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

Brian C. Williams and Robert J. Ragno

Space Systems and Artificial Intelligence Laboratories
Massachusetts Institute of Technology, Rm 37-381
Cambridge, Massachusetts 02139

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].

1 Introduction

The approach of focusing search based on summaries of logical inconsistency is a venerable problem solving method within AI. These descriptions 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

Email addresses: williams@mit.edu (Brian C. Williams), rjr@mit.edu (Robert J. Ragno).

[1] Supported by DARPA Mobies program under contract F33615-00-C-1702.

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 task of generalizing conflict-directed search to handle constraint-based optimization problems 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*.*

One of the earliest systems to exploit conflicts is Hacker [14], a repair-based planner that eliminates conflicts between a set of goals and a proposed plan. Subsequently, conflicts between current and intended states have been used to focus problem solving in a broad range of applications, including circuit analysis[3], diagnosis[12,11,1,15–17], qualitative reasoning[18,19], planning, scheduling[20], constraint satisfaction[4,6] and propositional inference[21]. In these approaches a conflict takes on many forms, such as a discrepancy between a solution and a goal, a hypothesis and a set of observables, or a set of constraints that are logically inconsistent.

Methods that use conflicts to focus search fall into three basic paradigms. Systematic, backtrack search methods use conflicts extensively to select backtrack points. Examples include dependency-directed backtracking [3], intelligent backtracking, conflict-directed backjumping[22] and dynamic backtracking[6].

Conflicts have proven equally useful for guiding local search. Representative examples include Hacker[14] for planning, Min-Conflict for constraint satisfaction [20], and GSAT or WalkSat for propositional satisfiability[21,23,24]. In these approaches a local operator is selected based on the number of conflicts it eliminates.

Conflict-directed A* builds upon a third approach, which uses conflicts to solve constraint satisfaction problems using divide and conquer. We will refer to this as *conflict-directed divide and conquer (CDC).* CDC plays an integral role in the General Diagnostic Engine (GDE) [1], and has been incorporated within a range of diagnostic methods [17]. 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. CDC begins by searching in parallel for all "smallest" partial assignments that produce an inconsistency. These partial assignments are called *conflicts.* The set of conflicts are then combined to produce compact descriptions of all feasible states, called *kernel diagnoses.* The key feature of CDC 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 CDC is that many practical applications only require one or a few best solutions, rather than all solutions. In this case CDC's 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 exponentially in the worst case. Hence in the model-based diagnosis community, CDC fell increasingly to the wayside during the 90's, being replaced by methods that enumerate the state space in best first order [7,25,8].

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 in this paper by first defining a family of problems called *Optimal Constraint Satisfaction Problems* (see Section 3). 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.

We then introduce *conflict-directed A**, a method for solving Optimal CSPs that satisfy preferential independence. Like A*, 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. In practice this has lead to a dramatic decrease in the number of states visited over an A* approach we introduce, called constraint-based A*, that does not exploit conflicts.

Variants of this algorithm have been demonstrated on the control of a variety of embedded and autonomous systems, including the task of monitoring the health of a robotic astronaut, and the repair of a 100 million dollar deep space probe, 6 light minutes from earth [8,26]. Variants have also been used to perform such tasks as model compilation [9], diagnosis[1], mode estimation[7,27,28], and hardware reconfiguration and repair[8–10]. This paper summarizes how a range of these tasks have been formulated as Optimal CSPs.

This paper presents a method for incorporating conflict-directed reasoning

into a pervasive family of discrete, constrained optimization problems. In two related papers we demonstrate how conflicts can be used to help accelerate the solution of two major families of continuous optimization problems. The first paper, [29], describes a method, called *activity analysis*, that solves non-linear, constrained optimization problems by ruling out portions of the state space that are sub-optimal. These subspaces are described by the conflicts of a qualitative version of the Karush Kuhn-Tucker conditions of optimality.

The second paper, [30], describes a method, called *decompositional model-based learning*, which uses conflicts to solve maximum likelihood problems, such as parameter estimation, state estimation and model-based learning. This approach accelerates the learning process by using algebraic versions of conflicts, called *dissents*, to decompose the learning problem into a set of simpler, but overlapping sub-problems.

