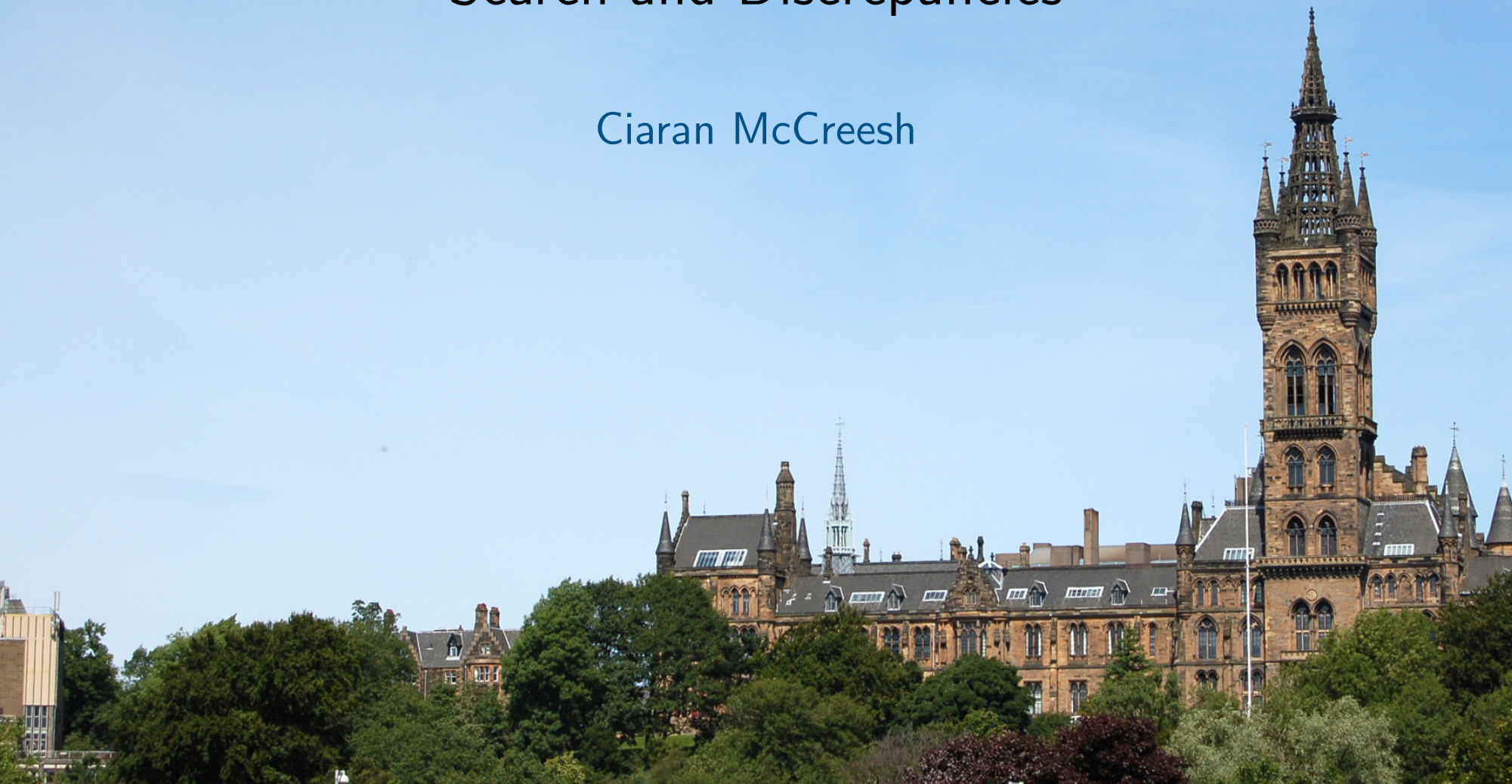
University of Glasgow

Search and Discrepancies

Ciaran McCreesh



This Week's Lectures

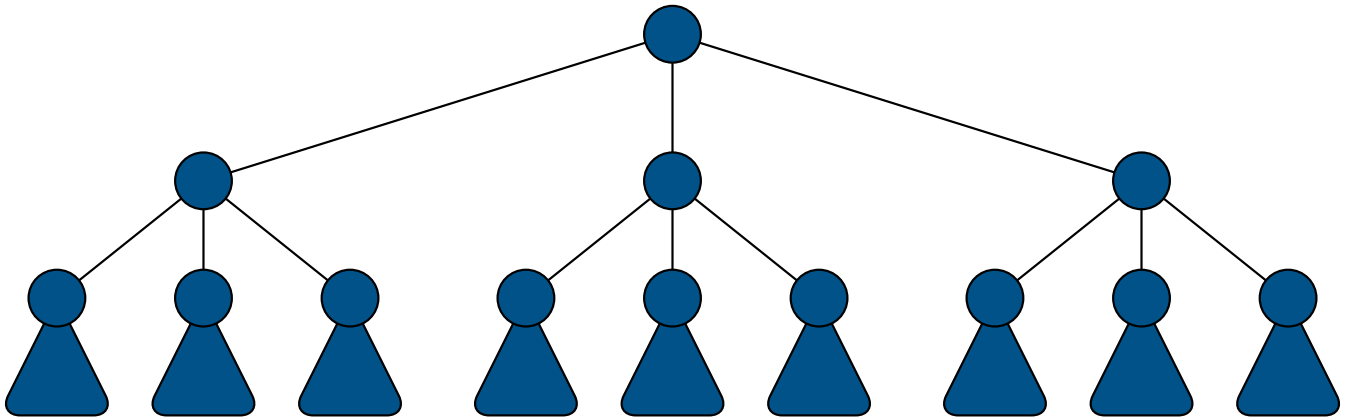
- Search and Discrepancies
  - Recap of search and heuristics
  - Ideas and techniques
  - Implementation
- Parallel Constraint Programming
- Parallel Search

"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.

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).

Limited Discrepancy Search

William D. Harvey and **Matthew L. Ginsberg**
CIRL
1269 University of Oregon
Eugene, Oregon 97403
U.S.A.
ginsberg@cs.uoregon.edu

Let us return to the search problems for which the successor ordering heuristic is a good one. Our intuition is that, when 1-samp fails, the heuristic probably would have led to a solution if only it had not made one or two "wrong turns" that got it off track. It ought to be possible to systematically follow the heuristic at all but one decision point. If that fails, we can follow the heuristic at all but two decision points. If the number of wrong turns is small, we will find a solution fairly quickly using this approach.

If the subtree below a mistake is large, chronological backtracking will spend all of the allowed run time exploring the empty subtree, without ever returning to the last decision that actually matters. If one is counting on the heuristics to find a goal node in a small fraction of the search space, then chronological backtracking puts a tremendous burden on the heuristics early in the search and a relatively light burden on the heuristics deep in the search. Unfortunately, for many problems the heuristics are *least* reliable early in the search, before making decisions that reduce the problem to a size for which the heuristics become reliable. Because of the uneven reliance on the heuristics, it is unlikely that chronological backtracking is making the best use of the heuristic information.

Two Claims Regarding Value-Ordering Heuristics

> Let us return to the search problems for which the successor ordering heuristic is a good one. Our intuition is that, when 1-samp fails, the heuristic probably would have led to a solution if only it had not made one or two "wrong turns" that got it off track. It ought to be possible to systematically follow the heuristic at all but one decision point. If that fails, we can follow the heuristic at all but two decision points. If the number of wrong turns is small, we will find a solution fairly quickly using this approach.

Two Claims Regarding Value-Ordering Heuristics

> If the subtree below a mistake is large, chronological backtracking will spend all of the allowed run time exploring the empty subtree, without ever returning to the last decision that actually matters. If one is counting on the heuristics to find a goal node in a small fraction of the search space, then chronological backtracking puts a tremendous burden on the heuristics early in the search and a relatively light burden on the heuristics deep in the search. Unfortunately, for many problems the heuristics are *least* reliable early in the search, before making decisions that reduce the problem to a size for which the heuristics become reliable. Because of the uneven reliance on the heuristics, it is unlikely that chronological backtracking is making the best use of the heuristic information.

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.
  - ...
- ...

