Conflict-directed A* and Its Role in Model-based Embedded Systems [1,2]

Brian C. Williams * and Robert J. Ragno

MIT Computer Science and Artificial Intelligence Laboratory
The Stata Center, 32 Vassar Street
Cambridge, Massachusetts 02139 USA

Abstract

Artificial Intelligence has traditionally used constraint satisfaction and logic to frame a wide range of problems, including planning, diagnosis, cognitive robotics and embedded systems control. However, many decision making problems are now being re-framed as optimization problems, involving a search over a discrete space for the best solution that satisfies a set of constraints. The best methods for finding optimal solutions, such as A*, explore the space of solutions one state at a time. This paper introduces *conflict-directed A**, a method for solving *optimal constraint satisfaction problems*. Conflict-directed A* searches the state space in best first order, but accelerates the search process by eliminating subspaces around each state that are inconsistent. This elimination process builds upon the concepts of conflict and kernel diagnosis used in model-based diagnosis[1,2] and in dependency-directed search[3–6]. Conflict-directed A* is a fundamental tool for building model-based embedded systems, and has been used to solve a range of problems, including fault isolation[1], diagnosis[7], mode estimation and repair[8], model-compilation[9] and model-based programming[10].

Key words: Constraint optimization with logical constraints, Propositional satisfiability, Conflict and clause learning, Model-based autonomous and embedded systems

* Corresponding author.
Email address: `williams@mit.edu` (Brian C. Williams).
URL: `mers.csail.mit.edu` (Brian C. Williams).

1 Preprint first submitted on January 9th, 2003.
2 Supported by the NASA Cross Enterprise Technology Development Program under contract NAG2-1466, and the DARPA MOBIES program under contract F33615-00-C-1702.

1 Introduction

The approach of focusing search based on summaries of logical inconsistency is a venerable problem solving method within AI. These summaries have gone under various names, such as *nogoods*[3], *conflicts* [11,12,1], *elimination sets*[6], or *exclusion relations*[13]; in this paper we use the term *conflict*. Past work has concentrated extensively on using conflicts to find a solution that is consistent with a set of constraints. Consistency, however, says nothing about the quality of the solution. Hence, AI is shifting increasingly towards problem formulations that involve finding a set of best solutions, given a *utility function* that measures the quality of the solution. The generalization of conflict-directed search to optimization is an open research frontier. In this paper we demonstrate how conflicts, when combined with A* search, provide a powerful method for finding optimal solutions to discrete constraint satisfaction problems. We call this method *conflict-directed A**.

AI has explored the paradigm of "search through diagnosis and repair", both as a fundamental problem solving paradigm[4,5,14,6], and as a strategy for solving most core AI reasoning tasks, such as planning, scheduling, diagnosis and qualitative reasoning[15,16,3,12,11,1,17–21]. In this paradigm the diagnosis of an incorrect solution is summarized by a conflict, which is then used to guide the repair step. Systematic, backtrack search methods use conflicts to select backtrack points. These methods include dependency-directed backtracking [3], intelligent backtracking, conflict-directed backjumping[22] and dynamic backtracking[6]. Local search methods use conflicts to select local moves that remove one or more conflicts. Representative examples include Hacker[15] for planning, Min-Conflict for constraint satisfaction [16], and GSAT or WalkSat for propositional satisfiability[14,23,24].

Conflict-directed A* builds upon a third approach, which uses conflicts to solve constraint satisfaction problems by divide and conquer. This approach plays an integral role in model-based diagnosis[19], and was first introduced within the General Diagnostic Engine (GDE) [1]. GDE frames diagnosis as a constraint satisfaction problem that involves finding assignments of modes to components that are consistent with a device model and a set of observations. GDE begins by searching in parallel for all conflicts, that is, "smallest" partial assignments that produce an inconsistency. The set of conflicts are then combined to produce compact descriptions of all feasible states, called *kernel diagnoses*. The key feature of a conflict-directed divide and conquer approach is its ability to reason intensionally about collections of states, rather than states individually. This reduces the effective size of the search space explored.

A significant limitation of this early approach is that many practical applications only require one or a few best solutions, rather than all solutions. In this case, the approach of generating all solutions and all conflicts in parallel can waste significant effort. This limitation is exacerbated by the fact that the set of abstract descriptions – conflicts and kernel diagnoses – grows exponential in the worst case. Hence, in the model-based diagnosis community, GDE's approach fell increasingly to the wayside during the 90's, being replaced by methods that focus on the small subset of the diagnoses that are likely, by enumerating the state space in best first order [7,25,8,26]. Research on these best first enumeration methods have grappled with three key questions:

- Can we use conflicts to effectively reason about classes of states when we are only interested in a few best solutions, not all solutions?
- Can theories of diagnosis based on conflicts and kernel diagnoses be rigorously unified with theories of diagnosis as best-first search?
- Can general purpose, conflict-directed methods for solving constraint satisfaction problems (CSPs) be unified with informed methods for best-first search?

We resolve these questions by addressing a family of problems called *Optimal Constraint Satisfaction Problems*. An optimal CSP is a multi-attribute decision problem whose decision variables are constrained by a set of finite domain constraints. We focus on the solution to optimal CSPs whose attributes are *preferentially independent*, a property shared by most practical multi-attribute decision problems. An Optimal CSP is differentiated from formalisms like Valued[27] and Semi-ring CSPs[28], in that optimal CSPs operate on hard constraints rather than soft constraints.

In this paper we introduce *conflict-directed A**, a method for solving Optimal CSPs that satisfy preferential independence. [3] Like A*[29], this approach tests a sequence of candidate solutions in decreasing order of utility. It differs from A* in that it uses the sources of conflict identified within each inconsistent candidate to jump over related candidates in the sequence. This approach is synergistic with constraint optimization research focussed on finding good heuristics, such as [30–32].

In practice, conflict-directed A* has lead to a dramatic decrease in the number of states visited over an A* approach that does not exploit conflicts. This has been demonstrated both on randomized problems and in real world application. Variants of this algorithm have been demonstrated on the control of a variety of embedded and autonomous systems, including the tasks of repairing a 100 million dollar deep space probe, 6 light minutes from earth [8,33], and monitoring the health of a robotic astronaut. Variants have also

[3] Conflict-directed A* is a generalization of the conflict-directed, best-first search algorithm introduced in [8], and is an evolution of conflict-directed algorithms introduced in [5,7,25].

been used to perform such tasks as model compilation [9], diagnosis[1], mode estimation[7,34,35], and hardware reconfiguration and repair[8–10].

This paper focuses on the pervasive family of discrete constraint optimization problems. In related research we demonstrate how conflicts extend to the solution of continuous and mixed discrete – continuous optimization problems and to branch and bound search. [36] describes a method, called *activity analysis*, that solves non-linear, constraint optimization problems by ruling out portions of the state space that are sub-optimal. In addition, [37] describes a method that solves mixed logic – linear programs within a branch and bound search framework, by learning conflicts from both infeasible and suboptimal relaxed solutions. Finally, [38] describes a method, called *decompositional model-based learning*, which uses conflicts to decompose and solve maximum likelihood problems, such as parameter estimation, state estimation and model-based learning.

The remainder of this paper is structured as follows. Section 6 introduces conflict-directed A* informally, both by stepping through its execution on simple examples, and by highlighting its role in creating model-based systems that reason at reactive time-scales. Section 2 defines optimal constraint satisfaction (OCSP) and introduces the property of mutual preferential independence. Section 3 provides an overview. It demonstrates how *conflict-directed A** uses conflicts to jump over leading states that are proven inconsistent, and it demonstrates how optimal CSPs are used at the core of embedded systems that are self-diagnosing and repairing.

The remaining sections develop the algorithms in detail. Section 4 develops an algorithm for solving Optimal CSPs, called *constraint-based A**, without using conflicts. Constraint-based A* leverages the property of preferential independence to focus search tree expansion on only the most promising children. Section 5 introduces the core algorithm underlying conflict-directed A*, called Next-Best-Kernel, which uses A* search to quickly find the region of state space, called a kernel, that contains the best utility state that resolves the known conflicts. Section 6 introduces the *conflict-directed A** algorithms for generating single and multiple solutions, by unifying the constraint-based A* and Next-Best-Kernels algorithms, introduced in the preceding two sections. Finally, in Section 7 we discuss experimental results that compare the performance of constraint-based A* and conflict-directed A*, applied to both randomly generated problems and a representative space application.

2 Optimal CSPs

To lay the ground work for our development of conflict-directed A*, we define *optimal constraint satisfaction problem* (OCSP) and introduce a pedagogical example. Recall that a constraint satisfaction problem (CSP) $\langle \mathbf{x}, \mathbf{D_x}, C_\mathbf{x} \rangle$ consists of a set of variables $x_i \in \mathbf{x}$ that range over finite domains $D_{x_i} \in \mathbf{D_x}$, and a set of constraints $C_\mathbf{x} : \mathbf{x} \to \{True, False\}$. A *solution* is any assignment to $\mathbf{x}$ that satisfies $C_\mathbf{x}$, that is, for which $C_\mathbf{x}[\mathbf{x}] = True$.

An *Optimal* CSP, $\langle \text{CSP}, \mathbf{y}, g, \rangle$, consists of a CSP $= \langle \mathbf{x}, \mathbf{D_x}, C_\mathbf{x} \rangle$, a set of decision variables $\mathbf{y} \subset \mathbf{x}$, and a cost function $g : \mathbf{y} \to \Re$.[4] We refer to the remaining variables $\mathbf{z} = \mathbf{x} - \mathbf{y}$, as *non-decision variables* and partition the domain $\mathbf{D_x}$ into $\mathbf{D_y}$ and $\mathbf{D_z}$. We call the elements of $\mathbf{D_y}$ decision states. A solution $\mathbf{y}^*$ to an OCSP is a minimum cost decision state that is consistent with the CSP. More precisely, let constraint $C_\mathbf{y}$ be the projection of $C_\mathbf{x}$ on to decision variables $\mathbf{y}$, where $C_\mathbf{y}(\mathbf{y})$ is *consistent* for $\mathbf{y} \in \mathbf{D_y}$ if and only if $\exists \mathbf{z} \in \mathbf{D_z}.C_\mathbf{x}(\mathbf{y}; \mathbf{z})$ is consistent. Then

$$\mathbf{y}^* = argmax_{\mathbf{v} \in \mathbf{D_y}} g(\mathbf{v}) \text{ such that } C_\mathbf{y}(\mathbf{v}) \text{ is consistent.}$$

A natural way of encoding g for an OCSP is through a *multi-attribute* cost function, which associates *attribute costs* $g_i(v_{ij})$ to individual variable assignments $x_i = v_{ij}$, and uses G to compose them into a global cost.[5] Most practical multi-attribute decision problems satisfy a property called *mutually preferential independent (MPI)*. This means that for any subset of the problem's decision variables $\mathbf{w} \subset \mathbf{y}$, our preference between assignments to $\mathbf{w}$ are independent of the particular assignments to the remaining decision variables $\mathbf{y} - \mathbf{w}$.[6] The key consequence of MPI, exploited by algorithms in this paper,

[4] We frame an OCSP as one of *minimizing cost*, to be consistent with the framework of A* search; however, it is equally valid to think in terms of maximizing utility, particularly for multi-attribute utility decision problems.

[5] More precisely, the cost function g of an OCSP is specified as $g = \langle \mathbf{g}, G, I_G \rangle$, where each attribute cost $g_i \in \mathbf{g}$ maps D_{y_i} to $\Re$, G is a binary function from $\Re \times \Re$ to $\Re$ that is associative and commutative, and I_G is the identity of G. G applied to n attribute costs is defined recursively in the standard manner,

$$\begin{aligned} G(u_1, u_2, \ldots u_n) &= G(u_n, G(u_1, u_2, \ldots u_{n-1})), \\ G(u_1) &= G(u_1, I_G), \text{ and} \\ g(\mathbf{y}) &= G(g_1(y_1), g_2(y_2), \ldots g_n(y_n)). \end{aligned}$$

[6] Preference is defined as better cost:
Definition 1 Let $\langle \mathbf{y}, g, \text{CSP} \rangle$ be an optimal CSP, and δ_1 and δ_2 be two sets of assignments to $\mathbf{y}$. Then δ_1 is *preferred* over δ_2 if g$(\delta_1) < g(\delta_2)$.

is that an assignment to $\mathbf{y}$ minimizes cost *by minimizing the attribute cost g_i of each y_i separately.*

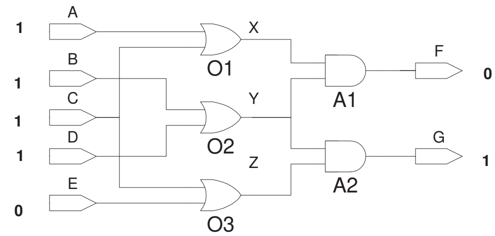

Fig. 1. Boolean polycell, with observed values indicated.

A simple example of an OCSP is the task of identifying the most likely, consistent diagnoses of a circuit, called Boolean polycell, consisting of three OR gates and two AND gates (Figure 1). The inputs and outputs are observed as indicated in the figure. Each component is in one of two possible *modes,* good (G) or broken (U). A good component correctly performs its boolean function. The behavior of a broken component is "unknown," it imposes no constraint [12,1].

The decision variables are component mode variables, $\mathbf{y}$, each of whose domain consists of $\{G, \mathbf{U}\}$. A *candidate* is a mode assignment to $\mathbf{y}$. A *diagnosis* is a candidate that is consistent with a set of constraints $C_{\mathbf{y}}$ that model Boolean polycell and the set of observations. For example, the model includes the constraint "If $O1 = G$ Then ($X = 1$ IFF ($B = 1$ Or $C = 1$)." Utility is the candidate probability $P(\mathbf{y})$. We take cost $g(\mathbf{y})$ to be $1/P(\mathbf{y})$. To keep our example simple, we use the candidate's prior probability, and assume that component failures are independent:

$$g(\mathbf{y}) = -\prod_i P_i(y_i),$$

The attribute utilities are the component probabilities, and are combined using multiplication, which satisfies MPI. Assuming that OR gates fail with probability 1% and AND gates with probability .5%, then the solution to the OCSP is that O1 is broken, that is $\{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\}$.

3 Conflict-directed A* in a Nutshell

This section provides a thumbnail sketch of conflict-directed A*, starting with a pictorial view, and then elaborating the algorithm, using Boolean polycell as the driving example.

3.1 A Pictorial View of Conflict-directed A*

A* is often the method of choice for finding optimal solutions to discrete state space search problems[29,39]. A* generates and tests states in increasing order of *heuristic cost*, as depicted in Figure 2. Note that this can also be equivalently formulated as search in order of decreasing utility. A* can be equivalently formulated in terms of maximizing utility. We use both terms, "cost" and "utility," in this paper, depending on what offers the most intuitive explanation for the given topic.

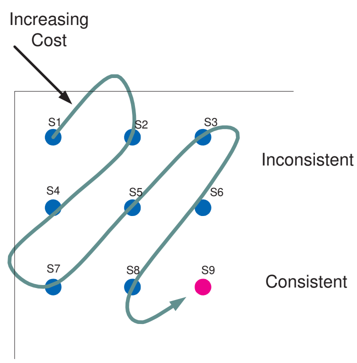

Fig. 2. A* examines all best cost states up to the best consistent state.