The remainder of this paper is structured as follows. In Section 2 we introduce the conflict-directed A* algorithm informally, first describing its role in creating model-based embedded systems that reason at reactive time-scales, and then stepping through the algorithm on a simple example called Boolean polycell. In Section 3 we define optimal constraint satisfaction (OCSP) and optimal proposition satisfiability (OpSat) problems, and introduce the property of mutual preferential independence. In Section 4 we develop an algorithm for solving Optimal CSPs, called *constraint-based A**, which leverages the property of preferential independence. In Section 5 we develop an algorithm, called Next-Best-Kernel, that 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. In Section 6 we present the *conflict-directed A** algorithm, which uses conflicts to jump over leading states that are proven inconsistent. This method unifies constraint-based A* and Next-Best-Kernel, developed in the two preceding sections. Finally, in Section 7 we discuss experimental results on the performance of constraint-based A* and conflict-directed A* applied to both randomly generated problems and space applications.

2 Conflict-directed A* Without Tears

This section provides an informal development of optimal CSPs, their solution through conflict-directed A*, and their application to model-based embedded systems.

2.1 A Pictorial View of Conflict-directed A*

A* is often the method of choice for finding optimal solutions to discrete state space search problems, particularly those framed as graph search. A* generates and tests states in decreasing order of estimated utility, typically called *heuristic cost*. This process is depicted in Figure 1.

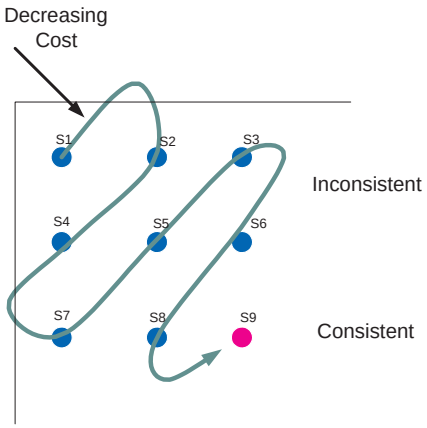

Fig. 1. A* Search examines all best cost states leading up to the first consistent state.

An *admissible* heuristic is optimistic, it never under estimates utility. For an admissible heuristic, A* has the key property that it is guaranteed to return an optimal feasible solution, if such a solution exists. A* is *efficient* in that it explores no search state with estimated utility less than the optimum.

However, to guarantee that its solution is optimal, A* visits every state whose estimated utility is greater than the true optimum. The number of states visited can be unacceptable for many practical problems, such as model-based embedded control systems that perform best first search within the reactive control loop [8,9,27,28,10].

Conflict-directed A* solves optimization problems that include a set of decision variables $\mathbf{y}$ that must be assigned values that maximize a utility function g. In addition, the assignment must satisfy a set of finite domain constraints on $\mathbf{y}$, framed as a CSP. Conflict-directed A* guides its search using descriptions of states that are *inconsistent* with the CSP, called *conflicts*. 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*.