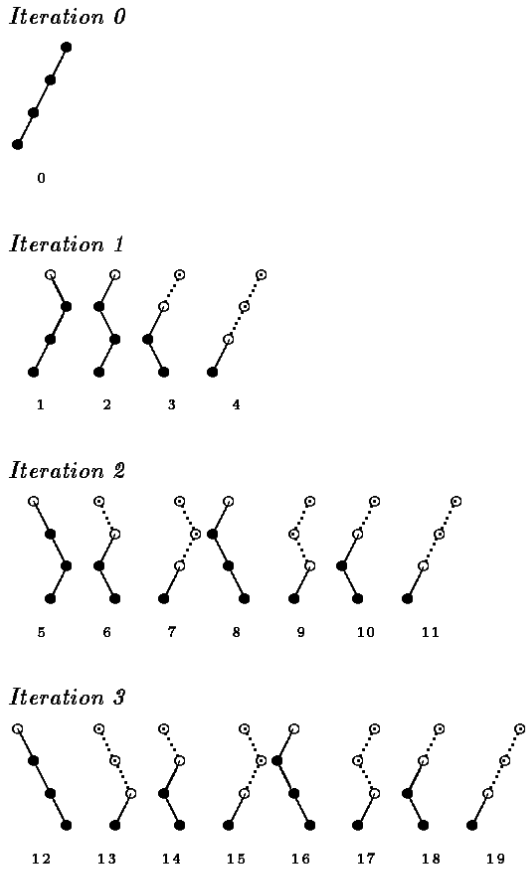


Figure 2: Execution trace of LDS.

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?

For simplicity, we will consider only the case of a full binary tree.

- 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?

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 Limited Discrepancy Search

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
korf@cs.ucla.edu

The main drawback of the original formulation of LDS, OLDS, is that it generates some leaf nodes more than once. In particular, the iteration for k discrepancies generates all paths with k *or less* right branches. Thus, each iteration regenerates all the paths of all previous iterations. For example, OLDS generates a total of 19 paths on a depth-three binary tree, only 8 of which are unique, as shown in Figure 2 of (Harvey and Ginsberg 1995). As an extreme case, while the rightmost path is the only new path in the last iteration, OLDS regenerates the entire tree on this iteration.

Given a maximum search depth, the algorithm can be modified so that each iteration generates only those paths with *exactly* k discrepancies. This is done by

Improved Limited Discrepancy Search

The main drawback of the original formulation of LDS, OLDS, is that it generates some leaf nodes more than once. In particular, the iteration for k discrepancies generates all paths with *k or less* right branches. Thus, each iteration regenerates all the paths of all previous iterations. For example, OLDS generates a total of 19 paths on a depth-three binary tree, only 8 of which are unique, as shown in Figure 2 of (Harvey and Ginsberg 1995). As an extreme case, while the rightmost path is the only new path in the last iteration, OLDS regenerates the entire tree on this iteration.

Given a maximum search depth, the algorithm can be modified so that each iteration generates only those paths with *exactly* k discrepancies. This is done by

Improved?

