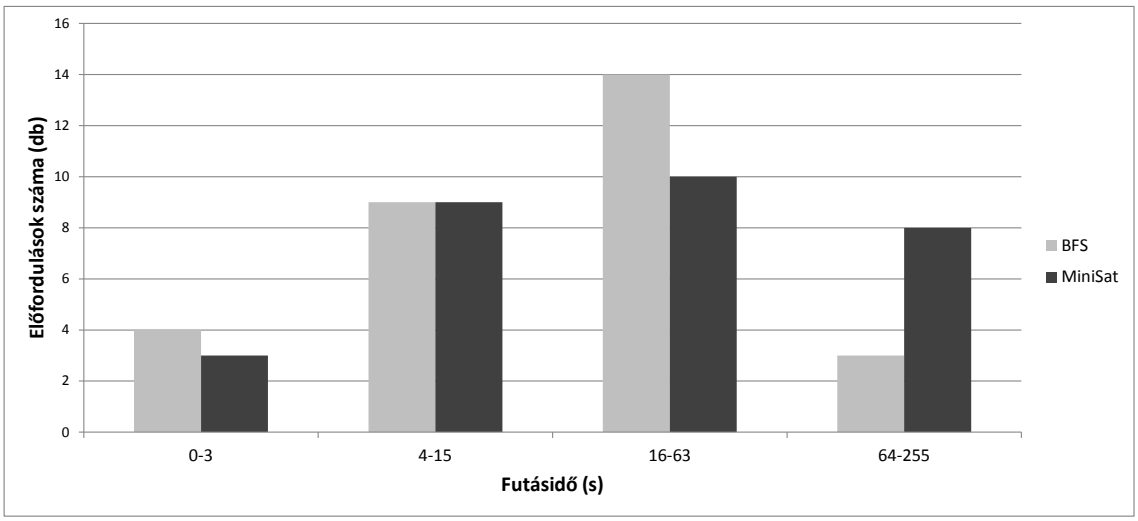
- increase=110
- assump_vars=1

A tesztelésre használt problémapéldányok immár nehezebbek: 39×39 méretű latin négyzetek, 44% lyukkal, szintén lsencode által generálva. A generált cnf fájlok így nagyjából 5 000 változót és 60 000 klózt tartalmaznak. Ez a grafikon más felépítésű, mint az előzőek; azt mutatja be, hogy melyik solver hányszor nyújtott egy adott időintervallumon belüli futásidőt. A jobb összehasonlíthatóság érdekében a vízszintes tengelyen logaritmikus skálát használok, ez azt jelenti, hogy az origótól távolodva az ábrázolt intervallumok nagysága exponenciálisan nő. Az ábrán jól látható, hogy a best-first search futási ideje lényegesen jobb az eredeti MiniSat restartokat használó változatánál. A BFS 39-szer futott 16 másodpercen belül, míg az eredeti MiniSat csupán 27-szer. A grafikonról ugyan nem olvasható le, de meggyőző adat az átlagos futásidő is: a BFS esetében ez $46,6$ másodperc, míg a MiniSatnál $59,9$.

A negyedik grafikonon (5.4. ábra) egy újabb, 30 problémán történő tesztelés eredményeit láthatjuk. Itt már nem lsencode által generált példányokat használtam, hanem nagy számok prímtényezőkre bontásának SAT-problémává konvertált formáját. A felbontandó számok mindegyike két, 1 400 000 körüli prímszám szorzataként állt elő. Ennek az az oka, hogy az így generált problémák nehézsége a tesztekhez éppen megfelelő. Ezek a cnf fájlok kb. 1 500 változót és 7 500 klózt tartalmaznak. A használt BFS algoritmus teljesen megegyezik az előző tesztben szereplővel.

Ezen az ábrán még jobban látszik, hogy a BFS segítségével jelentősen csökkenthető a szórás. Az eredeti MiniSatnál 8-szor nyúlt 64 másodperc fölé a futásidő, míg a BFS-nél ez csak 3-szor fordult elő. A best-first search 30 futtatásból 23-szor 4 és 63 másodperc között végzett. Az átlagos futásidők ennél a tesztnél a következőképpen alakultak: a MiniSat esetében $43,3$ másodperc, a BFS-nél pedig $24,9$.

A prímtényezőre bontás és az lsencode által generált problémák egymástól teljesen különbözőek, mégis ugyanaz a BFS konfiguráció mindkét esetben nagyon jó eredményt ad. A módszer ilyen szempontból tehát elég robusztusnak tekinthető.



5.4. ábra. Szórások alakulása prímtényezőre bontás-problémákon

6. fejezet

Befejezés

6.1. Eredmények

A TDK-dolgozatomban bemutattam a best-first search algoritmus alkalmazását SAT-problémákra. Ehhez először szükséges volt a témához kapcsolódó elméleti háttér körüljárására. Az alapfogalmak ismertetése után kitértem mind az általános, mind a SAT-problémához használt speciális kereső algoritmusok (DPLL és CDCL) felépítésére is. Ezután következett a használt programok bemutatása: a MiniSat az implementációhoz volt elengedhetetlen; az lsencode, a ToughSat és a BCAT pedig a tesztelésben segített. A kutatómunka leírása két szinten történt: egy absztrakt, logikai szinten, és a gyakorlati szinten. Ezen fejezet első részében kitalált ötleteket tehát a második részben implementáltuk a MiniSatba. Az ötletek sokaságából ki kellett azonban választani, hogy mik azok, amelyek valóban gyorsítják az algoritmus működését. Ennek menetét a mérésekről szóló fejezetben ismertettem, valamint a BFS hatékonyságát grafikonokkal támasztottam alá. Az adatgyűjtés nagy számú futtattás során, különböző típusú és nehézségű példányokon való teszteléssel történt, így valós képet adva a hatékonyságról.