If a heuristic is *admissible*, that is, it never overestimates cost, then A* is guaranteed to return an optimal feasible solution if one exists. A* is *efficient* in that it explores no search state with estimated cost greater than the optimum. However, to guarantee that the solution it returns is optimal, A* visits *every* state whose estimated cost is less than the true optimum. This is impractical for many real world applications, such as model-based systems that perform best-first search within the reactive control loop [8,9,34,35,10].

Conflict-directed A* accelerates this search process by leaping over many of these leading inconsistent states. Conflict-directed A* guides its search using *conflicts*, which are descriptions of states that are *inconsistent* with the CSP. Intuitively, a conflict denotes a set of states, all of which are proven inconsistent using the same proof. For example, we might deduce from a model that any state that has a shorted voltage regulator will produce the same symptom, such as a particular voltage being too low. We say that a state contained by a conflict *manifests the conflict*, and a state not contained by a conflict *resolves the conflict*.

In Figure 3, conflict-directed A* first selects state $S1$, which proves inconsistent. This inconsistency generalizes to Conflict 1, which eliminates states $S1$ - $S3$ (Figure 3, upper left). Conflict-directed A* then tests state $S4$ as the best cost state that resolves Conflict 1. $S4$ tests inconsistent and generates Conflict 2, eliminating states $S4$ - $S7$ (Figure 3, upper right). Next, conflict-directed A*

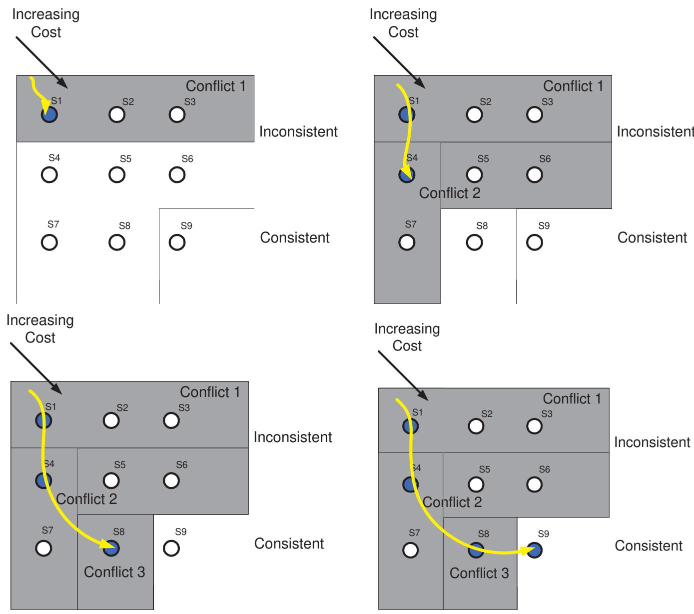

Fig. 3. Conflict-directed A* focuses search using discovered conflicts. Upper left — lower right represent successive snapshots along a prototypical search. Circles represent states. Filled in circles have been tested for consistency. Regions in grey have been ruled out by conflicts. Only state $S9$ is consistent.

tests state $S8$, which is the best cost state that resolves both Conflicts 1 and 2. $S8$ proves inconsistent as well, producing Conflict 3 (Figure 3, lower left). Finally, the search tests state $S9$ as consistent and returns it as an optimal solution (Figure 3, lower left).

In this example conflict-directed A* tested three inconsistent states, while jumping over five inconsistent states. In real-world examples the savings is more dramatic. For example, consider the problem of reconfiguring the main engine system of the Cassini Saturn space probe, which was performed in simulation by the Livingstone system [40]. The reconfiguration task consists of finding a minimum-cost set of component modes, such as closing valves and turning on drivers, that can be shown to thrust the engine system while maintaining a set of safety constraints. The state space consists of roughly 4^{80} states. Using conflict-directed A*, less than a dozen candidate states are tested in order to find an optimal solution (Section 7), in contrast to thousands visited when conflicts are not employed.

3.2 Conflict-directed A as Generate and Test*

Conflict-directed A* performs an interleaved best-first generate and test (Figure 4). It generates as a candidate, the best valued decision state that resolves all discovered conflicts. It tests each candidate S for consistency against the

function Conflict-directed-A*(*OCSP*)
  returns the leading minimal cost solutions.
  Conflicts[*OCSP*] $\leftarrow$ {}
  OCSP $\leftarrow$ Initialize-Best-Kernels(*OCSP*)
  Solutions[*OCSP*] $\leftarrow$ {}
  loop do
    decision-state $\leftarrow$ Next-Best-State-Resolving-Conflicts(*OCSP*)
    if no decision-state returned **or**
        Terminate?(*OCSP*)
      then return Solutions[*OCSP*]
    if Consistent?(CSP[*OCSP*], *decision-state*)
      then add *decision-state* to Solutions[*OCSP*]
    new-conflicts $\leftarrow$ Extract-Conflicts(CSP[*OCSP*], *decision-state*)
    Conflicts[*OCSP*] $\leftarrow$
      Eliminate-Redundant-Conflicts(Conflicts[*OCSP*] $\cup$ *new-conflicts*)
    Update-Known-Kernels-Based-On-New-Conflicts(*OCSP*)
  end

Fig. 4. Top-level loop of Conflict-directed A*.

CSP using function Consistent?. When S tests inconsistent, Extract-Conflicts generalizes the inconsistency to one or more conflicts, denoting states that are inconsistent in a manner similar to S. The candidate is tested using any suitable CSP algorithm that extracts conflicts. Conflict-directed A* uses discovered conflicts as a record of known inconsistent states, while pruning redundancy using Eliminate-Redundant-Conflicts. When a new conflict is discovered, it also updates the search queue of candidates to be explored using Update-Known-Kernels-Based-On-New-Conflicts. Conflict-directed A* then uses Next-Best-State-Resolving-Conflicts to jump down to the next best candidate S' that *resolves all conflicts discovered thus far.* This process repeats until the desired leading solutions are found or all states are eliminated. [7]

The candidate can be tested using any suitable CSP algorithm that extracts conflicts. The minimal committment to the form of the CSP solved, and the CSP algorithm applied, makes it easy to augment a range of CSP solvers to solve Optimal CSP problems. Appendix A discusses requirements and general implementation issues for the four subprocedures used by Conflict-directed-A*. Our presentation focuses primarily on the most subtle procedure, Next-Best-State-Resolving-Conflicts.

We next walk through the execution of the top-level loop of conflict-directed A* for Boolean polycell, demonstrating how it *jumps over most leading candidates that are inconsistent,* while guaranteeing optimality.

[7] Our implementation includes termination conditions such as finding n leading solutions, finding all solutions within an order of magnitude cost of the leading solution, or terminating after m states are tested.

3.2.0.1 First Candidate – All Components Okay: Initially, conflict-directed A* has no known conflicts, hence all states are under consideration. It generates

Candidate 1: $\{O1 = G, O2 = G, O3 = G, A1 = G, A2 = G\}$,

which specifies that all components are working correctly, with probability .961. Candidate 1 is tested for consistency against the model and observations (Figure 5, left), using the DPLL propositional satisfiability algorithm[41], modified to return conflicts. In particular, given that $O1$ and $O2$ are good, DPLL concludes from inputs $A - D$ that X and Y are 1. Next, $A1$ is Good, $X = 1$ and $Y = 1$ are used to conclude that output F is 1. This prediction is inconsistent with observation $F = 0$, hence Candidate 1 is eliminated as a solution. This inconsistency is generalized to

Conflict 1: $\{O1 = G, O2 = G, A1 = G\}$,

which is a subset of Candidate 1's assignments that is sufficient to produce an inconsistency with the constraints. Conflict 1 is extracted using *reductio ad absurdum*, that is, the conjunction $O1 = G$, $O2 = G$ and $A1 = G$, imply $F = 1$, which conflicts with $F = 0$, hence the conjunction is inconsistent.

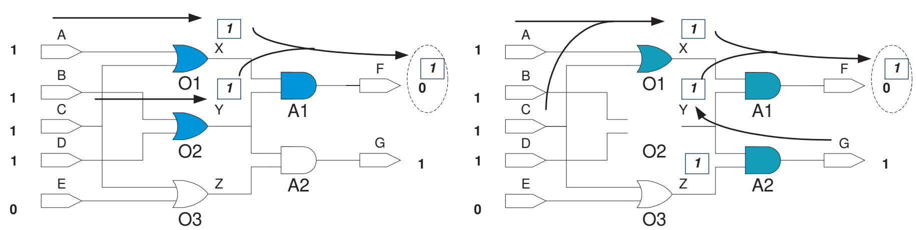

Fig. 5. Tested candidates, with conflicting components highlighted. Left) Candidate 1: All components okay. Right) Candidate 2: $O2$ is unknown.

3.2.0.2 Jump to Second Candidate – OR Gate O2 broken: In the second iteration, conflict-directed A* jumps over any leading candidates containing Conflict 1 as a subset, down to the best candidate resolving Conflict 1. A candidate *resolves a conflict* if it does not contain the conflict as a subset. A *conflict is resolved* by changing one of the assignments in the conflict to a different value, and by including this change in the new candidate. Hence, conflict-directed A* jumps over state $\{O1 = G, O2 = G, O3 = \mathbf{U}, A1 = G, A2 = G\}$, which contains Conflict 1 as a subset. It generates the next best state,

Candidate 2: $\{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = G\}$

with probability .0097. Candidate 2 resolves Conflict 1 by changing $O2 = G$ to $O2 = \mathbf{U}$.

Candidate 2 tests inconsistent, producing

Conflict 2: $\{O1 = G, A1 = G, O2 = G\}$.

This is shown on the right of Figure 5, with $O1 = \mathbf{U}$ depicted by removing component O1.

3.2.0.3 The Third Candidate is a Diagnosis – OR Gate O1 broken:

The next consecutive state is

Candidate 3: $\{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\}$,

with probability .0097, It resolves both Conflicts 1 and 2, by changing assignment $O1 = G$ to $O1 = \mathbf{U}$. Candidate 3 tests consistent (Figure 6, top), and hence, is our best diagnosis.

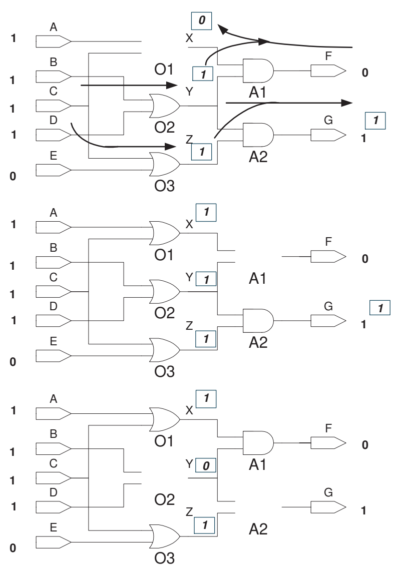

Fig. 6. Candidates 3-5 are diagnoses. Top) Candidate 3: $O1$ is unknown. Middle) Candidate 4: $A1$ is unknown. Bottom) Candidate 5: $O2$ and $A2$ are unknown.

3.2.0.4 Finding the Remaining Diagnoses Involves No Search:

Up until this point, conflict-directed A* has tested the consistency of three candidates, one of which is a diagnosis, and has jumped over one candidate. This is a modest savings over traditional A*. However, the initial phase of the search is typically invested in discovering conflicts, while the reward is reaped during the rest of the search. In this example, after testing the first two candidates, conflict-directed A* has discovered all conflicts for this example. Hence, at this point conflict-directed A* has sufficient knowledge to generate all remaining diagnoses *without generating any additional inconsistent candidates.*

Continuing the search, the three leading diagnoses are generated by jumping over 19 inconsistent candidates, and by explicitly considering only two inconsistent candidates (Figure 6). Measuring *search efficiency* as "Solutions Found / Candidates Tested," then traditional A* has efficiency $\frac{3}{21} = 14\%$, while conflict-directed A* has efficiency $\frac{3}{5} = 60\%$.

3.3 Generating the Best Kernel

The key to conflict-directed A* is the ability to efficiently generate, at each iteration, the next best candidate resolving all known conflicts. This is accomplished by mapping known conflicts to partial assignments called *kernels.* The best cost state is then extracted from these kernels. Each kernel describes a set of states that resolve the known conflicts. [8] For example, Conflicts 1 and 2 are both resolved by changing $O1 = G$ to $O1 = \mathbf{U}$, hence $\{O1 = \mathbf{U}\}$ is a kernel. We provide the intuitions behind this process in this section, presenting the details in Sections 4, 5 and 6.

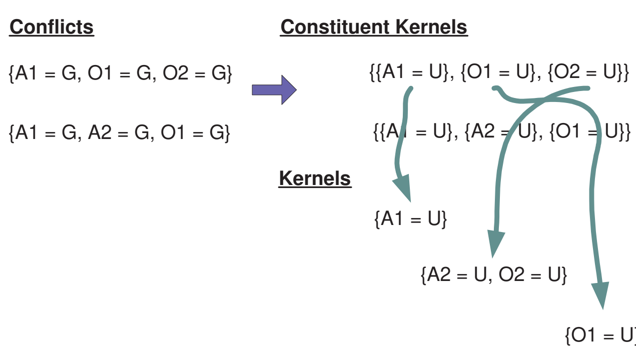

Fig. 7. Conflict-directed A* maps each conflict to a set of constituent kernels, which resolve that conflict alone. Kernels are generated by combining the constituents using minimal set covering.

Our mapping from conflicts to kernels is closely related to the candidate generation algorithm introduced within the General Diagnostic Engine (GDE) [1]. The first step generates *constituent kernels*, which resolve each conflict alone. The second step generates kernels that resolve *all* conflicts, by computing the minimal set covering of the constituent kernels. In particular, each combined kernel has the property that it contains a constituent kernel for every conflict, hence all conflicts are resolved. The constituent kernels for each of Conflict 1

[8] The concept of kernel generalizes from kernel diagnosis[2].

and 2 are shown at the top of Figure 7, and the three kernels covering the conflicts are shown at the bottom.

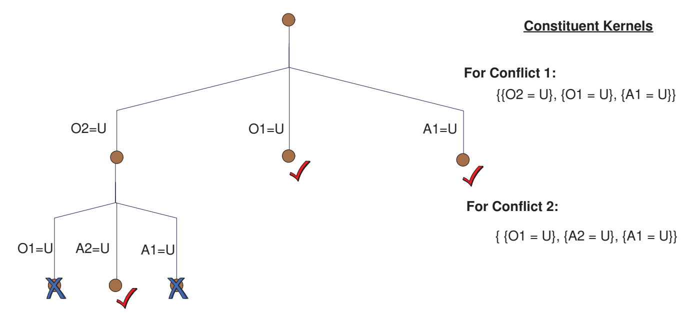

Fig. 8. The search tree created by Conflict-directed A* to identify all kernels. Visited nodes that are kernels are check marked, while those that are not are crossed off.