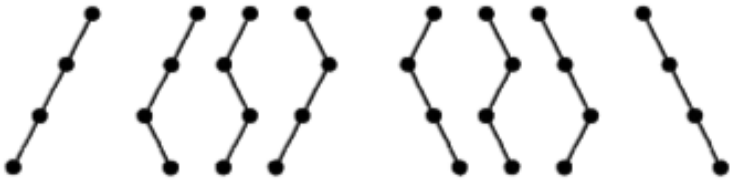


Figure 1: Paths with 0, 1, 2, and 3 discrepancies

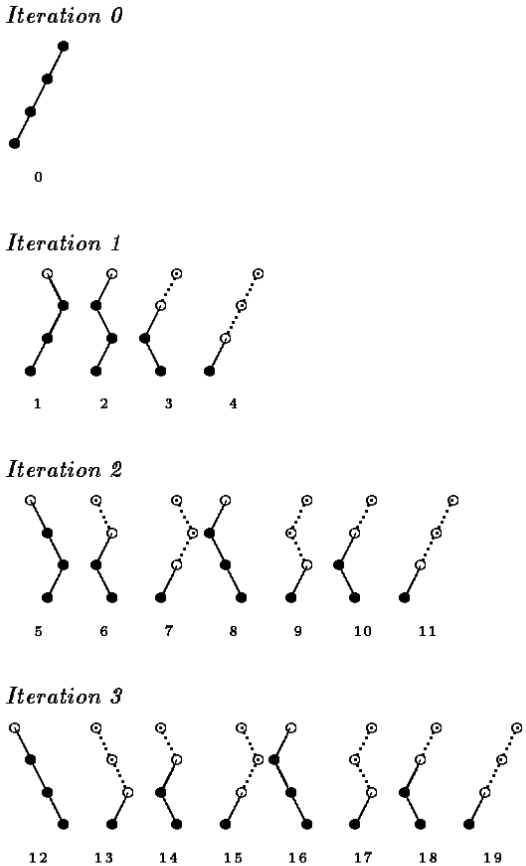


Figure 2: Execution trace of LDS.

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.

Is LDS Any Good?

problem	ildsn-e	ildsn-l	bt
dincbas88	15 (17)	13(14)	11 (11)
4-72	13660 (2659)	13188 (2417)	26522 (-)
16-81	21666 (193459)	**3415 (52519)**	- (-)
26-82	**29845 (30354)**	48974 (50986)	- (-)
41-66	**8530 (19222)**	17081 (38821)	109 (-)
60-01	**291552 (694)**	- (26270)	296391 (39138)
60-02	200 (200)	200 (200)	200 (200)
60-03	2096 (492)	**1393 (394)**	203 (218)
60-04	30403 (36755)	**15206 (5544)**	215 (313)
60-05	238 (270)	239 (209)	201 (201)
65-01	- (-)	- (-)	- (-)
65-02	700 (200)	**380** (200)	- (200)
65-03	**977 (362)**	1161 (390)	203 (346)
65-04	- (**2480**)	- (**14811**)	- (766)
65-05	**1137** (210)	9673 (209)	205 (201)
90-01	39727 (-)	**13497** (-)	- (-)
90-02	- (-)	- (-)	- (-)
90-03	6676 (-)	**2455** (-)	203 (-)
90-04	- (-)	- (-)	- (-)
90-05	30297 (-)	**14897** (-)	205 (231)

Table 1. The number of nodes explored to solve an instance of the car sequencing problem. First column is ILDSN taking discrepacies early, 2nd colum ILDSN with late dicrepancies, and 3d column for chronological backtracking. An entry of - corresponds to 500000 nodes explored and no solution found. Variables are statically ordered, by index. Static value ordering is H2 by default (and H1 in brackets). Best results are given in bold.

Depth-Bounded Discrepancy Search

- 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?).

Depth-bounded Discrepancy Search

Toby Walsh*
APES Group, Department of Computer Science
University of Strathclyde, Glasgow G1 1XL. Scotland
tw@cs.strath.ac.uk

Both LDS and ILDS treat all discrepancies alike, irrespective of their depth. However, heuristics tend to be less informed and make more mistakes at the top of the search tree. For instance, the Karmarkar-Karp heuristic for number partitioning [Korf, 1996] makes a fixed (and possibly incorrect) decision at the root of the tree. Near to the bottom of the tree, when there are 4 or less numbers left to partition, the heuristic always makes the optimal choice. Given a limited amount of search, it may pay to explore discrepancies at the top of the tree before those at the bottom. This is the motivation behind depth-bounded discrepancy search.

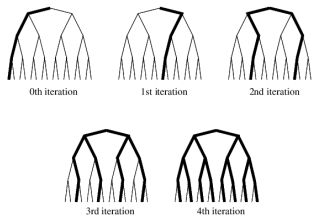


Figure 2: DDS on a binary tree of depth 4.

Depth-Bounded Discrepancy Search

Both LDS and ILDS treat all discrepancies alike, irrespective of their depth. However, heuristics tend to be less informed and make more mistakes at the top of the search tree. For instance, the Karmarkar-Karp heuristic for number partitioning [Korf, 1996] makes a fixed (and possibly incorrect) decision at the root of the tree. Near to the bottom of the tree, when there are 4 or less numbers left to partition, the heuristic always makes the optimal choice. Given a limited amount of search, it may pay to explore discrepancies at the top of the tree before those at the bottom. This is the motivation behind depth-bounded discrepancy search.