A dolgozatból kiderült, hogy a MiniSat tényleg könnyen módosítható, akár az alapfelépítésétől lényegesen eltérő algoritmusok megvalósítására is képes. Ennek az oka, hogy objektumorientáltan, egymástól jól elkülönülő részek segítségével dolgozik. Fény derült továbbá arra is, hogy bár a MiniSat egy nagyon hatékony solver, best-first search alkalmazásával a működése tovább gyorsítható, méghozzá jelentős mértékben. Ehhez többféle paraméter kipróbálására volt szükség, a legnagyobb áttörést azonban az hozta, hogy a best-first searchöt kombináltuk a restarttal: ennek hatására az átlagos futásidő mintegy 36%-kal csökkent. Ezt az ötletet a tesztek során a MiniSat újraindításokat használó változatának eredményessége adta. A siker oka valószínűleg abban is keresendő, hogy a MiniSat eredetileg is gyakori újraindításokra lett tervezve, és mivel az újraindítás során megtartjuk a tanult klózok egy részét, a keresés részeredményei sem vesznek teljesen el.

6.2. Egyéb ötletek

Az általam vizsgált téma annyira szerteágazó, hogy a munkám során nem volt időm és lehetőségem minden egyes felmerült ötletet kipróbálni. Érdekes kérdés lehet az, hogy a problémák típusának és méretének függvényében hogyan célszerű megadni a best-first search paramétereit. Lehet, hogy például nagyobb méretű problémák esetén érdemes a limit növekedését máshogyan beállítani.

Egy másik lehetséges irány az algoritmus fejlesztésére a tanult klózok közössé tétele. Ez a MiniSatban éppen a szigorúan elkülönülő solver objektumok miatt nehézkes. Lehetséges azonban, hogy ez a funkció nagyban gyorsítaná a BFS működését. Szerencsés esetben egy 1 literálból álló tanult klóz akár a használt solverek felének munkáját feleslegessé tenné, ha ez a literál éppen szerepelt az

assumptionökben.

Az előbbieken kívül érdemes lehet még megvizsgálni az elért eredmények hardverfüggését is. Lehet, hogy az újraindítás és a best-first search kombinációja azért sikeres, mert a solverváltáskor a cache tartalmát egyébként is ki kell ürítenünk. Emiatt, ha ezzel egyidőben egy restartot is elvégzünk, nem növeljük jelentősen az overheadet.

Az itt felmerült kérdésekkel a jövőben behatóbban is kívánok foglalkozni, remélhetőleg a BFS további gyorsítását elérve.

Irodalomjegyzék

[1] Dimitris Achlioptas, Carla P. Gomes, Henry A. Kautz, and Bart Selman. Generating satisfiable problem instances. In Henry A. Kautz and Bruce W. Porter, editors, *AAAI/IAAI*, pages 256–261. AAAI Press / The MIT Press, 2000.

[2] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In Dominique Borrione and Wolfgang Paul, editors, *Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 254–268. Springer Berlin Heidelberg, 2005.

[3] Gilles Audemard and Lakhdar Sais. Circuit based encoding of CNF formula. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 16–21. Springer Berlin Heidelberg, 2007.

[4] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[5] Zoltán Ádám Mann and Tamás Szép. Accelerating backtrack search through a best-first-search strategy. Information Sciences, submitted.

[6] Zoltán Ádám Mann and Tamás Szép. BCAT: A framework for analyzing the complexity of algorithms. In *Proceedings of the 8th IEEE International Symposium on Intelligent Systems and Informatics*, pages 297–302, 2010.

[7] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[8] Szádeczky-Kardoss Ibolya Éva. Számításelmélet. `http://napszel.com/creations/szke_szamitaselmelet.doc`.

[9] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting Combinatorial Search Through Randomization. In *National Conference on Artificial Intelligence*, pages 431–437, 1998.

[10] Carla P. Gomes and David Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of Computational Symposium on Graph Coloring and Generalizations*, 2002.

[11] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and intensification in parallel SAT solving. In David Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 252–265. Springer Berlin Heidelberg, 2010.

[12] Katona Gyula, Recski András, and Szabó Csaba. *A számítástudomány alapjai*. Typotex, 2002.

[13] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman. Dynamic restart policies. In *Eighteenth national conference on Artificial intelligence*, pages 674–681, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[14] Inês Lynce and João Marques-Silva. Efficient haplotype inference with boolean satisfiability. In *Proceedings of the 21st national conference on Artificial intelligence - Volume 1*, AAAI'06, pages 104–109. AAAI Press, 2006.

[15] João Marques-Silva. Practical applications of boolean satisfiability. In *WODES 2008 : 9th International Workshop on Discrete Event Systems*, pages 74–80, 2008.

[16] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.

[17] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[18] Henry Yuen and Joseph Bebel. Tough SAT project. `http://toughsat.appspot.com/`.