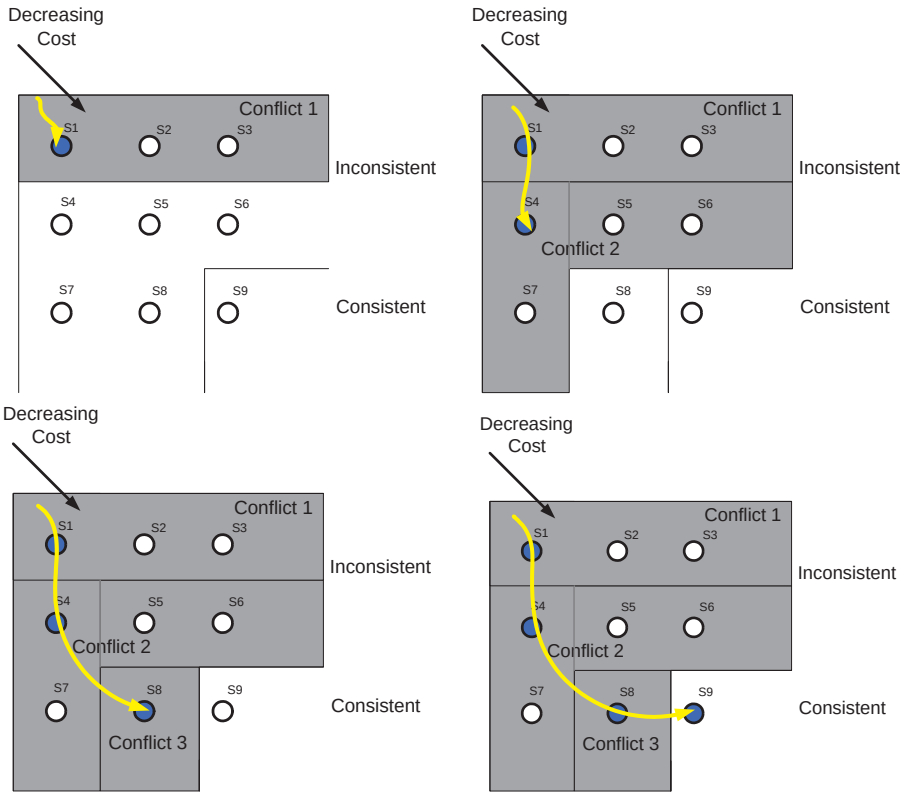

Fig. 2. Conflict-directed A* focuses search using a set of known conflicts. It jumps over the best inconsistent states, by searching only over the best cost states that resolve all known conflicts. a) - d) represent snapshots along a prototypical search. Circles represent states. Filled in circles have been tested for consistency. Regions in grey have been ruled out by conflicts. Only state $S9$ is consistent.

A pictorial example is shown in Figure 2. Conflict-directed A* first selects state $S1$, which proves inconsistent. This inconsistency generalizes to Conflict 1, which eliminates states $S1$, $S2$ and $S3$ (Figure 2a). Conflict-directed A* then tests state $S4$ as the best cost state that resolves Conflict 1. $S4$ tests inconsistent and generates Conflict 2, thus eliminating states $S4$ - $S7$ (Figure 2b). Next, conflict-directed A* 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 2c). The search finally tests state $S9$ and finds it consistent. Hence, $S9$ is an optimal solution (Figure 2d).

The result of conflict-directed A* in this pictorial example is to test three of the leading inconsistent states – $S1$, $S4$ and $S8$ – while jumping over five leading inconsistent states: $S2$, $S3$, $S5$, $S6$ and $S7$. The saving has proven much more dramatic in real world examples. 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 [31]. The reconfigura-

tion 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).

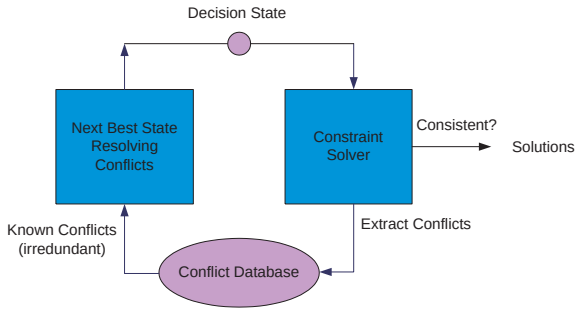

Fig. 3. Block diagram of Conflict-directed A* as generate and test. The generator produces candidates that resolve the current conflicts in best first order. Candidates are tested for constraint satisfiability using any CSP solver that is able to extract and return conflicts.

Figure 3 offers an alternative view of conflict-directed A* as interleaved best first generate and test. The generator is updated with the currently known conflicts, and returns, as a candidate, the best valued decision state that resolves all conflicts. The candidate is tested using a CSP algorithm that is able to determine satisfiability and is able to extract one or more conflicts from the candidate when the candidate proves inconsistent. It is important to note that conflict-directed A* makes minimal committments to the form of the CSP solved and the CSP algorithm applied. It simply requires that the decision variables range over a finite domain and that the CSP algorithm has the ability to extract conflicts. Hence conflict-directed A* makes it easy to augment a range of CSP solvers to solve Optimal CSP problems.

2.2 Enabling Model-based Embedded Systems

Conflict-directed A* is at that core of our approach to creating a new generation of model-based 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 and building control systems, have dramatically increased their use of computation to achieve unprecedented levels of robustness, with little human support. An extreme example of this class of embedded systems is a fleet of intelligent space probes that autonomously

explore the nooks and crannies of the solar system. These systems must operate robustly for years with minimum attention. To accomplish this an embedded system may need to radically reconfigure itself in response to failures, and then navigate around these failures during its remaining operation.