Depth-Bounded Discrepancy Search

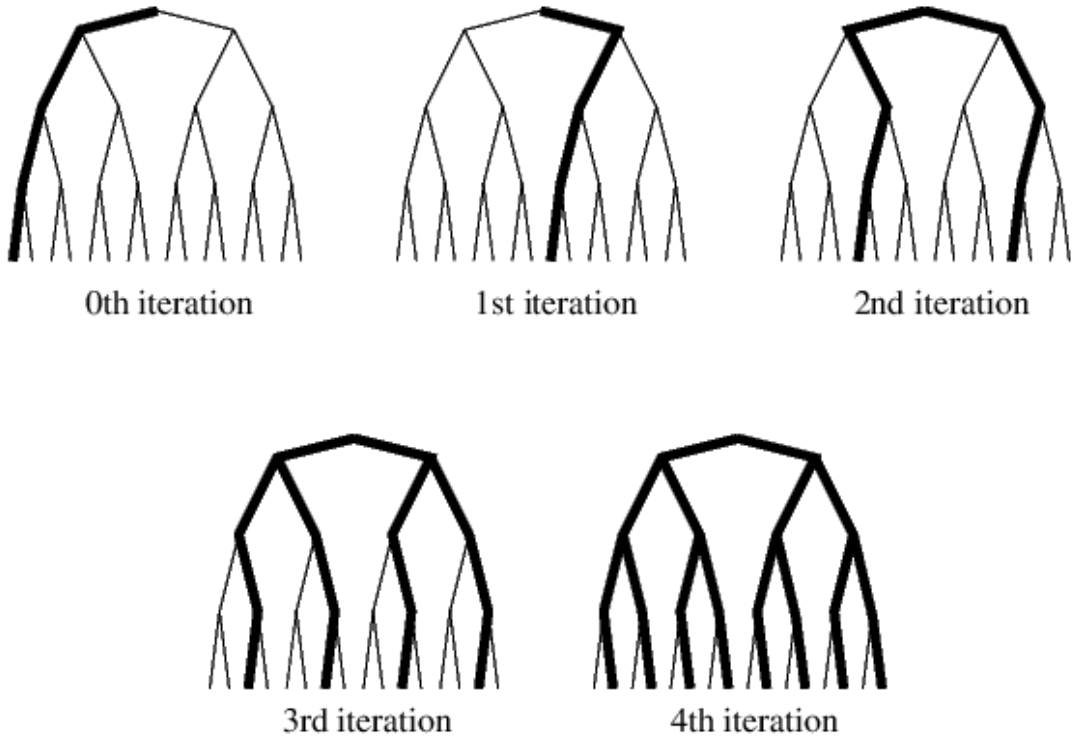


Figure 2: DDS on a binary tree of depth 4.

Is DDS Any Good?

Combined planning and scheduling in a divergent production system with co-production: A case study in the lumber industry

Jonathan Gaudreault [a,b,*], Jean-Marc Frayret [a,b], Alain Rousseau [a], Sophie D'Amours [a,b]

[a] FORAC Research Consortium, Pavillon Adrien-Pouliot, Université Laval, Québec (QC), Canada G1K 7P4
[b] CIRRELT, Centre Interuniversitaire de Recherche sur les Réseaux d'Entreprise, la Logistique et le Transport, Canada

This paper addresses a real industrial process planning and operations scheduling problem from the softwood lumber industry. More specifically, it deals with the planning and scheduling of drying and finishing operations. This production system is characterized as being (1) a divergent system with co-production (i.e., it produces different products at the same time from a single input product), (2) with alternative production processes.

In accordance with Section 1.1, the performances of both approaches (MIP and CP) were evaluated with regard to the solution quality according to computation time. The MIP model was solved using ILOG CPLEX 9.1 with an emphasis on the generation of feasible solutions. Computations were stopped after 11 h. The CP model was solved for the same computation time using the proposed search procedure (with DFS and DDS strategies). The first solution for DFS and DDS are the same (for trivial reasons). After that, DDS quickly improves solution quality. As for DFS, solution quality does not improve during the remainder of the 11 h. Therefore, DFS results were removed from the charts with no loss of information.