Unlike GDE, we only want to generate the kernel containing the best utility state. This is key, since the number of conflicts is worst case exponential in the number of decision variables. Our first idea is to view minimal set covering as a search, and to use A* search to find the kernel containing the best utility state, while explicitly enumerating as few kernels as possible. The search tree for Boolean polycell is shown in Figure 8. Its leaves are kernels and its intermediate nodes are partial coverings. For example, the bottom left leaf denotes kernel $\{O1 = \mathbf{U}, O2 = \mathbf{U}\}$ and its parent denotes $\{O2 = \mathbf{U}\}$. A tree node is expanded by selecting the constituent kernels of a conflict that is unresolved by that node, and creating a child for each constituent kernel of that conflict. For example, the root node does not resolve Conflict 1 or 2. Selecting Conflict 1, the children of the root are $\{O2 = \mathbf{U}\}$, $\{O1 = \mathbf{U}\}$ and $\{A1 = \mathbf{U}\}$. Nodes are eliminated when non-minimal, such as the first and third leaves at the bottom left of the tree.

Next, consider how the best candidate is extracted from a kernel. We generate the best candidate by assigning the remaining unassigned variables. To accomplish this we exploit a property called *mutual, preferential independence (MPI).* MPI says that to find the best candidate we assign each variable its best utility value, independent of the values assigned to the other variables. For example, initially there are no conflicts and the best kernel is the root node $\{\}$. For this kernel, Candidate 1 assigns the most likely value, G, to every variable, hence all components are working.

Continuing the process, when Candidate 2 is generated (Figure 9, left), only Conflict 1 has been discovered, hence the kernels correspond to the constituent kernels of Conflict 1. Kernel $\{O2 = \mathbf{U}\}$ contains the most likely candidate. Its estimated probability combines the probability of $\{O2 = \mathbf{U}\}$, .01, with an optimistic estimate (i.e., admissible heuristic) of the best probability of the unassigned variables. By MPI, this heuristic selects the best utility value for each unassigned variable, .97, resulting in .0097, for the best candidate of

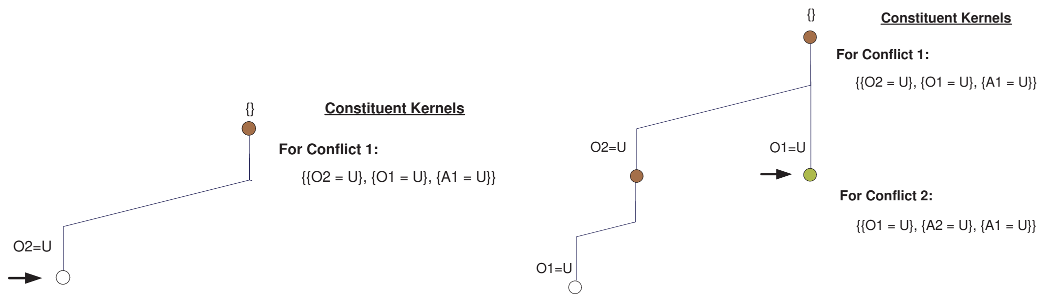

Fig. 9. Left) Tree expansion for kernel $\{O1 = \mathbf{U}\}$, producing Candidate 2. Only the best valued child of the root is expanded, not all children. Right) Tree expansion for kernel $\{O1 = \mathbf{U}\}$, producing Candidate 3. When node $O2 = \mathbf{U}$ is expanded, its best child and its next best sibling are created.

$\{O2 = \mathbf{U}\}$.

A key property of the search is that it only expands the best valued child of $\{\}$, which is $\{O2 = \mathbf{U}\}$, rather than all children. This is valid because MPI guarantees that $\{O2 = \mathbf{U}\}$ contains a state whose utility is at least as good as that of every state contained by the other children, such as $\{O1 = \mathbf{U}\}$. The best kernel must be $\{O2 = \mathbf{U}\}$, or one of its descendants. $\{O2 = \mathbf{U}\}$ resolves the known conflicts, and hence is a kernel. To maximize utility, the kernel's best candidate assigns G to the remaining components, that is, Candidate 2 has only $O2$ is broken.

When Candidate 3 is generated (Figure 9, right), Conflict 1 and 2 have been discovered. Node $\{O2 = \mathbf{U}\}$ does not resolve Conflict 2, and is expanded by creating its best child $\{O2 = \mathbf{U}, O1 = \mathbf{U}\}$. This is a kernel, whose best candidate has probability $.01 \times .098 = .00098$.

At this point it is no longer valid to just expand the best child of $\{O2 = \mathbf{U}\}$. Conflict 2 pruned out one or more of the states below node $\{O2 = \mathbf{U}\}$, hence we are no longer guaranteed that $\{O2 = \mathbf{U}\}$ contains a state that is as good as its sibling – this sibling may now contain the next best kernel. To achieve completeness we also expand its next best sibling, which is $\{O1 = \mathbf{U}\}$, with probability .0097. The next best sibling has higher probability than the best child, and hence the sibling is selected next. It is a kernel, and produces candidate 3,which is our most likely diagnosis.

An important property of the search strategy is the distinctive pattern of expanding a node at every step by creating its best child and its next best sibling. This strategy has the effect of growing the search queue to the modest size of at most $2N$ after visiting N nodes.

Often we will want to continue the search, for example, to find the set of most likely diagnoses that cover most of the probability density space. To accomplish this we need the capability to systematically explore the states within the kernels in best first order. This is more complicated than extracting the best state of the best kernel, as demonstrated above. We develop this complete strategy in Section 6.

3.4 Self-Repairing Systems That Reason Reactively

Conflict-directed A* is at the core of our approach to creating a new generation of model-based autonomous and embedded systems that achieve robustness by reasoning extensively at reactive time scales. In this section we outline the link between conflict-directed A* and model-based embedded systems.

Embedded systems, such as automobiles, power networks and building control systems, have dramatically increased their use of computation to achieve unprecedented levels of robustness, with little human support. These systems must operate robustly for years with minimum attention. An extreme example of this class of embedded systems is a fleet of intelligent space probes, which autonomously explores the nooks and crannies of the solar system. These embedded systems may need to radically reconfigure themselves in response to failures, and then navigate around these failures during their remaining operation.

The space of potential failures that an embedded system may be faced with over its lifespan is far too large for a programmer to successfuly pre-enumerate. Current hand coded systems achieve tractability by severly limiting the number of faults covered. In addition, the injection of undetected software bugs has caused significant system loss, such as the failure of the Mars Polar Lander. Instead, an embedded system should be able to automatically diagnose and plan courses of action for itself.

Our solution is a paradigm, called *model-based programming*, in which everyday embedded systems and explorers are programmed by specifying strategic guidance in the form of a few high-level control behaviors, called *model-based programs*[42]. These behaviors specify the system's intended state evolution, while abstracting away the detailed problem of controlling, estimating, diagnosing or repairing these states. These specifications look like traditional embedded programs, except that, while traditional programs read sensed variables and write control variables, model-based programs are allowed to read and write *hidden variables*.

A model-based program is executed by automatically generating a control sequence that moves the physical plant to the states specified by the control program (Figure 10). We call these specified states *configuration goals*. Program execution is performed using a *model-based executive*, consisting of a *control sequencer* and a *deductive controller*. The control sequencer repeat-

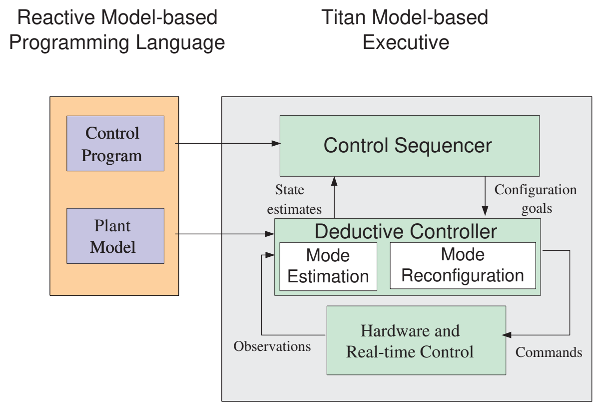

Fig. 10. Architecture for a model-based executive.

edly generates the next configuration goal, based on the control program and plant state. The deductive controller then generates a sequence of control actions that achieve this goal, based on knowledge of the current plant state and model. The deductive controller is responsible for estimating the plant's most likely current state, based on observations from the plant (*mode estimation*), and for issuing commands to move the plant through a sequence of states that achieve the goals (*mode reconfiguration*).

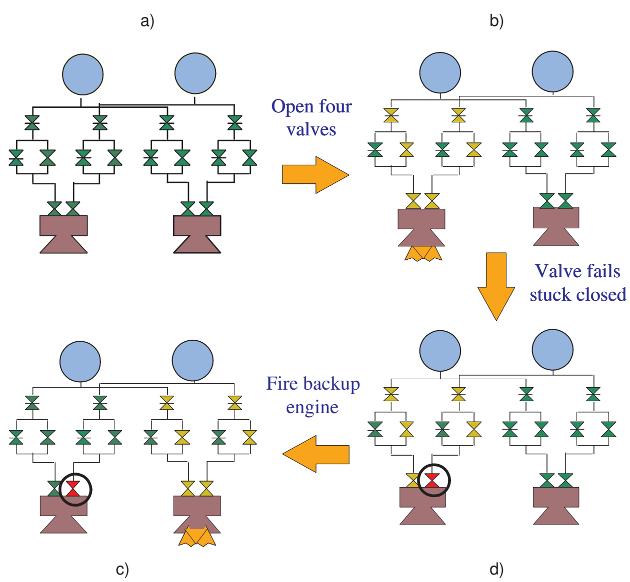

Fig. 11. Diagnosis and repair sequence for a simplified Cassini spacecraft. Pyro valves have horizontal bars through them. A valve is closed if filled in, otherwise, it is open. The faulty valve is circled.

For example, consider the problem of controlling the Cassini spacecraft as it inserts itself into Saturn's orbit. One configuration goal generated during this manuever is to achieve the state of engine thrusting. A series of idealized schematics of the main engine subsystem of Cassini are shown in Figure 11. It consists of two propellant tanks, two main engines (A on the left and B on the right), redundant latch valves and pyro valves. When propellant paths to a main engine are open, the propellants flow into the engine and produce thrust. The pyro valves are used to isolate parts of the engine. They can open or close only once, and are more costly to use than the latch valves. The system offers a wide range of configurations for achieving the goal of producing thrust.

Given the configuration goal of engine thrust, first, mode estimation determines that both engines are currently shut down (Figure 11, upper left). Mode reconfiguration then deduces that the goal may be accomplished by opening the latch valves leading to engine A (Figure 11 upper right), and sends out commands to open the valves. Suppose now that engine A fails to provide the desired thrust. Mode estimation identifies the likely cause of failure, for example, that the right latch valve going into engine A is stuck closed (Figure 11, lower right). Mode reconfiguration then searches for an alternate set of component modes that achieve the goal of engine thrust. Engine A cannot be used because of the stuck valve. Hence, mode reconfiguration deduces that the least costly way to achieve this goal is to fire the two pyro valves leading to Engine B, and to open the remaining latch valves (rather than firing additional pyro valves) (Figure 11, lower left).

Conflict-directed A* forms the core of both mode estimation and mode reconfiguration. We refer to its implementation as OpSat. The model-based executive compiles all hardware models into clauses in a propositional state logic. Mode estimation and mode reconfiguration are then framed as optimal CSPs of the form

$$\begin{aligned} & \arg\min f(\mathbf{x}) \\ & \text{s. t. } C_S(\mathbf{x}) \text{ is satisfiable,} \\ & \phantom{\text{s. t. }} C_U(\mathbf{x}) \text{ is unsatisfiable,} \end{aligned}$$

where C_S is a conjunction of propositional clauses that *must be satisfied* by the solution $\mathbf{x}$, and C_U is a conjunction of propositional clauses that *must not be satisfied* by $\mathbf{x}$.

Mode estimation selects, at each time step, most likely sets of component mode transitions that are consistent with the plant model and current observations. As discussed in [8], ME is framed roughly as

$$\begin{aligned} & \arg\min P_t(m'), \\ & \text{s. t. } M(m') \wedge O(m') \text{ is satisfiable,} \end{aligned}$$

where m' is a set of component modes that the system can transition to, P_t is a transition probability, M is the plant model and O is the current set of observations.

At each time step, mode reconfiguration first chooses a least cost set of reachable component modes that is consistent with the model and that entails the current configuration goals, as discussed in [9]. Mode reconfiguration is framed roughly as

$$\begin{aligned} &\arg\max R_t(m') \\ &\text{s. t. } M(m') \text{ is satisfiable,} \\ &\qquad M(m') \text{ entails } G(m'), \end{aligned}$$

where R_t is the cost of transitioning to mode m', G is a conjunction of configuration goals, and the constraint "$M(m')$ entails $G(m')$" is equivalent to $M(m') \wedge \neg G(m')$ being unsatisfiable.

Having identified a reachable set of component modes, mode reconfiguration then generates a command sequence to move to those modes. To accomplish this, mode reconfiguration generates a compact encoding of a universal plan at compile time. The first step of this process involves compiling the model into a set of automata that eliminate any reference to intermediate variables. As discussed in [9], OpSat is used to compile the model, by generating the complete set of prime implicates of the model that only refer to control assignments, current and next mode assignments.

To summarize, conflict-directed A* plays a central role in creating robust, model-based embedded systems, both during runtime, through state estimation and control, and at design time, through model compilation.

3.5 Summary

Thus far we have introduced conflict-directed A*, which uses discovered conflicts to jump over sets of inconsistent states, and we have demonstrated this process on Boolean polycell. In addition, we demonstrated the process of generating the best kernel, a consistent subspace containing the best cost solution, through A* search of a minimal covering tree. Finally we demonstrated how conflict-directed A* is at the core of building self-reparing systems, that reason at reactive time scales.

The remainder of this paper presents Optimal CSPs and conflict-directed A* more formally, and two supporting methods, constraint-based A* and next-best-kernels. Constraint-based A* offers a point of comparison, as a method for solving optimal CSPs that exploits preferential independence but not conflicts. Next-Best-Kernel offers a method for generating parsimonious descriptions of the "best" solutions, while offering an any-time approach to avoiding an exponential growth in the descriptions.

4 Constraint-based A*

In this section we generalize A* search to a method for efficiently solving Optimal CSPs, by exploiting the added structure imposed by the CSP and its cost function. We begin with a quick review of state space search and A*.

*4.1 Review of A**

Recall that a generic state space search problem is comprised of a set of states Σ, an initial state $\Theta \in \Sigma$, a set of search operators, $op : \Sigma \rightarrow \Sigma$, which map states to next states, and a Goal-Test: $\Sigma \rightarrow \{\text{True}, \text{False}\}$ which specifies whether or not a state satisfies the problem goals. A solution is an operator sequence that maps the initial state to a goal state. A problem also includes a cost function g, which returns the cost of applying an operator sequence, starting at initial state Θ. An optimal solution is one that minimizes cost.

A search tree is induced by rooting the tree at the initial state, and by recursively expanding each tree node, by mapping each node's state to child states, using the applicable operators. A* search explores the tree by expanding tree nodes in best first order according to a function $f(n)$, which estimates the cost of the best solution that goes through node n. A* is guaranteed to find the shortest path to a node first, and avoids expanding sub-optimal paths by exploiting an instance of the dynamic programming principle. A* terminates when it reaches a state that satisfies the goal test. Given a node n with state s, A* computes f by adding to g an estimate h of the minimum cost to reach a goal state from s.