The space of potential failures an embedded system may be faced with over its lifespan is far too large for a programmer to successfuly pre-enumerate. Hence the embedded system needs to automatically diagnose and plan courses of action for itself. We accomplish this by creating increasingly powerful, model-based reactive systems[8–10,27,32] that perform significant deduction within their sense/response loop. At the core of each of our model-based reactive systems is OpSat, an algorithm that quickly deduces optimal responses. OpSat is a unification of our work on online propositional inference [33] and the conflict-directed A* algorithm presented in this paper.

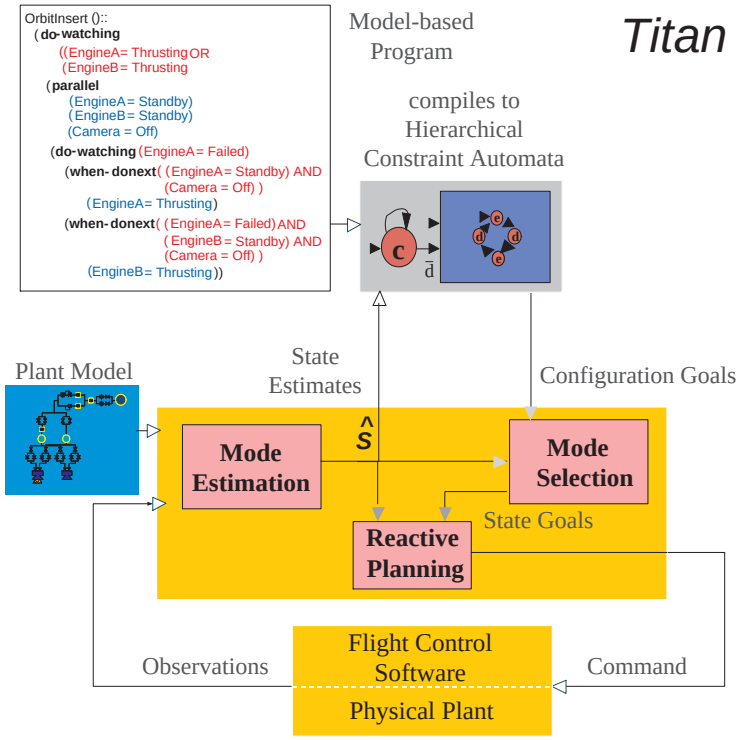

Fig. 4. Model-based Programming and Execution.

We view a model-based embedded system as consisting of a *strategic component* coupled to a low-level *reactive control system*. The strategic component provides high-level guidance by describing the desired evolutions of the system's variables, such as engine thrust, in terms of a sequence of *configuration goals*.

In the Remote Agent system [26], flown on the NASA Deep Space One probe,

these state evolutions are deduced by a temporal planner that operates on models of operational constraints. In our Titan system [10], being developed for the Air Force Tech Sat 21 multi-spacecraft mission [34], the programmer specifies these state evolutions by writing an embedded program, called a *model-based program*, that is able to read and write hidden state variables of the physical system (top of Figure 4). In the Kirk system [32] we take a middle ground, allowing the programmer to specify model-based programs containing alternatives that are chosen at runtime using a fast variant of temporal planning.

The reactive control system automatically moves along a trajectory that achieves the configuration goals. Reactive control is achieved using a *model-based executive* that generates a sequence of control actions based on knowledge of the current state, current configuration goals and a model of the physical system being controlled (bottom of Figure 4). A control action is a control variable assignment, for example, corresponding to closing a switch or sending a reset message across a bus. The current state is (partially) observable through a set of variables corresponding to sensors.

A model-based executive continually tries to transition the system towards a lowest cost state that satisfies the desired goals. When the physical system strays from the specified goals due to failures, the executive analyzes sensor data to identify the current state of the system, and then moves the system to a new state which, once again, achieves the desired goals. The executive is reactive in the sense that it reacts immediately to changes in goals and to failures, *i.e.*, each control action is incrementally generated using the new observations and configuration goals provided in each state.

As an example consider the problem of controlling the Cassini spacecraft as it inserts itself into Saturn's orbit. The executive is responsible for executing the configuration goals generated by the model-based program, shown in the upper left corner of Figure 4. One configuration goal to be achieved during this maneuver is to thrust Engine A. A series of idealized schematics of the main engine subsystem of Cassini are shown in Figure 5. 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.