Is DDS Any Good?

This paper addresses a real industrial process planning and operations scheduling problem from the softwood lumber industry. More specifically, it deals with the planning and scheduling of drying and finishing operations. This production system is characterized as being (1) a divergent system with co-production (i.e., it produces different products at the same time from a single input product), (2) with alternative production processes.

Is DDS Any Good?

Is DDS Any Good?

In accordance with Section 1.1, the performances of both approaches (MIP and CP) were evaluated with regard to the solution quality according to computation time. The MIP model was solved using ILOG CPLEX 9.1 with an emphasis on the generation of feasible solutions. Computations were stopped after 11 h. The CP model was solved for the same computation time using the proposed search procedure (with DFS and DDS strategies). The first solution for DFS and DDS are the same (for trivial reasons). After that, DDS quickly improves solution quality. As for DFS, solution quality does not improve during the remainder of the 11 h. Therefore, DFS results were removed from the charts with no loss of information.

What About Unsatisfiable Instances?

What About Unsatisfiable Instances?

The reason we have focused on analyzing the early iterations of limited discrepancy search is that we believe in practice they are the only iterations that matter. Earlier, we argued on intuitive grounds that they would be more important than the later iterations. We will now take the position that in practice the later iterations don't matter at all. The reason is that if the objective is to maximize the probability of finding a solution in a given number of nodes, there are always better things to do than use those nodes on later iterations of limited discrepancy search.

Discrepancy search strategies like DDS, LDS and ILDS are designed for under-constrained and soluble problems, where we expect to explore just a small fraction of the search tree. Harvey and Ginsberg report that such problems are common in areas like planning and scheduling [Harvey and Ginsberg, 1995]. Discrepancy search strategies offer no advantage on insoluble problems where the tree must be traversed completely. Indeed, for a balanced binary tree, our asymptotic results, along with those of [Korf, 1996], show that we explore approximately double the number of nodes of DFS. When the tree is unbalanced, as a result of constraint propagation and pruning, the overhead can be even greater. We therefore restricted the experiments in this section to soluble and under-constrained problems, on which discrepancy search strategies can offer advantages.

on discrepancy-based search? We, therefore, compare ILDS-early against chronological backtracking (BT) over the same datasets (Figure 8). The left of Figure 8 shows that in the region where problems are mostly unsatisfiable ($25 \leq n \leq 35$) chronological BT is almost one order of magnitude better than ILDS, but when instances are satisfiable ($n \geq 40$), ILDS is the algorithm of choice, and this agrees remarkably well with the results in Korf [1996].

Better Stopping Conditions

THEOREM 4.1. *If a probe terminates without consuming its quota of discrepancies, the problem is unsatisfiable and search can terminate.*

YLDS performs identically to ILDS-early over the satisfiable instances $n \geq 40$. YLDS should be compared against ILDS-early. When all problems are unsatisfiable, YLDS shows a clear advantage over ILDS-early, being about 36% faster when $n = 25$. When instances are a mix of satisfiable and unsatisfiable, YLDS is about 31% faster at $n = 30$ and 25% faster when $n = 35$, where percentage satisfiable is 52%. The reason for these gains is due to the reduction in discrepancies required to prove unsatisfiability (shown in brackets): Typically, YLDS uses less than half of the discrepancies required by ILDS.

But wait, there's more!

- Interleaved Depth First Search
- Interleaved and Discrepancy Based Search
- Yet ImprovEd Limited Discrepancy Search
- Limited Discrepancy Beam Search
- Beam search Using Limited discrepancy Backtracking
- Probing and restarting
- ...

What About Optimisation Problems?

- Discrepancy-Bounded Depth First Search
- Climbing Depth-Bounded Adjacent Discrepancy Search
- Branch and Bound (in Algorithmics 4)
- Russian Doll Search (FATA, 4pm on 2nd Dec, SAWB423)
- Dichotomic Search
- ...