A* is guaranteed to return the best solution, when h is *admissible*, that is, it never overestimates the minimum cost to reach a goal. A* is also characterized as *optimally efficient*[39], in the sense that no other optimal algorithm that expands search paths from the root is guaranteed to expand fewer nodes than A*. Intuitively, any algorithm that *does not* expand all nodes in the contours between the root and the goal contour runs the risk of missing the optimal solution.

Our leverage point for improving upon A* is the fact that an optimal CSP imposes additional structure that traditional A* does not exploit, in particular, states are factored into variable assignments, and constraint $C_{\mathbf{x}}$ is factored into a set of constraints. Our generalizations of A*, should preserve efficiency, that is, it should not explicitly consider any state whose g is worse than the optimal solution. For correctness it must also rule out any state whose g is better than the optimum. However, while A* rules out these states *explicitly*, our generalizations to A* will rule out many of these states *implicitly.*

4.2 Generalizing to Constraint-based A*

In this section we develop constraint-based A*, a variant of A* that solves optimal CSPs by exploiting MPI, but not conflicts. Framing an optimal CSP as a state space search problem, each search state is a partial assignment to decision variables $\mathbf{y}$. The initial state is the empty assignment $\{\}$. An operator takes a partial assignment, and adds an assignment to one of its unassigned variables. The Goal-Test returns true if the search state is a consistent assignment to all variables in $\mathbf{y}$, and the assignment satisfies each of the CSP's constraints. g is the cost of the partial assignment, and is computed by combining the individual assignment costs g_i, as defined in Section 2. By associativity and commutativity, cost is a function only of search state, and is independent of the order in which assignments are made.

The search tree of an optimal CSP, called an *assignment tree*, is similar to that for CSPs. Examples were given in Section 3.3. An unassigned variable is selected for each tree node that is not a leaf, and the branches of the node are labeled with alternative assignments to that variable. The set of assignments along a path from the tree root to a node is a partial assignment for the CSP, and represents the node's search state. The search state of a leaf node is a decision state of the Optimal CSP. Functions supporting the manipulation of assignment trees are given in Appendix B.

Our constraint-based variant of A* is given in Figure 12. It is distinguished by heuristic cost f and its node expansion function Expand-Variable, defined, respectively, in the next two subsections.

4.3 An Admissible Heuristic for Optimal CSPs

To be admissible, the cost f of a node n must be a lower bound on the cost of all states appearing below node n that satisfy Goal-Test. In the absence of additional information, we take this to be a lower bound on the cost of all full extensions to n's partial assignment. Pseudocode for functions corresponding to g and h are given in Appendix C. $g(n)$ is the cost of n's partial assignment, and is computed by applying $g[\text{OCSP}]$ to n's assignments. The heuristic cost of completion $h(n)$ is a lower bound on the cost of assignments to n's unassigned variables.

To define an h that may be computed efficiently, we exploit mutual preferential independence (MPI). Recall that if a cost function g is MPI, it follows that,

if $u \leq v$, then $G(u, w) \leq G(v, w)$.

```
function Constraint-based-A*(OCSP)
   returns the leading minimal cost solutions of OCSP.
   f(x) ← G[problem](g[OCSP](x), h[problem](x))
   Nodes[OCSP] ←
        Make-Queue(Make-Search-Tree-Node(Θ[OCSP],NoParent))
   solutions ← {}
   loop do
       if Terminate?(OCSP, solutions) or
              Nodes[OCSP] is empty
          then return solutions
          else node ← Remove-Best(Nodes[OCSP], f)
              Nodes[OCSP] ←
                   Enqueue(Nodes[OCSP], Expand-Variable(node, OCSP), f)
              if Goal-Test-State[OCSP] applied to State(node) succeeds
                then add State[node] to solutions
   end

function Goal-Test-State(node, problem)
   returns True iff node is a complete, consistent, decision state.
   if State[node] assigns values to all decision variables in problem
     then return Consistent?(State[node], CSP[problem])
     else return False
```

Fig. 12. Constraint-based A*.

Hence, the cost of a decision state is minimized by minimizing the cost of the assignment to each variable $y_i \in \mathbf{y}$ separately. Let $\mathbf{z}$ denote the set of unassigned variables of the OCSP at a particular search node. Then the minimum cost of assignment $\mathbf{z}$ is [9]

$$h(\mathbf{z}) = G(\{g_{z_i}^{min} | z_i \in \mathbf{z}, g_{z_i}^{min} = \min_{v_{ij} \in D_{z_i}} g_{z_i}(v_{ij})\}).$$

For example, in Figure 8, Boolean polycell included a tree node $n1$, corresponding to kernel $\{A2 = \mathbf{U}, O2 = \mathbf{U}\}$. The utility of the assignments in this kernel is

$$1/g(n1) = P_{A2}(U) \times P_{O2}(U) = .005 \times .01 = .00005.$$

Cost is minimized by maximizing probability, and the probability of each component is maximized if it is in the “Good” mode, hence,

$$1/h(n1) = P_{A1}(G) \times P_{O1}(G) \times P_{O3}(G) = -.995 \times .99 \times .99 = .975.$$

In general, since the definition of $h(\mathbf{z})$ is an optimistic estimate on the cost

[9] Let S be a set of attribute costs $\{s_i\}$. We use $G(S)$ to denote $G(s_1, s_2, \ldots s_n)$.

of all extensions, h is admissible, hence, constraint-based A* is guaranteed to come up with an optimal solution. h is only an estimate since, although a state must exist with cost $h(n)$, that state may be inconsistent with $C_{\mathbf{y}}$.[10]

4.4 Expanding the Best Child

To complete our development we define Expand-Variable. A straight forward implementation would perform expansion similar to backtrack search. Given a node n, Expand-Variable might first check to see if the state of n is consistent. If it is, it would then select any unassigned decision variable, and if such a variable exists, it would then generate a child of n for each possible value in that variable's domain. As with any CSP, the solution is insensitive to the order in which the variables are assigned, hence any one variable may be chosen to expand at each step, rather than all variables. In addition, since expansion is systematic, the A* search does not need to detect multiple paths to the same search state.

We can do better by exploiting mutual preferential independence to reduce the number of branches of the tree expanded during search.

Proposition 1 Let $c1$ and $c2$ be sibling nodes, where $c1$ is labeled with assignment $y_i = v_{ij}$, $c2$ is labeled with $y_i = v_{ik}$, and $g_i(v_{ij}) \leq g_i(v_{ik})$. Then there exists a leaf node $l1$ under $c1$ such that for all leaf nodes $l2$ under $c2$, $g(\text{State[l1]}) \leq g(\text{State[l2]})$.

For example, suppose we have a node n with state $\{O1 = \mathbf{U}\}$ (Figure 13). Furthermore, suppose we expand n using $O2$, hence, n has a child $c1$ for $\{O2 = G\}$ and a child $c2$ for $\{O2 = \mathbf{U}\}$. $g_i(O2 = G) = 1/.99$, while $g_i(O2 = \mathbf{U}) = 1/.01$, hence, $c1$ has a leaf that is $\leq$ all the leaves of $c2$. In particular, by MPI the best leaf, l_{1i}, of $c1$ is $\{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\}$, with cost $1/(.01\times.99\times.99\times.995\times.995) = 1/.0097$. This is better than the best leaf, l_{2j}, of $c2$, which by MPI is $\{O1 = \mathbf{U}, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = G\}$, with cost $1/(.01 \times .01 \times .99 \times .995 \times .995) = 1/.00098$. We note that these two best children only differ by the assignments to $O2$. This is a consequence of MPI.

Now consider how constraint-based A* expands a node n. It starts by selecting an unassigned variable y_i. We assume the values of D_i are ranked in increasing order of cost g_i, with v_1 denoting the value that minimizes g_i, v_2 denoting the

[10] This article employs a simple heuristic. More accurate heuristics, for example, based on Russian Doll search[30], mini-bucket heuristics[31] and lazy dynamic programming[32], have been extensively explored in the literature, and are synergistic with the approach presented here.

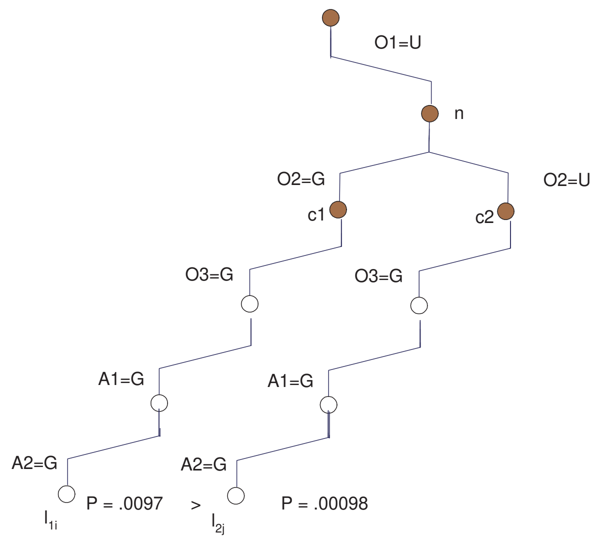

Fig. 13. An example demonstrating that, given two children, $c1$ and $c2$, of node n, the child $c1$ with the preferred cost, always contains a state l_{1i} with a preferred cost than the best state, l_{2j}, of $c2$.

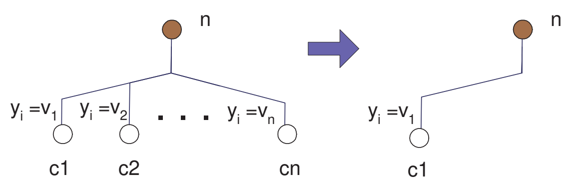

Fig. 14. Due to MPI, only the child of a node with the best cost assignment needs to be expanded (right), rather than all children (left).

second best value, and so forth. Likewise, we use $c1$ to denote the child with the best assignment, $y_i = v_1$, we use $c2$ to denote the child with the second best assignment, $y_i = v_2$, and so forth. The key consequence of Proposition 1 is that function Expand-Variable *only needs to create a node for the child,* $c1$, *with the best assignment,* $y_i = v_1$ (Figure 14). This follows because some leaf, l_{1n}, of $c1$ must exist whose cost is less than or equal to all leaves of the siblings of $c1$. Hence the best cost unexplored state contained by node n must be contained within $c1$, not its siblings. This best child is created by function Expand-Variable-Best-Child in Figure 15.[11]

Node c_1 is guaranteed to contain the best state *only until one or more of the states of* $c1$ *have been eliminated,* for example, by selecting one of $c1$'s leaf nodes as a candidate and testing consistency. At this point we may have eliminated c_1's best decision state $l1_n$, in which case the best leaf node of c_2 may be of lesser cost than the remaining unexplored leaves of c_1. Hence,

[11] For simplicity of presentation, in the figure, Expand-Domain orders the assignments by cost when the node is created. For efficiency, the implementation performs this ordering when the domains are created.

function Expand-Variable-Best-Child(*node, problem*)
  returns for *node*, a child with a best cost assignment.
  if all variables are assigned in State[*node*]
    then return {}
    else return Expand-Domain(*node, problem*)

function Expand-Domain(*node, problem*)
  returns the child with the best cost assignment of an unassigned variable.
  $y_i \leftarrow$ an unassigned variable in State[*node*] with the smallest domain.
  $C \leftarrow \{y_i = v_{ij} | v_{ij} \in D_i[problem]\}$
  Child-Assignments[*node*] $\leftarrow$ Sort C such that for i from 1 to $|C| - 1$,
    Better-Assignment?($C[k]$, $C[k+1]$, *problem*) is True
  $y_i = v_{ij} \leftarrow C[1]$, which is the best assignment in the domain of y_i
  return {Make-Node($\{y_i = v_{ij}\}$, *node*)}

function Better-Assignment?($y_i = v_{ij}, y_i = v_{ik}$, *problem*)
  returns True if the lower bound cost of a child node that adds
    assignment $y_i = v_{ij}$ is at least as good as a sibling adding $y_i = v_{ik}$.
  return $g_{y_i}[problem](v_{ij}) \leq g_{y_i}[problem](v_{ik})$

Fig. 15. Expanding the best child for Constraint-Based-A*.

once a leaf of a node c_n is eliminated, constraint-based A* must create a node for c_n's next best sibling c_{n+1}. This sibling is created using the function Expand-Next-Best-Sibling, shown in Figure 16. When a leaf is expanded, a next best sibling is created for every ancestor of the leaf by function Expand-Next-Best-Sibling-of-Ancestors. This approach to expansion is summarized in Figure 16.[12]

A* traditionally expands all children of a node, producing at most b nodes, where b is the maximum variable domain size, $b = \max_i |D_i|$. Each call to expand increases the size of the queue by $b - 1$ nodes, producing a worst case growth of $(b - 1) \times n$ after n iterations. In contrast, our strategy grows the queue by one node at each step (two new nodes are added, and one is removed), producing a worst case growth of only n nodes after n iterations. In practice, this is an important reduction in queue growth. Our strategy preserves the key properties of optimality and completeness, that is, it expands leaves in best first order and it eventually reaches all tree leaves, given that the variable domains are finite.

[12] Also note that constraint-based A* only expands a node when its partial assignment proves consistent, similar to backtrack search.

```
function Expand-Variable(node, problem)
  returns the best nodes expanded from node.
  if Consistent?(State[node], CSP[problem])
    then nodes ← Expand-Variable-Best-Child(node, problem)
        if Leaf-Node?(node, problem)
          then nodes ← nodes ∪
                Expand-Next-Best-Sibling-of-Ancestors(node, problem)
        return nodes
    else return {}

function Expand-Next-Best-Sibling-of-Ancestors(node, problem)
  returns siblings of node and its ancestors with the next best assignment.
  if Root?[node]
    then return {}
    else return Expand-Next-Best-Sibling(node, problem) ∪
      Expand-Next-Best-Sibling-of-Ancestors(Parent[node], problem)

function Expand-Next-Best-Sibling(node, problem)
  returns node's sibling with the next best assignment
    in Child-Assignments[Parent[node]].
  if Root?[node]
    then return {}
    else {y_i = v_ij} ← Assignment[node]
         {y_k = v_kl} ← next assignment in Child-Assignments[Parent[node]]
                        after {y_i = v_ij}
         if no next assignment {y_k = v_kl}
               or Parent[node] already has a child with {y_k = v_kl}
           then return {}
           else return {Make-Node({y_k = v_kl}, Parent[node])}
```

Fig. 16. Expanding the best sibling for constraint-based A*.

4.5 Constraint-based A Applied to Boolean Polycell*

Returning to Boolean polycell, constraint-based A* begins with the root node n_1 on the search queue. The root is dequeued and its best child n_2 is expanded and enqueued, by selecting $O3$ as an unassigned variable and assigning it its best assignment, G. A similar process generates n_3 - n_5 and finally n_6, which is the best state, and hence Candidate 1 (Figure 17, top left),

$$\{O1 = G, O2 = G, O3 = G, A1 = G, A2 = G\}.$$

Node n_6 is a leaf node, hence when it is dequeued, Expand-Variable generates the next best sibling of that node and all its ancestors, producing $n_7 - n_{11}$ (Figure 17, bottom left).

