This Week’s Lectures

- **Search and Discrepancies**
  - Recap of search and heuristics
  - Ideas and techniques
  - Implementation
- **Parallel Constraint Programming**
- **Parallel Search**

“It’s search, Jim, but not as we know it.”
Maintaining Arc Consistency (MAC)

- Achieve (generalised) arc consistency (AC3, etc).
- If we have a domain wipeout, backtrack.
- If all domains have one value, we’re done.
- Pick a variable (using a heuristic) with more than one value, then branch:
  - Try giving it one of its possible values (using a heuristic), and recurse.
  - If that failed, reject that value, pick a new value, and try again.
  - If we run out of values, backtrack.
Maintaining Arc Consistency (MAC)

Solver solver = new Solver("MAC");

/* variables and constraints go here */
solver.propagate();

if (mac(solver)) {
    System.out.println("satisfiable");
    for (int i = 0 ; i < solver.getNbVars() ; ++i)
        System.out.println(solver.getVar(i));
} else
    System.out.println("unsatisfiable");
Maintaining Arc Consistency (MAC)

```java
static boolean mac(Solver solver)
{
    /* find an uninstantiated variable */
    int branchVarNum = -1;
    for (int v = 0 ; v < solver.getNbVars() ; ++v)
        if (!solver.getVar(v).isInstantiated()) {
            branchVarNum = v;
            break;
        }

    /* every variable has domain size 1? */
    if (branchVarNum == -1)
        return true;
    return false;
}
```
Maintaining Arc Consistency (MAC)

```java
/* try giving it each value, in turn */
DisposableValueIterator values = ((IntVar) solver.getVar(branchVarNum)).getValueIterator(true);

while (values.hasNext()) {
    int v = values.next();
    solver.getEnvironment().worldPush();
    try {
        /* try the assignment (hack!), propagate, recurse */
        IntVar var = (IntVar) solver.getVar(branchVarNum);
        var.instantiateTo(v, null);
        solver.propagate();
        if (mac(solver))
            return true;
    } catch (ContradictionException e) { }
    solver.getEnvironment().worldPop();
}
values.dispose();
```
Maintaining Arc Consistency (MAC)

/* we ran out of values */
return false;
}
Search as a Tree

- Circles are recursive calls, triangles are ‘big’ subproblems.
- Heuristics determine the ‘shape’ of the tree:
  - Variable-ordering heuristics determine the number of children at each level.
  - Value-ordering heuristics determine the paths explored.
- MAC is like Depth-First Search (DFS).
Heuristics and Discrepancies

- If our value-ordering heuristics are perfect, and an instance is satisfiable, we walk straight to a solution by going left at every level.
- If an instance is unsatisfiable, perfect variable-ordering heuristics would give the smallest possible search tree.
- But heuristics aren’t perfect…
- We call going against a value-ordering heuristic choice a “discrepancy”.
Two Claims Regarding Value-Ordering Heuristics

1. The total number of discrepancies to find a solution is usually low (our value-ordering heuristics are *usually* right).

2. Value-ordering heuristics are most likely to wrong higher up in the tree (there is least information available when no or few choices have been made).
So What?

- If these claims are true, depth-first search is a bad idea: we’re committing entirely to the first decision made, which is most likely to be wrong.
Limited Discrepancy Search

- First, search with no discrepancies.
- Then search allowing one discrepancy.
  - First try one discrepancy at the top.
  - Then try one discrepancy at the second level.
  - Then try one discrepancy at the third level.
  - ...
- Then search allowing two discrepancies.
  - At the top, and at the second level.
  - Then at the top, and at the third level.
  - ...
  - Then at the second level and the third level.
  - ...
  - ...

Figure 2: Execution trace of LDS.
Completeness

- **Complete**: yes means yes, no means no.
- **Incomplete**: yes means yes, no means maybe.
- LDS is *quasi-complete*: if the total number of discrepancies is allowed to go high enough, it is complete.
What About Non-Binary Trees?

- We can rewrite our search tree to be binary. Instead of branching on each value for a variable in a loop, pick a variable and a value, and branch twice:
  - Yes, the variable takes that value.
  - No, the variable does not take that value.
- But this means that giving the 10th value to a variable counts as 9 discrepancies. Is this good or bad?
- Alternatively, we can treat the left branch as no discrepancy, and all right branches as discrepancies.
Improved Limited Discrepancy Search

LDS explores some parts of the search tree more than once.

Improved Limited Discrepancy Search (ILDS) does less repeated work.
Improved?

Figure 1: Paths with 0, 1, 2, and 3 discrepancies

Figure 2: Execution trace of L.D.S.
So is the Second Claim Important?

**Limited Discrepancy Search Revisited**

PATRICK PROSSER and CHRIS UNSWORTH, Glasgow University

Harvey and Ginsberg's limited discrepancy search (LDS) is based on the assumption that costly heuristic mistakes are made early in the search process. Consequently, LDS repeatedly probes the state space, going against the heuristic (i.e., taking discrepancies) a specified number of times in all possible ways and attempts to take those discrepancies as early as possible. LDS was improved by Richard Korf, to become improved LDS (ILDS), but in doing so, discrepancies were taken as late as possible, going against the original assumption. Many subsequent algorithms have faithfully inherited Korf's interpretation of LDS, and take discrepancies late. This then raises the question: Should we take our discrepancies late or early? We repeat the original experiments performed by Harvey and Ginsberg and those by Korf in an attempt to answer this question.