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

Writing Our Own Search in Old Choco

```
public boolean lds(DecisionVar[] v, int kMax,
        boolean lateDiscrepancies) throws ContradictionException
{
    boolean consistent = true;
    int n = v.length;
    inst = new int[n];
    nodes = 0;
    worldPush();
    try {
        super.propagate();
    }
    catch (ContradictionException e) {
        consistent = false;
    }
    if (consistent) {
        consistent = false;
        for (int k = 0 ; k <= kMax && !consistent ; k++) {
            worldPush();
            consistent = ldsProbe(v, n, k, 0, lateDiscrepancies);
            worldPop();
        }
    }
    worldPop();
    if (consistent)
        for (int i = 0 ; i < n ; i++) v[i].setVal(inst[i]);
    return consistent;
}
```

Writing Our Own Search in Old Choco

```
private boolean ldsProbe(DecisionVar[] v, int n, int k, int i,
        boolean lateDiscrepancies) throws ContradictionException
{
    if (i == n) return true;   // all vars instantiated
    DecVarEnumeration dve = new DecVarEnumeration(v[i]);
    boolean result = false;
    int m  = v[i].getDomainSize();
    int x  = dve.getNextValue();
    int pd = getPotentialDiscrepancies(v, n, i + 1);
    if (lateDiscrepancies && k <= pd && ! result) {
        worldPush();
        result = isConsistent(v, i, x) && ldsProbe(v, n, k, i + 1, lateDiscrepancies);
        worldPop();
    }
    for (int j = 1 ; j < m && j <= k && ! result ; j++) {
        int y = dve.getNextValue();
        worldPush();
        if (isConsistent(v, i, y))
            result = ldsProbe(v, n, k - j, i + 1, lateDiscrepancies);
        worldPop();
    }
    if (! lateDiscrepancies && k <= pd && ! result) {
        worldPush();
        result = isConsistent(v, i, x) && ldsProbe(v, n, k, i + 1, lateDiscrepancies);
        worldPop();
    }
    return result;
}
```

Writing Our Own Search in the Shiny New Choco 3

- Easy in Choco 3 due to more abstraction: it's an example in the manual. (But is this LDS, ILDS, or something else?)
- We still have to write code...

```
public class LimitedSearch extends AbstractIntBranching implements IntBranching {

    IntBranching delegate;
    IStateInt discrepancyCount;
    int maxDiscrepancy;

    public LimitedSearch(Problem pb, IntBranching delegate, int nbViolsMax) {
        this.delegate = delegate;
        discrepancyCount = new StoredInt(pb.getEnvironment(), 0);
        maxDiscrepancy = nbViolsMax;
    }

    public int getNextBranch(Object x, int i) {
        discrepancyCount.add(1);
        return delegate.getNextBranch(x, i);
    }

    public boolean finishedBranching(Object x, int i) {
        if (discrepancyCount.get() < maxDiscrepancy)
            return delegate.finishedBranching(x, i);
        return true;
    }

    public Object selectBranchingObject() {
        return delegate.selectBranchingObject();
    }

    public int getFirstBranch(Object x) {
        return delegate.getFirstBranch(x);
    }

    public void goDownBranch(Object x, int i) throws ContradictionException {
        delegate.goDownBranch(x, i);
    }

    public void goUpBranch(Object x, int i) throws ContradictionException {
        delegate.goUpBranch(x, i);
    }
}
```

Writing Our Own Search in the Shiny New Choco 3

```
public class LimitedSearch extends AbstractIntBranching implements IntBranching {

   IntBranching delegate;
   IStateInt discrepancyCount;
   int maxDiscrepancy;

   public LimitedSearch(Problem pb, IntBranching delegate, int nbViolsMax) {
      this.delegate = delegate;
      discrepancyCount = new StoredInt(pb.getEnvironment(), 0);
      maxDiscrepancy = nbViolsMax;
   }

   public int getNextBranch(Object x, int i) {
      discrepancyCount.add(1);
      return delegate.getNextBranch(x, i);
   }

   public boolean finishedBranching(Object x, int i) {
      if (discrepancyCount.get() < maxDiscrepancy)
        return delegate.finishedBranching(x, i);
      return true;
   }
```

Search Combinators?