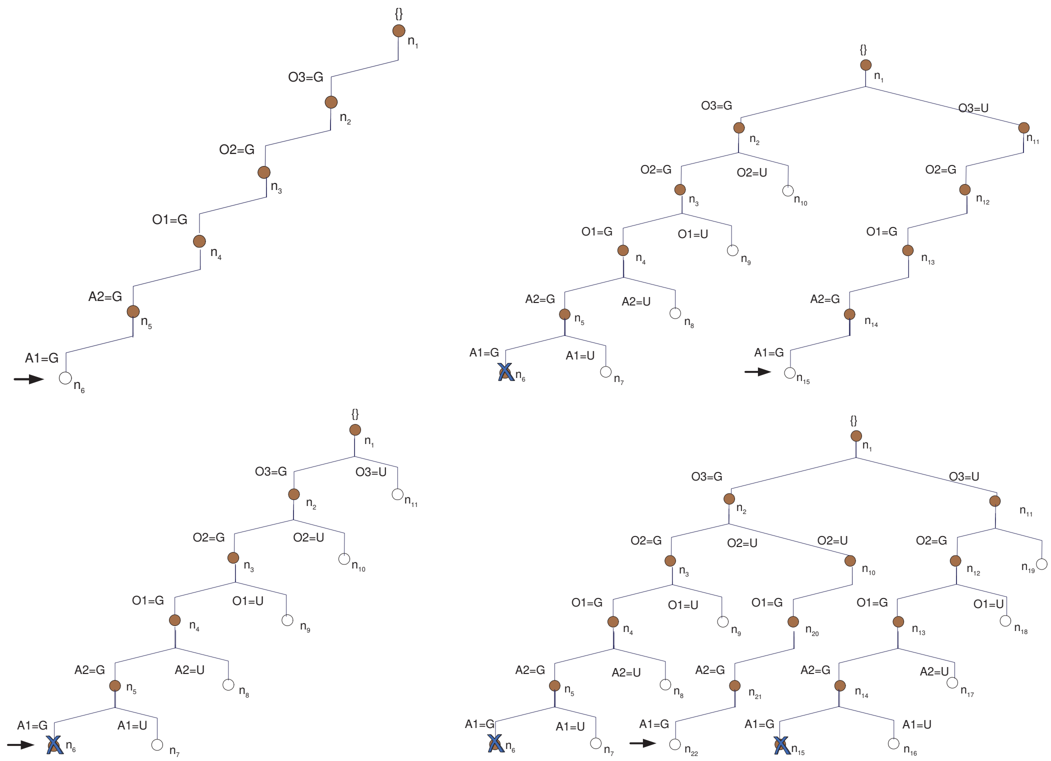

Fig. 17. Constraint-based-A* search tree for Boolean Polycell, for the best three states. Top left) Expanding the best descendants to create the best state (n_6). Bottom left) When n_6 is dequeued, its best sibling and ancestors are created. Top right) Expanding the descendants of n_{11} to produce the second best state (n_{15}). Bottom right) Expanding the descendants of n_{10} to produce the third best state (n_{15}).

Constraint-Based-A* uses Goal-Test-State to test Candidate 1, which proves inconsistent. Continuing the search, nodes $n_9 - n_{11}$ are at the front of the queue, all with the same cost. Assuming that n_{11} is first dequeued, Expand-Variable repeatedly generates its best descendants, producing $n_{12} - n_{15}$ (Figure 17, top right). n_{15} is a complete assignment, and is returned as the second candidate,

$$\{O1 = G, O2 = G, O3 = \mathbf{U}, A1 = G, A2 = G\}.$$

Since n_{15} is a leaf node, when it is dequeued, Expand-Variable generates its next best sibling and ancestors, $n_{16} - n_{19}$ (Figure 17, bottom right). No next best sibling for n_1 is generated, because the domain of O_3 has been exhausted. The candidate tests inconsistent.

Likewise, the third round of the search expands n_{10}, generating the best descendants $n_{20} - n_{22}$ (Figure 17, bottom right), and the third candidate,

$$\{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = G\}.$$

which also proves inconsistent. [13] The process repeats until the desired set of best consistent candidates has been found.

4.6 Summary

Constraint-based A* is based on three concepts. First, an OCSP may be solved by performing an A* search on a tree representing the space of all partial assignments, similar to traditional backtrack search. Second, MPI enables us to efficiently estimate the cost-to-go of a partial assignment. This function, h, simply selects the assignment with the best attribute cost for each unassigned variable. Finally, the most interesting concept is that queue growth is reduced by only expanding the best child for each node, waiting until one of the child's states is removed, before expanding its next best sibling.

5 Generating the Best Kernel

Recall that conflict-directed A* uses Next-Best-State-Resolving-Conflicts to jump over the leading set of *conflicting* states, as we demonstrated in Section 3.3. It accomplishes this by using function Next-Best-Kernel to generate the *kernel* containing the best state. We develop Next-Best-Kernel in this section.

Early diagnostic approaches [1,19] generated a complete description of the diagnostic space by generating all kernels, given all conflicts. In the worst case, however, the complete set of kernels may be exponential in the number of components. Next-Best-Kernel allows us to address this problem by generating the kernels in best first order, stopping when the generated kernels cover most or all "good" solutions. The approach offers an any-time, any-space algorithm, which increases its coverage of the solution space as additional time and memory permits.

For diagnosis, this approach is particularly effective in the common case, where a small collection of kernels covers most of the probability density of valid diagnoses, while the remaining, exponential number of kernels collectively cover a small portion of the probability density space.

Within conflict-directed A*, Best-Kernels operates on a subset of the complete set of conflicts, and hence produces "approximate" kernels, which together contain all solutions, but may also include inconsistent decision states. These decision states are pruned by Goal-Test, by testing consistency using a CSP solver.

[13] This corresponds to the second candidate tested by conflict-directed A*.

5.1 *Conflicts to Kernels*

We begin by making our terms precise. A partial assignment to the variables of a CSP denotes a subset of the state space of the CSP. A conflict is a partial assignment that is inconsistent. Any state that is a superset of this conflict is also inconsistent. Hence a conflict denotes an inconsistent subset of the state space.

Definition 2 Let $\mathbf{y}$ be a set of variables with state space $\mathbb{S}_{\mathbf{y}}$, and let $C_{\mathbf{y}}$ be a constraint on $\mathbf{y}$. A *conflict* α of constraint $C_{\mathbf{y}}$ is a partial assignment to $\mathbf{y}$ such that every state that extends α is inconsistent with $C_{\mathbf{y}}$. Let α be a conflict of $C_{\mathbf{y}}$, and s be a state $s \in \mathbb{S}_{\mathbf{y}}$, then s *manifests* α if $\alpha \subset s$; otherwise, s *resolves* α.

For example, from Section 3.2, Candidate 1 of Boolean polycell is a state $s_1 : \{O1 = G, O2 = G, O3 = G, A1 = G, A2 = G\}$, which manifests conflicts:

Conflict 1: $\{O1 = G, O2 = G, A1 = G\}$ and

Conflict 2: $\{O1 = G, A1 = G, A2 = G\}$,

To jump over subspaces containing conflicting states, Conflict-directed A* *inverts* the known conflicts, by generating descriptions of all subsets of the state space that resolve these conflicts. A *kernel* is a partial assignment denoting a subspace, such that each state in the subspace resolves the complete set of conflicts. An essential property of the set of all kernels is that it forms a complete description. Every state contained by a kernel resolves every known conflict, and each state that resolves all conflicts is the state of at least one kernel. To be complete, conflict-directed A* must be able to generate all kernels for a given set of known conflicts. [14]

Definition 3 Let $\mathbf{y}$ be a set of variables with partial assignments $\mathbb{P}_{\mathbf{y}}$, let $C_{\mathbf{y}}$ be a constraint on $\mathbf{y}$, and let Γ be a set of conflicts for $C_{\mathbf{y}}$. A partial assignment $\alpha \in \mathbb{P}_{\mathbf{y}}$ *resolves conflicts* Γ if every state of α resolves every conflict $\gamma \in \Gamma$. Partial assignment α is a *kernel* of Γ if α resolves Γ, and no proper subset β of α exists that resolves Γ. The *kernels of* Γ is the set $\{\beta \in \mathbb{P}_{\mathbf{y}} | \beta \text{ is a kernel of } \Gamma\}$.

[14] Note, our concept of kernel is similar to that of kernel diagnosis in [2], except that a kernel resolves the *known* conflicts, while a kernel diagnosis resolves *all* conflicts.

The kernels for Conflict 1 and 2 are:

Kernel 1: $\{O1 = \mathbf{U}\}$

Kernel 2: $\{A1 = \mathbf{U}\}$

Kernel 3: $\{O2 = \mathbf{U}, A2 = \mathbf{U}\}$.

To generate the kernels of conflicts Γ we first generate the kernels of each conflict separately. We call these the *constituent kernels* of Γ.

Proposition 2 Let $C_{\mathbf{y}}$ be a constraint on $\mathbf{y}$, and Γ be a set of conflicts of $C_{\mathbf{y}}$, then the *constituent kernels* of Γ is the set $\{\text{kernels of } \gamma | \gamma \in \Gamma\}$. *The set of constituent kernels of conflict γ is*

$$K_\gamma \equiv_{def} \left\{ \{a\} | a \in \left(\cup_{\langle x_i = v_{ik} \rangle \in \gamma} D_{x_i} \right) - \gamma \right\}.$$

For example, suppose Γ consists of Conflict 1 and 2, identified earlier. We create the complete set of constituent kernels for Conflict 1 by replacing each assignment of Conflict 1 with its alternative domain assignments, and likewise for Conflict 2, respectively,

$$\{\{\{O1 = \mathbf{U}\}, \{O2 = \mathbf{U}\}, \{A1 = \mathbf{U}\}\} \text{ and}$$
$$\{\{O1 = \mathbf{U}\}, \{A1 = \mathbf{U}\}, \{A2 = \mathbf{U}\}\}\}.$$

The procedure Constituent-Kernels follows directly from Proposition 2. The proof of Proposition 2, and the pseudocode for Constituent-Kernels, is presented in Appendix D. Constituent-Kernels incurs negligible computational cost; its worst case computational complexity is on the order of $\sum_{D_{x_i} \in \mathbf{D_x}} |D_{x_i}|$.

Next, to generate the kernels of Γ from its constituent kernels, we note:

Proposition 3 *A kernel k resolves a set of conflicts Γ if and only if it resolves each conflict $\gamma_i \in \Gamma$. k resolves γ_i if and only if it contains one of the kernels of γ_i.*

Hence, each kernel, $k \in K_\Gamma$, is a set that selects at least one kernel from each set of constituent kernels, K_γ, and takes their union. For example, we might combine $\{O2 = \mathbf{U}\}$, from the constituent kernels of Conflict 1, with $\{A2 = \mathbf{U}\}$, from the constituent kernels of Conflict 2, producing kernel $\{O2 = \mathbf{U}, A2 = \mathbf{U}\}$. A kernel must be minimal, hence we exclude any union that is a superset of another union. To be consistent a kernel can assign at most one value to any variable; hence, we eliminate any union containing two distinct assignments for the same variable. This is analogous to the candidate generation algorithm used in the GDE system[1], whose soundness and completeness was demonstrated by Corollary 1 of [2]. Kernel generation is NP Hard and the number of kernels is worst case exponential in the number of conflicts.

5.2 In Search of the Best

To make conflict-directed A* tractable, we require an efficient means for finding the kernel that contains the best cost state, while generating as few kernels as possible. To accomplish this we note that the process of generating kernels may be viewed as a state space search through a space of partial kernels. A search tree for our example was shown earlier in Figure 8. At each iteration of this search, a partial kernel is expanded to resolve an additional conflict, terminating when all conflicts are resolved. A partial kernel is pruned if it either proves inconsistent, redundant, or non-minimal. The function Best-Kernels is given in Figure 18, as a variant on A* search.

The heuristic cost function, $f(n) = g(n) + h(n)$, for Best-Kernels is the same as that for Constraint-based A*, defined in Appendix C. The goal test and node expansion functions must be modified, as discussed below.

5.2.1 Detecting Kernels

One difference from constraint-based A* is that the leaves of the tree for Next-Best-Kernel are kernels, rather than full assignments. This requires modification to Goal-Test, so that it returns true as soon as a node covers all constituent kernels, and hence all conflicts have been resolved (Figure 19).

5.2.2 Expanding Partial Kernels

Expand-Conflict selects one of the sets of constituent kernels for an unresolved conflict, and creates a child for each kernel in the constituent. For example, the root node $\{\}$, shown earlier in Figure 8, does not resolve Conflict 1 or Conflict 2. It is expanded by selecting Conflict 1, and its constituent kernels are used to generate three children, labeled $O2 = \mathbf{U}$, $O1 = \mathbf{U}$ and $A1 = \mathbf{U}$. Given multiple possible conflicts to choose from, Expand-Conflict selects the conflict with the fewest number of constituent kernels. This corresponds to the standard most-constrained-variable-heuristic, used by most CSP algorithms.

Like constraint-based A*, mutual preferential independence allows conflict-directed A* to reduce the number of branches of the tree expanded during search. However, there is an important difference, due to the fact that for

function Initialize-Best-Kernels(*KGP*)
  returns Kernel generation problem, KGP, with its search-tree initialized.
  Best-Kernels[*KGP*] $\leftarrow \{\}$
  Nodes[*KGP*] $\leftarrow$
    Make-Queue(Make-Search-Tree-Node(Θ[*KGP*],NoParent))
  Visited[*KGP*] $\leftarrow \{\}$
  return KGP

function Next-Best-Kernel(*KGP*)
  returns the next best cost kernel of Conflicts[*KGP*]
    for kernel generation problem *KGP*.
  $f(\mathbf{x}) \leftarrow G[KGP](g[KGP](\mathbf{x}), h[KGP](\mathbf{x}))$
  loop do
    if Nodes[*KGP*] is empty **then return** failure
    node $\leftarrow$ Remove-Best(Nodes[*KGP*], *f*)
    Add State(*node*) to Visited[*KGP*]
    new-nodes $\leftarrow$ Expand-Conflict(*node*, *KGP*)
    for each *new-node* in *new-nodes*
      unless $\exists\ n \in$ Nodes[*KGP*] such that State(*new-node*) = State(n)
        or State(*new-node*) is in Visited[*KGP*]
       then Nodes[*KGP*] $\leftarrow$ Enqueue(Nodes[*KGP*], *new-node*, *f*)
    if Goal-Test-Kernel[*KGP*] applied to State(*node*) succeeds
      then *best-kernel* = State(*node*)
        Best-Kernels[*KGP*] $\leftarrow$
         Add-To-Minimal-Sets(Best-Kernels[*KGP*], *best-kernel*)
        if *best-kernel* $\in$ Best-Kernels[*KGP*]
         then return *best-kernel*
  end

Fig. 18. Generating the best kernels of a set of conflicts using A* search. A kernel generation problem, KGP, includes a set of Conflicts and initial state $\Theta = \{\}$. Functions Goal-Test-Kernel and Expand-Conflict are shown in Figures 19 and 20. Functions Make-Tree-Node, Root?, State and Theta are the same as for Constraint-Based-A*, and were given in Appendix B. g, h, G_{min} and g_{min} are also the same, and were given in Appendix C.

function Goal-Test-Kernel(*node, problem*)
  returns True iff the state of *node* resolves all known conflicts.
  if for all $K_\gamma \in K_\Gamma[problem]$, State[*node*] contains a kernel in K_γ
   then return True
   else return False

Fig. 19. Goal-Test-Kernels used by Next-Best-Kernel to detect kernels.

conflict-directed A*, the assignments associated with siblings may involve distinct variables.