In moving from LDS to ILDS, we have inadvertently lost the assumption underpinning limited discrepancy search (i.e., that costly heuristic errors are made early on in the search process). We have put this assumption to the test with two variants of ILDS, one taking discrepancies early and one taking discrepancies late. In our number partitioning experiments, we see clear regions were early beats late, and that is when problems are hard and satisfiable, suggesting that Harvey and Ginsberg's assumption holds in that region. Conversely, late beats early when number partitioning problems are easy and satisfiable, refuting Harvey and Ginsberg's assumption in that region. However, in both regions, the improvements in performance are either relatively small or absolutely small. Since a static variable ordering is imposed, both versions must take the same number of discrepancies to find a solution, and thus the gain can only be found in the last probe. An analysis of the data revealed that in soluble instances, the majority of search effort occurs in the last probe. This was somewhat surprising.
If the second claim *is* important, why not emphasise it more?

Depth-bounded discrepancy search considers $k$ discrepancies, but only at depth up to $k - 1$ (but in which order?).

![Diagram of DDS on a binary tree of depth 4.](image)

Figure 2: DDS on a binary tree of depth 4.
What About Unsatisfiable Instances?

- Discrepancy searches do more total work if there is no solution.
- There are search variants which are better for unsatisfiable instances, and for optimisation problems.
  - One useful observation: if no leaf nodes are found when exploring for $k$ discrepancies, then no leaf nodes will be found when exploring for $k + 1$ discrepancies, so we can stop immediately.
Using LDS?

[gecode-users] Important: Licensing information regarding LDS

Christian Schulte cschulte at kth.se
Sat Oct 9 20:35:51 CEST 2010

---

* Previous message: [gecode-users] Gecode 3.4.2 released
  * Messages sorted by: [ date ] [ thread ] [ subject ] [ author ]

---

Dear all,

We have been informed by one of the patent holders that LDS (limited discrepancy search) is patented in the United States of America. While this does not pose an issue per se for us as developers (the MIT license under which Gecode is released makes that clear), it does to you as users.

After weighing the merits of offering LDS in Gecode with the effort for obtaining a non-commercial license, we have decided to remove LDS from Gecode. Gecode 3.4.2 removes LDS.

This decision just reflects our current understanding of how useful LDS is compared to any effort regarding licensing.

If you feel strongly about having LDS for Gecode available, we might make it available as an additional contribution with an explicit statement that it is patented in the United States of America. The patent holder has informed me that he is willing to give a non-commercial license to anybody who seeks one.

Please also take this information into account when using versions of Gecode before 3.4.2: you need to have a license to use LDS in the United States of America.

Christian
A Rough Hack at DDS in Choco

```java
static boolean dds(Solver solver, int depth, int cutoff) {
    /* find an uninstantiated variable */
    int branchVarNum = -1;
    for (int v = 0; v < solver.getNbVars(); ++v) {
        if (!solver.getVar(v).isInstantiated()) {
            branchVarNum = v;
            break;
        }
    }
    /* every variable has domain size 1? */
    if (branchVarNum == -1)
        return true;
}
```
A Rough Hack at DDS in Choco

/* try giving it each value, in turn */
DisposableValueIterator values = ((IntVar) solver.getVar(
    branchVarNum)).getValueIterator(true);

boolean first = true;
while (values.hasNext()) {
    int v = values.next();
}
A Rough Hack at DDS in Choco

/* if we’re one before the cutoff, don’t consider the first value */
if (depth != (cutoff - 1) || !first) {
solver.getEnvironment().worldPush();

try {
    /* try the assignment, propagate, recurse */
    IntVar var = (IntVar) solver.getVar(branchVarNum);
    var.instantiateTo(v, null);
    solver.propagate();

    if (dds(solver, depth + 1, cutoff))
        return true;
}
catch (ContradictionException e) { }

solver.getEnvironment().worldPop();
}
A Rough Hack at DDS in Choco

/* If we’re below the cutoff, don’t consider * any remaining values */
if (depth >= cutoff)
    break;

    first = false;
}
values.dispose();

/* we ran out of values */
return false;
A Rough Hack at DDS in Choco

```java
boolean success = false;
for (int cutoff = 0; cutoff < solver.getNbVars(); ++cutoff) {
    if (dds(solver, 0, cutoff)) {
        success = true;
        System.out.println("satisfiable");
        for (int i = 0; i < solver.getNbVars(); ++i)
            System.out.println(solver.getVar(i));
        break;
    }
}

if (!success)
    System.out.println("unsatisfiable");
```
What are the two assumptions regarding value ordering heuristics which underly limited discrepancy search?

Why are discrepancy searches a bad choice if instances are expected to be unsatisfiable?

When using MAC, what effect do value ordering heuristics have on unsatisfiable instances?