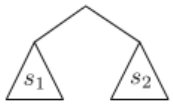
Constraints (2013) 18:269–305
DOI 10.1007/s10601-012-9137-8

Search combinators

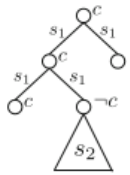
Tom Schrijvers · Guido Tack · Pieter Wuille · Horst Samulowitz · Peter J. Stuckey

1) and($[s_1, s_2]$)

2) or($[s_1, s_2]$)

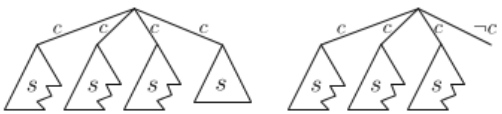


3) if(c, s_1, s_2)



4) portfolio($[s_1, s_2, s_3]$)

5) restart(c, s)



For the user we provide a modeling language for expressing complex search heuristics based on an (extensible) set of primitive combinators. Even if the users are only provided with a small set of combinators, they can already express a vast range of combinations. Moreover, using combinators to program application-tailored search is vastly more productive than resorting to a general-purpose language.

The general objective of search combinators is to reduce the effort of developing implementations of search heuristics. For that purpose, the search combinators approach applies well-known and widely researched tools from the fields of programming languages and software engineering: *modularity* and *reuse*. Modularity means that different aspects of a system can be developed (implemented, compiled, maintained) independently in software artifacts called modules or *components*. Components interact with one another through well-defined interfaces. If interfaces are sufficiently general and the means to compose components into systems are sufficiently flexible, the same component can be reused in different configurations to build different systems.

Limited discrepancy search [8] with an upper limit of l discrepancies for an underlying search s.

$$\mathsf{lds}(l, s) \equiv \mathsf{for}(n, 0, l, \mathsf{limit}(\mathsf{discrepancies} \leq n, s))$$

The for construct iterates the maximum number of discrepancies n from 0 to l, while limit executes s as long as the number of discrepancies is smaller than n. The search makes use of the discrepancies statistic that is maintained by the search infrastructure. The original LDS [8] visits the nodes in a specific order. The search described here visits the same nodes in the same order of discrepancies, but possibly in a different individual order—as this is determined by the global queuing strategy.

This work directly extends our earlier work on *Monadic Constraint Programming* (MCP) [21]. MCP introduces stackable search transformers, which are a simple form of search combinators, but only provide a much more limited and low level form of search control. In trying to overcome its limitations we arrived at search combinators.

Search Combinators?

For the user we provide a modeling language for expressing complex search heuristics based on an (extensible) set of primitive combinators. Even if the users are only provided with a small set of combinators, they can already express a vast range of combinations. Moreover, using combinators to program application-tailored search is vastly more productive than resorting to a general-purpose language.

Search Combinators?

The general objective of search combinators is to reduce the effort of developing implementations of search heuristics. For that purpose, the search combinators approach applies well-known and widely researched tools from the fields of programming languages and software engineering: *modularity* and *reuse*. Modularity means that different aspects of a system can be developed (implemented, compiled, maintained) independently in software artifacts called modules or *components*. Components interact with one another through well-defined interfaces. If interfaces are sufficiently general and the means to compose components into systems are sufficiently flexible, the same component can be reused in different configurations to build different systems.

Search Combinators?

Limited discrepancy search [8] with an upper limit of l discrepancies for an underlying search s.

$$\mathsf{lds}(l, s) \equiv \mathsf{for}(n, 0, l, \mathsf{limit}(\mathsf{discrepancies} \leq n, s))$$

The for construct iterates the maximum number of discrepancies n from 0 to l, while limit executes s as long as the number of discrepancies is smaller than n. The search makes use of the discrepancies statistic that is maintained by the search infrastructure. The original LDS [8] visits the nodes in a specific order. The search described here visits the same nodes in the same order of discrepancies, but possibly in a different individual order—as this is determined by the global queuing strategy.

Search Combinators?

This work directly extends our earlier work on *Monadic Constraint Programming* (MCP) [21]. MCP introduces stackable search transformers, which are a simple form of search combinators, but only provide a much more limited and low level form of search control. In trying to overcome its limitations we arrived at search combinators.

http://dcs.gla.ac.uk/~ciaran
c.mccreesh.1@research.gla.ac.uk