Proposition 4 Let $c1$ and $c2$ be sibling nodes with parent n, where $c1$ is labeled with assignment $y_i = v_{ij}$, $c2$ is labeled with $y_k = v_{kl}$, and neither y_i nor y_k appear in State[n]. Let $g_{y_i}^{min}$ and $g_{y_k}^{min}$ denote the best attribute costs of y_i and y_k, respectively. If $G(g_{y_i}(v_{ij}), g_{y_k}^{min}) \leq G(g_{y_i}^{min}, g_{y_k}(v_{kl}))$, then there exists a leaf node $l1$ under $c1$ such that for all leaf nodes $l2$ under $c2$, $g(\text{State}[l1]) \leq g(\text{State}[l2])$.

Note that $c1$ doesn't restrict the value of y_k, and $c2$ doesn't restrict the value of y_i. Hence to identify the child with the best state, the comparison is performed under the assumption that the two children take on best cost values for their sibling's variable.

For example, consider the node labeled $O2 = \mathbf{U}$ in Figure 8. The first of its three children, $c1$, has assignment $O1 = \mathbf{U}$, and the second child, $c2$, has assignment $A2 = \mathbf{U}$. $c1$ is preferred over $c2$, since

$$1/(P(O1 = \mathbf{U}) \times P_{max}(A2)) \leq 1/(P(A2 = \mathbf{U}) \times P_{max}(O1)).$$

Next, consider how this proposition is incorporated into function Expand-Conflict of Next-Best-Kernel. Given a node n, Expand-Conflicts begins by identifying an unresolved conflict. A conflict is unresolved by node n if none of the conflict's constituent kernels is a subset of State(n). We order the constituent kernels of the conflict using function Better-Kernel?, shown in Figure 20. Let k_n denote the nth kernel in this ordering, and c_n denote the corresponding child. It follows from Proposition 4 that only the first child, c_1, needs to be expanded. This is performed by function Expand-Conflict-Best-Child, in Figure 20.

Proposition 4 only holds until one or more of the states of a child c_n has been eliminated. This occurs *as soon as c_n is expanded*, in order to resolve an additional conflict, since that conflict may eliminate one or more of the states of c_n. Hence, as soon as a child of node c_n is expanded, the next best sibling, c_{n+1}, of c_n must be expanded as well. The pattern of node expansion is then to repeatedly replace the best cost node on the search queue, with its best child and its next best sibling. This expansion is achieved with functions Expand-Conflict (Figure 20) and Expand-Next-Best-Sibling (Figure 16).

Next-Best-Kernel uses a variant of the dynamic programming principle of A* search to avoid extending multiple paths that go to the same state. To accomplish this, Next-Best-Kernel keeps track of nodes that it has already explored using the variable *visited*. As each node is queued, we check to see if a node with the same search-state already exists on the queue or visited list. If

function Expand-Conflict(*node, problem*)
  returns the best nodes expanded from *node*.
  return Expand-Conflict-Best-Child(*node, problem*) $\cup$
    Expand-Next-Best-Sibling(*node, problem*)

function Expand-Conflict-Best-Child(*node, problem*)
  returns for *node,* a child with the best cost extension.
  if for all $K_\gamma \in$ Constituent-Kernels(Γ[*problem*])
    State[*node*] contains a kernel in K_γ
   then return {}
   else return Expand-Constituent-Kernel(*node, problem*)

function Expand-Constituent-Kernel(*node, problem*)
  returns for *node*, the child containing the best cost kernel of a
    conflict not already resolved by State[*node*].
  $K_\gamma \leftarrow$ the smallest set in Constituent-Kernels(Γ[*problem*]),
    such that no kernel in the set is contained in State[*node*].
  $C \leftarrow \{y_i = v_{ij} | \{y_i = v_{ij}\} \in K_\gamma, y_i = v_{ij}$ is consistent with State[*node*]$\}$
  Sort C such that for all i from 1 to $|C| - 1$,
   Better-Kernel?($C[i]$, $C[i+1]$, *problem*) is True
  Child-Assignments[*node*] $\leftarrow C$
  $y_i = v_{ij} \leftarrow C[1]$, which is the best kernel in K_γ consistent with State[*node*]
  return {Make-Node($\{y_i = v_{ij}\}$, *node*)}

function Better-Kernel?($y_i = v_{ij}, y_k = v_{kl}$, *problem*)
  returns True if the lower bound cost of a child node that adds
    kernel $y_i = v_{ij}$ is better than a sibling that adds kernel $y_k = v_{kl}$.
  if $y_i = y_k$
   then return $g_{y_i}[problem](v_{ij}) \leq g_{y_k}[problem](v_{kl})$
   else return $G[problem](g_{y_i}[problem](v_{ij}),\ g_{min}(y_k, problem))$
    $\leq G[problem](g_{min}(y_i, problem),\ g_{y_k}[problem](v_{kl}))$

Fig. 20. Expand-Conflict used by Next-Best-Kernel to cover known conflicts. Expand-Next-Best-Sibling is the same as for Constraint-Based-A* and is shown in Figure 16. g, h, G_{min} and g_{min} are also the same, and are given in Appendix B. Likewise, Make-Tree-Node, Root?, State and Theta are given in Appendix B.

so, then the node is ignored. This has a substantial impact on our performance experiments, discussed in Section 7.

5.3 Summary

In this section we introduced an algorithm, called Next-Best-Kernel, that generates the kernels of a set of conflicts in best first order. Next-Best-Kernel combines A* search with traditional algorithms for generating kernel diagnoses. It achieves efficiency by exploiting mutual preferential independence and a special case of the dynamic programming principle, in order to restrict the set of nodes expanded during search. Next-Best-Kernel is used by Conflict-Directed-A* to extract the best state that resolves the known conflicts, as we will see in the next section. It also provides an any-time, any-space algorithm for generating parsimonious descriptions of the best solutions.

6 Conflict-directed A*

The top-level procedure of conflict-directed A* (Figure 4) was introduced and demonstrated in Section 3. This section completes the development of conflict-directed A*, by specifying the candidate generation procedure, Next-Best-State-Resolving-Conflicts. First we consider the case where we are only interested in the single best solution, building upon the function Next-Best-Kernel (Section 5). This case corresponds to the algorithm demonstrated earlier in Section 3. Next, we generalize conflict-directed A* to finding any number of leading solutions. To accomplish this we develop a hybrid version of Next-State-Resolving-Conflicts that unifies Constraint-based-A* and Next-Best-Kernel, (from Sections 4 and 5, respectively).

6.1 Conflict-directed A: One Solution*

To generate the single best solution of an optimal CSP, at each iteration Next-Best-State-Resolving-Conflicts simply extracts the best kernel, and then extends the kernel to the best complete decision state (Figure 21). The best kernel K is extracted from the conflicts using Next-Best-Kernel, developed in the preceding section. In our preceding development, Next-Best-Kernel assumed that the set of conflicts was fixed. In this case, if a partial assignment was found to be a kernel, then it would be removed from the search queue without further expansion. For conflict-directed A*, however, newly discovered conflicts can be added after kernels have been generated. In this case the set of generated kernels may no longer resolve all conflicts, and must be incrementally updated. To address this, whenever a new conflict is extracted (Figure 4), the function Update-Known-Kernels-Based-On-New-Conflicts is called, which searches through the set of generated kernels, Best-Kernels[*OCSP*], and tests whether each kernel resolves all new conflicts. If a kernel does not, the kernel is removed from Best-Kernels and its corresponding search node, Node(*kernel*), is queued for expansion, in order to resolve all new conflicts (Figure 21). The net result is an incremental best-first kernel generation algorithm that interleaves kernel generation with conflict insertion.

function Terminate?(*OCSP*)
  returns True when first solution of *OCSP* is found.
  return True iff Solutions[*OCSP*] is non-empty.

function Next-Best-State-Resolving-Conflicts(*OCSP*)
  returns the best cost state consistent with Conflicts[*OCSP*].
  best-kernel ← Next-Best-Kernel(*OCSP*)
  if *best-kernel* = failure
    then return failure
    else return Kernel-Best-State[*problem*](*best-kernel*)

function Update-Known-Kernels-Based-On-New-Conflicts(*OCSP*)
  returns OCSP with kernels and queue updated to reflect kernels requiring further expansion.
  for each *kernel* in Best-Kernels[*OCSP*]
      unless Goal-Test-Kernel[*OCSP*] applied to *kernel* succeeds
        then Best-Kernels[*OCSP*] ← Remove(*kernel*, Best-Kernels[*OCSP*])
          {*new-node*} ← Expand-Constituent-Kernel(Node[*kernel*], *OCSP*)
          unless ∃ $n \in$ Nodes[*OCSP*] such that State(*new-node*) = State(n)
              or State(*new-node*) is in Visited[*OCSP*]
            then Nodes[*OCSP*] ← Enqueue(Nodes[*OCSP*], *new-node*, *f*)
  return *OCSP*

function Kernel-Best-State[*problem*](*kernel*)
  returns the best utility state of *kernel*.
  unassigned ← all variables not assigned in *kernel*
  return *kernel* ∪ Best-Assignment(*unassigned*)

function Best-Assignment[*problem*](*variables*)
  returns the maximum utility assignment to *variables*.
  if *variables* = {}
    then return {}
    else y_i = one of *variables*
        remaining = *variables* - $\{y_i\}$
        return $\{y_i = v_{max}[problem](y_i)\} \cup$ Best-Assignment[*problem*](*remaining*)

function $v_{max}[problem](y_i)$
  returns the value with the maximum attribute utility for y_i.
  return $\arg\max_{v_{ij} \in D_i[problem]} g_i[problem](v_{ij})$

Fig. 21. Support functions for Conflict-directed-A* for the case of generating a single best solution.

To extract the best state of kernel K, let $\mathbf{z}$ be the set of variables not assigned in K. Then the best cost decision state, s, of K is the one that selects for each unassigned variable $z_i \in \mathbf{z}$ its best cost value,

$$s \equiv K \cup \left\{ z_i = v_{i_{min}} \mid z_i \in \mathbf{z}, v_{i_{min}} = \arg \min_{v_{ij} \in D_{z_i}} g_i(v_{ij}) \right\}.$$

This corresponds to Function Kernel-Best-State (Figure 21). This version of conflict-directed A* was demonstrated in detail in Section 3.

A key property of most systematic CSP search algorithms is that the same search state is not visited twice; this property, called *systematicity* can have enormous impact on search efficiency. Systematicity is satisfied for constraint-based A*. To ensure this property for conflict-directed A*, our development requires modification. In particular, two kernels can denote overlapping subspaces, in which case the kernels can be extended to the same full assignment. This full assignment can be generated more than once, for example, if it represents the optimal state for both kernels.

The overlap in kernels is introduced by Expand-Conflict (Figure 20), which splits on a set of constituent kernels, and results from child nodes making assignments to different variables. To eliminate this overlap, we augment the state of a child node so that it excludes the kernel assignment of each sibling node that was created before it. In particular, consider the last line of function Expand-Next-Best-Sibling (Figure 16), which creates each child node after the first. For conflict-directed A*, whenever this function is called *by Expand-Conflict*, the search state $\{y_k = v_{kl}\}$ is augmented with $\neg(y_i = v_{ij})$ for any sibling assignment $y_i = v_{ij}$ that precedes it in Child-Assignments[Parent(*Node*)]. In addition, a node is not generated if its augmented state is inconsistent. For simplicity of presentation, this addition is not included in the pseudocode of this article, but is straightforward to introduce (see [48]).

6.2 Conflict-directed A: Multiple Solutions*

To generate multiple leading solutions, we introduce a variant of Next-Best-State-Resolving-Conflicts that is able to enumerate, in best first order, multiple decision states of one or more kernels. This is in contrast to the preceding section, where Next-Best-State-Resolving-Conflicts can only enumerate the single best decision state of each kernel.

Our augmented Next-Best-State-Resolving-Conflicts, is defined in Figure 22. It generates kernels similar to Next-Best-Kernel (Figure 20), and enumerates the states of these kernels, similar to Constraint-Based-A* (Figure 22). To efficiently focus the search, it interleaves the processes of generating best kernels and best states. In particular, at each iteration it selects for expansion the node from the two search processes that looks most promising according to f. To implement this, Next-Best-State-Resolving-Conflicts uses a single search queue that contains nodes of both search types. The function Expand-State-Resolving-Conflicts expands each node based on this type, using Expand-Conflict to expand partial kernels and Expand-Variable to expand

```
function Next-Best-State-Resolving-Conflicts(OCSP)
  returns the best cost state consistent with Conflicts[OCSP].
  f(x) ← G[problem](g[problem](x), h[problem](x))
  loop do
     if Nodes[OCSP] is empty
        then return failure
        else node ← Remove-Best(Nodes[OCSP], f)
           Add State(node) to Visited[OCSP]
           new-nodes ← Expand-State-Resolving-Conflicts(node, OCSP)
           for each new in new-nodes
              unless ∃ n ∈ Nodes[OCSP] such that State(new) = State(n)
                     or State(new) is in Visited[OCSP]
                 then Nodes[OCSP] ← Enqueue(Nodes[OCSP], new, f)
           if Goal-Test-State-Resolves-Conflicts[OCSP](State(node)) succeeds
             then return node
  end

function Expand-State-Resolving-Conflicts(node, problem)
  returns Best nodes expanded from node.
  if forall K_γ ∈ Constituent-Kernels(Γ[problem]),
       State[node] contains a kernel in K_γ
    then if all variables are assigned in State[node]
         then return {}
         else return Expand-Variable(node, problem)
    else return Expand-Conflict(node, problem)

function Goal-Test-State-Resolves-Conflicts(node, problem)
  returns True iff node is a complete decision state
    that resolves all known conflicts.
  if forall K_γ ∈ Constituent-Kernels(Γ[problem]),
       State[node] contains a kernel in K_γ
    then if all variables are assigned in State[node]
         then return True
         else return False
    else return False

function Update-Known-Kernels-Based-On-New-Conflicts(OCSP)
  return OCSP
```

Fig. 22. Support functions for Conflict-directed-A* for the case of generating multiple solutions. Combines expansion functions for Next-Best-Kernel (Figure 20) and Constraint-based-A* (Figure 22). Terminate? is application specific and is not supplied.

kernels to states. The goal-test function, Goal-Test-State-Resolves-Conflicts, returns true when a search state is a complete assignment and it resolves all conflicts. The application of the dynamic programming principle is the same as outlined at the end of Section 5.2.2 for Next-Best-Kernel.

Note that Next-Best-State-Resolving-Conflicts does not need to maintain the set of Best-Kernels in order to generate solutions. If these kernels are desired, they can be maintained incrementally similar to Section 6.1. The key difference is that, unlike Section 6.1, Update-Known-Kernels-Based-On-New-Conflicts does not need to search Best-Kernels for kernels requiring additional expansion, since kernels are *always* expanded by Next-Best-State-Resolving-Conflicts.