To start, both engines are shut down (Figure 5 a). The model-based program first requests the left main engine to produce thrust. The model-based executive deduces that this may be accomplished by opening the latch valves leading to it (Figure 5 b). Suppose now that engine A fails to provide the de-

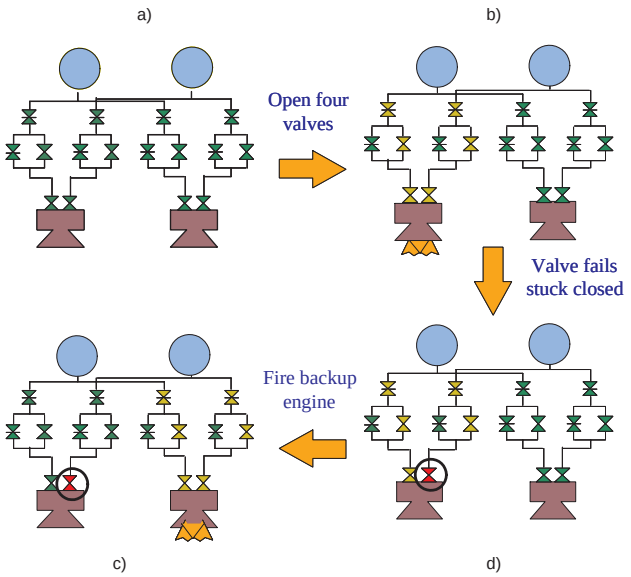

Fig. 5. Diagnosis and repair sequence for a simplified Cassini spacecraft. Pyro valves have horizontal bars through them. A valve is closed if filled in, otherwise, it is open. The faulty valve is circled.

sired thrust. The executive diagnoses the cause of failure, e.g., that the right latch valve going into engine A is stuck closed (Figure 5 c). The executive then searches for an alternate set of component modes that achieve the goal of thrusting engine A. This search fails because of the stuck valve, hence the program advances to the configuration goal of thrusting engine B. The executive decides 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 5 d)).

Using a model of the physical plant, a model-based executive determines the desired control sequence in three stages—*mode estimation* (ME), *mode selection* (MS) and *reactive planning* (RP). ME and MS setup the planning problem, identifying initial and target states, while RP reactively generates a plan solution. More specifically, ME incrementally generates the set of most likely plant trajectories consistent with the plant transition model and the sequence of observations and control actions. This is maintained as a set of most likely current states. MS uses a plant transition model and the most likely current state generated by ME to determine a reachable target state that satisfies the goal configuration. RP then generates the first action in a control sequence for moving from the most likely current state to the target state. After that action is performed ME confirms that the intended next state is achieved. ME is discussed in [8,28,27] while MS and RP are discussed in [9].

OpSat and its underlying conflict-directed A* algorithm play a central role in

implementing each of the components – mode estimation, mode selection and reactive planning. OpSat solves optimization problems of the form

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

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, a most likely set of component mode transitions that are consistent with the plant model and current observations. As discussed in [8], ME is framed as an OpSat problem, roughly of the form

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

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

Mode selection chooses, at each time step, 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]. MS is framed as an OpSat problem, roughly of the form

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

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.

Finally, the reactive planner (RP) 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* and OpSat play a central role in developing model-based embedded systems, both during runtime, through state estimation and control, and at design time, through model compilation.

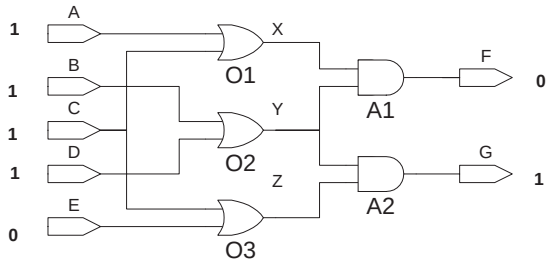

Fig. 6. The Boolean polycell example consists of three OR gates and two AND gates. Input and output values are observed as indicated.

Model-based embedded systems demonstrate a powerful application of Optimal CSPs and conflict-directed A*. However, Optimal CSPS and conflict-directed A* are best understood using a simple, pedagogical example. During the remainder of this paper we focus on the diagnosis of a boolean variant of the venerable polycell circuit, which was first introduced in [11]. Our Boolean version of polycell consists of three OR gates and two AND gates, as shown in figure 6. The assignment of values to the input variables, A, B, C, D and E, and output variables, F and G, are observed as indicated on the figure. We denote this assignment OBS.