To ensure that search is systematic, the function Expand-Next-Best-Sibling is modified as outlined in the preceding section (Section 6.1), thus guaranteeing that the subspaces denoted by child nodes are non-overlapping. Once again, this change is only made for calls to Expand-Next-Best-Sibling from Expand-Conflict; siblings generated by calls from Expand-Variable are already guaranteed to be non-overlapping.

6.3 Full Conflict-directed A Applied to Boolean Polycell*

Consider a trace of our extended version of conflict-directed A*, applied to Boolean polycell. Initially there are no conflicts, and the root node n_1 is on the search queue. On the first iteration of conflict-directed A*, Next-Best-State-Resolving-Conflicts starts by dequeuing n_1. Since there are no conflicts to be resolved, the best descendants of n_1 are expanded similar to constraint-based A* (Section 4.5), producing nodes $n_2 - n_6$ (Figure 23, top left). The best state, n_6, is returned,

Candidate 1: $\{O1 = G, O2 = G, O3 = G, A1 = G, A2 = G\}$.

Conflict-Directed-A* finds Candidate 1 inconsistent, generating

Conflict 1: $\{O1 = G, O2 = G, A1 = G\}$.

Next-Best-State-Resolving-Conflicts is reinvoked with this new conflict and the current search agenda. Since n_6 is eliminated, its next best sibling and ancestors are generated. $n_9 - n_{11}$ (Figure 23, top right) are at the front of the queue, all with the same cost. Assuming that n_{11} is dequeued, n_{11} does not resolve Conflict 1, hence a best child n_{12} is generated for n_{11} that selects the best cost constituent kernel, $\{O2 = \mathbf{U}\}$, for Conflict 1. Note that this kernel

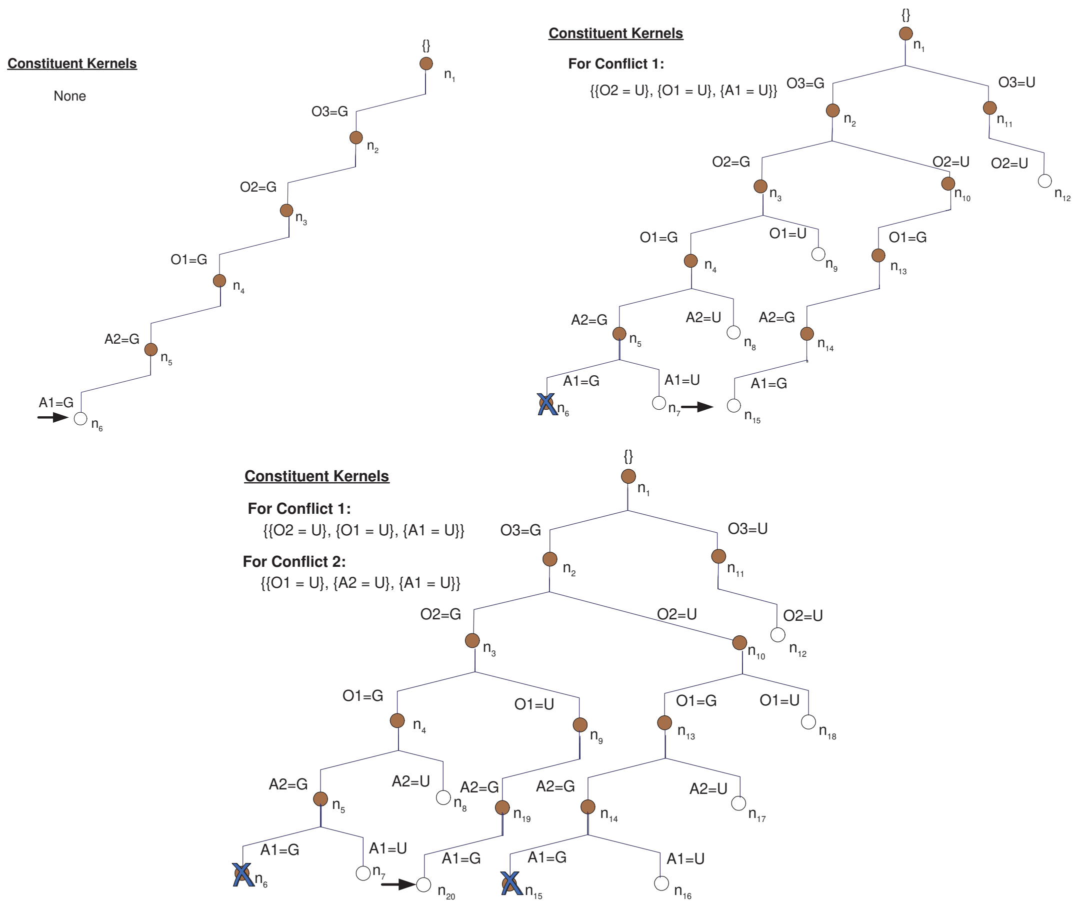

Fig. 23. Search trees generated for Boolean polycell, by the extended version of Next-Best-State-Resolving-Conflicts. The best decision state is indicated by an arrow. A closed/open circle indicates an expanded/unexpanded node. Top left) the best cost state, given no conflicts. Top right) the best cost state, given Conflict 1. Bottom) the best cost state, given Conflicts 1 and 2.

adds an additional failure (O2 broken), and hence the cost n_{12} is about an order of magnitude worse than that of n_{11}.

The next best node taken off the search queue is n_{10}, which has the same cost as n_{11}. This node already resolves Conflict 1, hence the node is recursively expanded to its best state, by repeatedly selecting an unassigned variable and assigning it its best cost value ($n_{13} - n_{15}$, Figure 23, top right). n_{15} is returned as the best state that resolves the known conflicts,

$$\text{Candidate 2:} \;\; \{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = G\}.$$

Conflict-Directed-A* determines that Candidate 2 is also inconsistent, generating

Conflict 2: $\{O1 = G, A1 = G, O2 = G\}$.

Conflict 2 is added and Next-Best-State-Resolving-Conflicts is invoked for a third round. Since n_{15} was removed, its next best siblings and ancestors are generated, producing $n_{16} - n_{18}$. Next the best node n_9 is dequeued. n_9 resolves both Conflict 1 and Conflict 2, hence its best descendants n_{19} and n_{20} are expanded (Figure 23, bottom), producing

Candidate 3: $\{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\}$.

Conflict-directed A* determines that Candidate 1 is consistent, and hence the best diagnosis. At this point all conflicts have been discovered, hence subsequent invocations of Next-Best-State-Resolving-Conflicts generates all diagnoses in best first order, without visiting any additional, inconsistent states.

7 Experimental Results

We evaluated the performance of conflict-directed A* both on applications to real world space systems and on randomly generated problems. To measure the effectiveness of exploiting conflicts and mutual preferential independence, we ran parallel tests with constraint-based A*. Starting with real-world applications, we have employed variants of conflict-directed A* in a range of model-based diagnosis and model-based autonomous systems, including Livingstone[8], Burton[9], MiniMe[34] and Titan[10]. These have been applied to space, naval and automotive systems. Applications to space systems in flight include NASA's Deep Space One probe and Earth Observer One satellite. Space demonstrations in a ground testbed or in simulation include the Cassini Saturn space probe, the Air Force TechSat 21 cluster, NASA's Messenger mission, the X-34 and X-37 NASA's ST-7 concept mission, and the MIT Sphere's mission. The performance of an earlier variant of conflict-directed A* for the Cassini scenario was reported in [8,43].

7.1 Cassini Saturn Space Probe Scenarios

Cassini is interesting as NASA's most complex space craft to date, and hence a representative case study of a complex embedded system. The Cassini scenario consists of roughly 80 components, which correspond to 80 decision variables, with an average domain size of roughly four values. Constraints are encoded in propositional logic using approximately 3,000 propositional variables and 12,000 clauses. This results in a decision space whose size is approximately 4^{80} and a state space whose size is approximately 2^{3000}.

We compared the performance of conflict-directed A* to that of constraint-based A* (i.e., no conflict-direction) by measuring the total number of nodes expanded and the largest length of the search queue. This was performed for six failure recovery scenarios supplied by Cassini engineers. Each of these scenarios involved selecting a set of component mode changes that re-established the spacecraft's configuration goals after a failure (i.e., mode selection).

Conflict-directed A* was able to focus the search dramatically for all the test cases. Performance broke into three categories: Several of the failures involved simple recoveries, such as the inertial reference unit and accelerometer failures, whose best recovery action involved changing the mode of a single component. In these cases, conflict-directed A* found the best solution with 12 or less node expansions and a maximum queue size of 3.

Recoveries of moderate difficulty, such as the main engine overheating or a spacecraft attitude failure, required recoveries that changed up to 10 component modes. These were solved with approximately 50 node expansions and a maximum queue size of 10.

The most complex recoveries, such as a low acceleration reading, needed approximately 100 node expansions and a maximum queue size of 50. For all cases, the computational cost in terms of time and space usage is extremely modest, compared to the complexity of the search space and the number of mode changes in the solution.

Constraint-based A* performed well overall, considering the effective size of the search space, but its performance was much worse in comparison to conflict-directed A*. Also note that the performance without conflict-direction was very sensitive to variable ordering. For comparison with conflict-directed A*, we consider optimistic orderings.

For the family of simplest recoveries, constraint-based A* required at least 50 times as many node expansions as conflict-directed A*, and the increase in space usage was worse. The increase in the number of expanded nodes and queue size was a result of considering nodes that could not contribute to restoring the configuration goal.

For recoveries of moderate complexity, the performance of constraint-based A* varied considerably, consuming from 20 to over 500 times as much space and time as conflict-directed A*. This variation was the result of a large dependence on the order of the variables and values searched, and the number of mode changes in the final solution.

Recoveries of greatest complexity were the most difficult to discover. For these recoveries, A* without conflict-direction increased the number of nodes expanded by an average factor of 200 over conflict-directed A*, and increased the maximum queue size by a factor of 250.

7.2 Randomized Experiments

Problem Parameters					Constraint-Based		Conflict-Directed			Mean CD-CB Ratio	
Variables	Domain Size	Decision Variables	Constraint Clauses	Clause Length	Nodes Expanded	Queue Size	Nodes Expanded	Queue Size	Conflicts Used	Nodes Expanded	Queue Size
30	5	10	10	5	683	1230	3.33	6.33	1.2	4.5%	5.6%
30	5	10	30	5	2360	3490	8.13	17.9	3.2	2.4%	3.5%
30	5	10	50	5	4270	6260	12	41.3	2.6	0.83%	1.1%
30	10	10	10	6	3790	13400	5.75	16	1.6	2.0%	1.0%
30	10	10	30	6	1430	5130	9.71	94.4	4.2	4.6%	5.8%
30	10	10	50	6	929	4060	6	27.3	2.3	3.5%	3.9%
30	5	20	10	5	109	149	4.2	7.2	1.6	13%	13%
30	5	20	30	5	333	434	6.4	9.2	2.2	6.0%	5.4%
30	5	20	50	5	149	197	5.4	7.2	2	12%	11%

Table 1
Average performance of Constraint-based A* and Conflict-directed A* on randomly-generated problems.

Turning to randomized experiments, we verified the performance improvements discussed above through a series of experiments on randomly generated problems. For these experiments, each randomized data set was generated based on five parameters characterizing optimal CSP problems: the number of state variables, the maximum domain size of each state variable, the number of decision variables, the number of constraints, and the size of each constraint. The size of the variable domains and constraints were selected with uniform distribution between 2 and the allowed maximum. Cost for each variable assignment was selected in a similar manner.

Conflict-directed A* and constraint-based A* were each applied to the sets of randomly generated problems and rated based on total number of nodes expanded and maximum search queue length. The results of these experiments are shown in Figure 1. Once again, the data shows a significant improvement in performance for conflict-directed A* across the range of problems tested. The degree of improvement varied depending on how constrained the problem was and the difficulty of the optimization problem.

The data suggests that the performance benefit of conflict-direction for A* increases as the problems become more constrained and as the maximum domain size increases. For highly constrained problems, conflicts tend to arise with fewer assignments. This allows conflict-directed A* to rule out larger portions of the state space that would otherwise be explored.

Conflict-directed A* also performs well for problems that are lightly-constrained, because the problem contains fewer conflicts. Hence, the kernels that resolve all conflicts tend to be short, and are discovered at a very shallow point in the search. Once the kernel is found, extracting its best state involves little search. Note that the results for lightly constrained problems is less significant, simply because these problems are more easily solved in general.

7.3 Summary

To summarize, the performance of both constraint-based A* and conflict-directed A* scale well for systems of real-world complexity. The excellent performance of both approaches on the Cassini example demonstrates the effectiveness of the approach to using mutual preferential independence to guide search. In addition, the substantial and consistent increase in performance of conflict-directed A* over constraint-based A* demonstrates the effectiveness of conflict-directed search as a focussing mechanism for real-world applications. These performance results are confirmed for a broad set of randomly generated problems.

8 Conclusion

Many artificial intelligence decision making problems, such as diagnosis, planning, and embedded systems control, are being translated from CSPs to optimization problems involving a search over a discrete space for the best solution that satisfies a set of constraints. This has opened a new research frontier at the boundary between optimization and automated reasoning research.

We described this family of problems as optimal constraint satisfaction problems, that is, multi-attribute decision problems whose decision variables are constrainted by a set of finite domain constraints. We highlighted the pervasive family of optimal CSPs that are mutually, preferentially independent.

The remainder of the paper introduced conflict-directed A*, an algorithm for tackling optimal CSPS by extending A* search. Traditional A* search guarantees optimality by visiting all states whose cost is better than that of the optimal feasible solution. Conflict-directed A* is able to reason about subsets of the infeasible states implicitly, by exploiting the structure of the

CSPs and the source of conflicts.

Conflict-directed A* searches the state space in best first order, using mutual preferential independence (MPI) to construct an admissible heuristic that guides the search through the space of partial assignments. The search is accelerated by identifying the sources of conflict within each inconsistent candidate found and using this information to jump over related candidates in the search tree. This elimination process builds upon the concepts of conflict and kernel, generalized from model-based diagnosis[1,2] and dependency-directed search[3–6]. In Section 7 we saw that this approach leads to a several order of magnitude increase in performance over A* without conflicts.

At the core of conflict-directed A* is the ability to identify a feasible region of state space, called a kernel, that contains the best utility state resolving all known conflicts. The computational challenge is that an exponential number of kernels may exist in the worst case. We focus the process of generating kernels towards only the best kernel, by introducing an algorithm, called Next-Best-Kernel, that combines minimal set covering with A* search. Next-Best-Kernel guides the search and reduces node expansion by exploiting MPI similar to Constraint-based A*. In Section 7 we saw, during the Cassini and randomized experiments using Conflict-directed-A*, that Next-Best-Kernel generates a set of search nodes that is extremely modest compared to the total size of the search space.

Next-Best-Kernel also offers a powerful algorithm for candidate generation[1,17,19] that generates parsimonious descriptions of solutions in best first order. This results in an any-time, any-space algorithm that generates the most useful descriptions first, and can be terminated at any point.

This paper has focussed on the interrelationship between A* search, constraint satisfaction, and conflict-directed reasoning. These are just a few of a rich set of computationally powerful methods that have been developed over the last decade for solving constraint satisfaction problems (e.g.,[27,30,28,31,32]). The extension of these methods to Optimal CSPs is a rich area for future research.

9 Acknowledgments

This research leverages insights developed in collaboration with a range of researchers. Conflict-directed A* generalizes from ideas explored jointly with Johan de Kleer on assumption-based dependency-directed backtracking, and focussed diagnostic search algorithms. These ideas were extended, jointly with Pandu Nayak, to create a model-based reactive system, Livingstone, which was demonstrated on the NASA DS1 Probe. Finally, we would like to thank members of the MIT Model-based Embedded and Robotic Systems group, particularly Seung Chung, Paul Elliott, Michael Hofbaur and Mitch Ingham, as well as Kaijen Hsaio, for their extensive efforts at improving and demonstrating the principles of conflict-directed A*.

References

[1] J. de Kleer, B. C. Williams, Diagnosing multiple faults, Artificial Intelligence 32 (1) (1987) 97–130.

[2] J. de Kleer, A. Mackworth, R. Reiter, Characterizing diagnoses and systems, Artificial Intelligence 56 (1992) 197–222.

[3] R. Stallman, G. J. Sussman, Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, Artificial Intelligence 9 (1977) 135–196.

[4] J. Gaschnig, Performance measurement and analysis of certain search algorithms, Tech. Rep. CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA (1979).

[5] J. de Kleer, B. C. Williams, Back to backtracking: Controlling the ATMS, in: Proceedings of AAAI-86, 1986, pp. 910–917.

[6] M. Ginsberg, Dynamic backtracking, Journal of Artificial Intelligence Research 1 (1993) 25–46.

[7] J. de Kleer, B. C. Williams, Diagnosis with behavioral modes, in: Proc IJCAI-89, 1989, pp. 1324–1330.

[8] B. C. Williams, P. Nayak, A model-based approach to reactive self-configuring systems, in: Proc AAAI-96, 1996, pp. 971–978.

[9] B. C. Williams, P. Nayak, A reactive planner for a model-based executive, in: Proceedings of IJCAI-97, 1997.

[10] M. Ingham, R. Ragno, B. C. Williams, A reactive model-based programming language for robotic space explorers, in: Proceedings of ISAIRAS-01, 2001.

[11] M. R. Genesereth, The use of design descriptions in automated diagnosis, Artificial Intelligence 24 (1984) 411–436.

[12] R. Davis, Diagnostic reasoning based on structure and behavior, Artificial Intelligence 24 (1984) 347–410.

[13] A. Blum, M. Furst, Fast planning through planning graph analysis, Artificial Intelligence 90 (1-2) (1997) 281–300.

[14] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: Proceedings of AAAI-92, 1992, pp. 440–446.

[15] G. J. Sussman, A Computational Model of Skill Acquisition, North Holland, 1975.

[16] S. Minton, M. D. Johnston, A. B. Philips, P. Laird, Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems, Artificial Intelligence 58 (1992) 161–205.

[17] R. Reiter, A theory of diagnosis from first principles, Artificial Intelligence 32 (1) (1987) 57–96.

[18] P. Struss, O. Dressler, Physical negation: Integrating fault models into the general diagnostic engine, in: Proc IJCAI-89, 1989, pp. 1318–1323.

[19] W. Hamscher, L. Console, J. de Kleer, Readings in Model-Based Diagnosis, Morgan Kaufmann, San Mateo, CA, 1992.

[20] J. de Kleer, J. S. Brown, A qualitative physics based on confluences, Artificial Intelligence 24 (1984) 7–83.

[21] K. Forbus, Qualitative process theory, Artificial Intelligence 24 (1) (1984) 85–168.

[22] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, Computational Intelligence 3 (1993) 268–299.

[23] I. Gent, T. Walsh, Towards an understanding of hill-climbing procedures for sat, in: Proceedings of AAAI-93, 1993, pp. 28–33.

[24] B. Selman, H. Kautz, B. Cohen, Noise strategies for improving local search, in: Proceedings of AAAI-94, 1994, pp. 337–343.

[25] J. de Kleer, B. C. Williams, Focusing the diagnosis engine, in: Proceedings of DX-90, 1990.

[26] O. Dressler, A. Farquhar, Focusing atms-based problem solvers, Tech. Rep. Report INF-2-ARM 13, Siemens (1989).

[27] T. Schiex, H. Fargier, G. Verfaillie, Valued constraint satisfaction problems: hard and easy problems, in: Proceedings of IJCAI-95, pp. 631–637.

[28] S. Bistarelli, U. Montanari, F. Rossi, Semiring-based constraint satisfaction and optimization, Journal of the ACM, 44 (2), (1997), 201-236.

[29] P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Science and Cybernetics SSC-4 (2) (1968) 100–107.

[30] G. Verfaillie, M. Lemaitre, T. Schiex, Russian doll search for solving constraint optimization problems, in: Proceedings of AAAI-96, 1996, pp. 181–187.

[31] K. Kask, R. Dechter, Mini-bucket heuristics for improved search, in: Proceedings of UAI-99, 1999.

[32] M. Sachenbacher, B. C. Williams, On-demand bound computation for best-first constraint optimization, in: Proceedings of CP-04, 2004.

[33] N. Muscettola, P. Nayak, B. Pell, B. C. Williams, The new millennium remote agent: To boldly go where no ai system has gone before, Artificial Intelligence 100.

[34] S. Chung, J. V. Eepoel, B. C. Williams, Improving model-based mode estimation through offline compilation, in: Proceedings of ISAIRAS-01, 2001.

[35] B. C. Williams, S. Chung, V. Gupta, Mode estimation of model-based programs: Monitoring systems with complex behavior, in: Proceedings of IJCAI-01, 2001.

[36] B. C. Williams, J. Cagan, Activity analysis: Simplifying optimal design problems through qualitative partitioning, Engineering Optimization 27 (1996) 109–137.

[37] H. Li, B. C. Williams, Generalized conflict learning for hybrid discrete/linear optimization, in: Proceedings of CP-05, 2005.

[38] B. C. Williams, B. Millar, Decompositional, model-based learning and its analogy to model-based diagnosis, in: Proceedings of AAAI-98, 1998, pp. 197–203.

[39] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of a*, Journal of the Association for Computing Machinery 32 (3) (1985) 505–536.

[40] B. C. Williams, Doing time: Putting qualitative reasoning on firmer ground, in: Proceedings of AAAI-86, 1986, pp. 105 – 112.

[41] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Communications of the ACM 5 (1962) 394–397.

[42] B. C. Williams, M. Ingham, S. Chung, P. Elliott, Model-based programming of intelligent embedded systems and robotic explorers, to appear in IEEE Proceedings, Special Issue on Embedded Software .

[43] P. Nayak, B. C. Williams, Fast context switching in real-time propositional reasoning, in: Proceedings of AAAI-97, 1997, pp. 50–56.

[44] D. Mc Allester, A three-valued truth maintenance system, Tech. Rep. S.B. Thesis, Dept. of EECS, MIT (1978).

[45] J. Doyle, A truth maintenance system, Artificial Intelligence 12 (1979) 231–272.

[46] K. Forbus, J. de Kleer, Building Problem Solvers, MIT Press, 1992.

[47] J. de Kleer, An assumption-based TMS, Artificial Intelligence 28 (1) (1986) 127–162.

[48] R. Ragno, Solving optimal satisfiability problems through clause-directed A*, M. Eng. Thesis, MIT, May 2002.

A Conflict-directed A*: Requirements and Implementation

The four subprocedures within the Conflict-directed-A*(CSP, $\mathbf{y}$, g) loop are defined through the following requirements:

Requirement 1 Let $OCSP = \langle \mathbf{y}, g, \text{CSP} \rangle$ be an optimal constraint satisfaction problem, α be a decision state of OCSP, and Γ be the set of known conflicts of OCSP, then:

Consistent? Consistent?(CSP, α) is True if and only if $C_{\mathbf{y}}(\alpha)$ is consistent.
Extract-State-Conflicts Let Δ = Extract-State-Conflicts(CSP, α). Δ is empty if and only if α is consistent with $C_{\mathbf{y}}$; otherwise, each $\delta \in \Delta$ is a state conflict of α for $C_{\mathbf{y}}$.
Eliminate-Redundant-Conflicts: Let Δ = Eliminate-Redundant-Conflicts(Γ), where Γ is a set of conflicts. Then $\Delta \subset \Gamma$ and States(Δ) = States(Γ).
Next-Best-State-Resolving-Conflicts Let α = Next-Best-State-Resolving-Conflicts(OCSP). Then $\alpha = \{\}$ if no state in $S_{\mathbf{y}}$ exists that resolves conflicts Γ and that is not in *solutions*. Otherwise, α is a decision state in $S_{\mathbf{y}}$ such that α is not in *solutions*, α resolves conflicts Γ, and no state $\beta \in S_{\mathbf{y}}$ exists such that β resolves Γ and $g(\beta) < g(\alpha)$.

We do require that *Consistent?* be able to determine inconsistency as well as consistency. An inconsistency is typically found using a systematic search procedure that performs limited inference, such as back track search with forward checking or the DPLL propositional satisfiability procedure[41]. Local search methods, such as Min-Conflict [16] or GSAT[14], are efficient at determining consistency, but can not alone determine inconsistency.

Note that *Extract-State-Conflict* does not need to return a complete set of conflicts, and the conflicts are not required to be minimal, since this does not impact the correctness of the algorithm. Of course a complete set of minimal conflicts rules out the largest set of inconsistent states. However, this must be traded against the computational cost of extracting conflicts, since generating the complete set of minimal conflicts is NP Hard. *Extract-State-Conflict* must return at least one conflict when called with a decision state α that is inconsistent. This can always be performed efficiently, since α may always be returned as a conflict, for example, if no other conflict can be extracted efficiently.

The most common way to extract a conflict, as mentioned in Section 3.2, is based on local constraint propagation. Assignments α are propagated using a local inference rule, such as unit propagation, while maintaining a dependency trace of the deductions performed. When an inconsistency is derived, the dependency trace is examined to extract the subset of α that was used to derive the inconsistency. For example, Figure 5 shows the dependency traces for generating Conflicts 1 and 2, respectively. The dependencies in Figure 7 show how $O1 = G$, $O2 = G$, and $A2 = G$ were used to detect the symptom at F.

The implementation discussed in this paper uses propositional clauses as constraints. *Consistent?* is implemented using a variant of the DPLL satisfiability procedure [41] that uses Boolean Constraint Propagation (BCP) [44–46,43] to perform unit propagation incrementally. BCP maintains dependencies during propagation. *Extract-State-Conflict* uses these dependencies to quickly extract a single conflict when an inconsistency is found. A range of alternatives are possible. For example, a prime implicant algorithm, such as an ATMS[47], might be used to identify one or more subsets of α that, together with the CSP constraints, entail False. These algorithms are exponential in the worst case. It is an open question as to whether or not the benefit of discovering additional conflicts can out weight the added computational cost.

The function Eliminate-Redundant-Conflicts(Γ) eliminates conflicts that are redundant in the sense that their removal doesn't alter the set of states that manifest one or more of the conflicts in Γ. Note that there does not always exist a unique subset of Γ that is irredundant. Also note that identifying an irredundant set of conflicts is a common task studied in the circuit synthesis literature, and is not tractable in the general case. However, Eliminate-Redundant-Conflicts does not need to eliminate all redundant conflicts, since the existence of redundant conflicts does not alter the solution, only the solution time. It is an open empirical question as to whether or not redundant conflicts speed up or slow down the process. It is, however, the case that including a conflict that is a strict superset of another conflict offers no computational benefit, hence, our implementation of Eliminate-Redundant-Conflicts simply eliminates these superset conflicts.

B Search Trees for Optimal CSPs

Below are functions for constructing and examining a search tree for a CSP, called an assignment tree, introduced in Section 4. These functions are used by Constraint-Based-A* (Figure 12), Next-Best-Kernel (Figure 18) and Next-Best-State-Resolving-Conflicts of Conflict-Directed-A* (Figure 22).

function Make-Tree-Node(*assignment, parent*)
  return $\langle$*assignment, parent*$\rangle$

function Root?[*node*]
  returns True if *node* is the root of the search tree.
  if Assignment[*node*] = {}
    then return True

else return False

function State[*node*]
returns the (partial) assignments along the path from root to node.
if Root?[*node*]
then return {}
else return Assignment[*node*] $\cup$ State[Parent[*node*]]
end

function Theta[*Problem*]
returns the initial state of the search, which is the empty assignment.
return {}

C Heuristic Cost of an Optimal CSP

The following are function definitions for cost g and heuristic cost h for an optimal CSP. These functions are used by Constraint-Based-A* (Figure 12), Next-Best-Kernel (Figure 18) and Next-Best-State-Resolving-Conflicts of Conflict-Directed-A* (Figure 22).

function g[*problem*](*node*)
returns the cost of *node*'s state.
if Root?[*node*]
then return I_G[*problem*]
else $\{y_i = v_j\}$ = Assignment[*node*]
return G[*problem*]($g_i(v_j)$, **g**[*problem*](Parent[*node*]))

function h[*problem*](*node*)
returns the best cost to complete *node*'s state.
unassigned $\leftarrow$ all variables not assigned in State[*node*]
return G_{min}[*problem*](*unassigned*)

function G_{min}[*problem*](*variables*)
returns the minimum cost of all assignments to *variables*.
if *variables* = {}
then return I_G[*problem*]
else y_i = one of *variables*
remaining = *variables* - $\{y_i\}$
return G[*problem*](g_{min}[*problem*](y_i), G_{min}[*problem*](*remaining*))

function g_{min}[*problem*](y_i)
returns the minimum attribute cost for y_i.
return $\min_{v_{ij} \in D_i[problem]} g_i[problem](v_{ij})$

D Constituent Kernels

The procedure for generating Constituent-Kernels of a set of conflicts, Γ, is provided below, and directly follows from Proposition 2. Its worst case computational cost is negligible, on the order of $\sum_{D_{x_i} \in \mathbf{D_x}} |D_{x_i}|$. Constituent-Kernels is used by functions Next-Best-Kernel (Figure 18) and Next-Best-State-Resolving-Conflicts (Figure 22).

function Constituent-Kernels(Γ)
  returns a set whose elements are sets of kernels for each conflict in Γ.
  constituent-kernels $\leftarrow \{\}$
  for γ **in** Γ
    $K_\gamma \leftarrow \{\}$
    for $(x_i = v_{ij})$ **in** γ
      $K_\gamma \leftarrow K_\gamma \cup \{\{x_i = v_{ik}\} | v_{ik} \in D_{x_i}, v_{ik} \neq v_{ij}\}$
    constituent-kernels $\leftarrow$
      Add-To-Minimal-Sets(*constituent-kernels*, K_γ)
  return *constituent-kernels*

function Add-To-Minimal-Sets(*Set, S*)
  returns Adds S to *Set* and removes any element of S that is a
    superset of another element.
  for E **in** *Set*
    if $E \subset S$
      then return *Set*
      else if $S \subset E$
        then *Set* $\leftarrow$ remove E from *Set*
    finally return *Set* $\cup \{S\}$