We frame this as a multiple fault diagnosis problem similar to that of [1]. We assume that each component is in one of two possible *modes,* good or broken. If a component is good (denoted G) then it correctly performs its boolean function. If a component is broken (denoted U) then no assumption is made about its behavior. This "Unknown mode" captures the concept of constraint suspension, introduced by Davis[12].

Using the terminology of model-based diagnosis, an assignment of modes to components is called a *candidate.* A *diagnosis* is a candidate that is consistent with a model of Boolean polycell and the set of observations. For example, given the observations of figure 6, a diagnosis for Boolean polycell is

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

This diagnosis is shown in Figure 7.

For most diagnosis problems a large percentage of the candidates are diagnoses. For example, Boolean polycell has 27 diagnoses out of 32 possible candidates. The reason for this is that the Unknown mode imposes no constraint. Thus a candidate with a number of U's often results in an under constrained

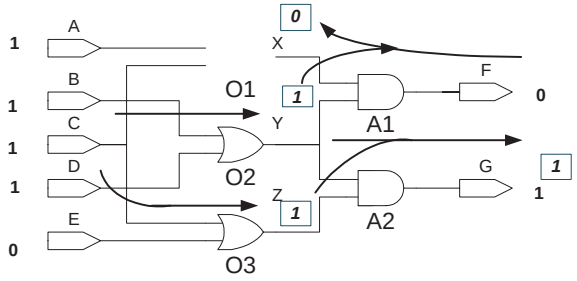

Fig. 7. Boolean polycell for the consistent diagnosis $\{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\}$. The unknown mode for O1 is depicted by removing component O1 from the figure. Values predicted by the model are placed in boxes, and dependencies between predictions are indicated by arrows.

model that is easily satisfied. Consequently, the large set of diagnoses results in poor diagnostic discrimination and computational intractability.

We have addressed this problem in the past [7,1] by shifting to a Bayesian approach that ranks diagnoses based on probability and identifies only the most likely diagnoses. The key consequence of this shift is to turn diagnosis from a constraint satisfaction problem into a constrained optimization problem.

Systems like GDE and Sherlock [7,1] rank diagnoses based on the probability of the candidate conditioned on the set of observations. They assume that failures are independent and use the model to estimate the probability of an observation, given a candidate. For simplicity of presentation we rank diagnoses based on prior probability and assume that component failures are independent. That is,

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

where P_i is a probability mass function denoting the *a priori* probability that component i is in each of its modes. Note that any realistic implementation of a diagnostic engine based on conflict-directed A* will readjust these priors to posterior probabilities, for example, as outlined in [7].

For this example we assume that OR gates have a 1% probability of failure and that AND gates have a .5% probability of failure. That is, $P(O_n = G) = .99$, $P(O_n = \mathbf{U}) = .01$, $P(A_n = G) = .995$ and $P(A_n = \mathbf{U}) = .005$. These probabilities have the property that OR gates are more likely to fail than AND gates. Components are two orders of magnitude more likely to work than to fail. All single faults are more likely than any double fault, all double faults are more likely than triple faults, and so forth. In addition, given the choice of two mode assignments with equal probability, we always select the

assignment with the component that has the higher number. For example, given a choice between equally probable assigments $O1 = G$ and $O2 = G$, we select $O2 = G$ first. This is for consistency of presentation.

The diagnosis of Boolean polycell is an instance of an optimal CSP. An optimal CSP consists of a set of decision variables, $\mathbf{y}$, ranging over a finite domain of values, $D_{\mathbf{y}}$, a utility function on the decision variables, $g : \mathbf{y} \rightarrow \Re$, and a set of constraints that the decision variables must satisfy, $C_{\mathbf{y}} : \mathbf{y} \rightarrow \{True, False\}$. For Boolean polycell the decision variables are components, its domain is the set of candidate component modes, the utility function is mode probability $P(\mathbf{y})$, and the constraint is that the solution must be consistent with the component model Φ and observations OBS.

The solution to the optimal CSP is the set of n leading candidates, ranked by decreasing g, that satisfy the constraints. For Boolean polycell the leading solutions in decreasing order of likelihood are

$$\begin{aligned}
&\text{Diagnosis 1:} && \{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\},\\
&\text{Diagnosis 2:} && \{O1 = G, O2 = G, O3 = G, A1 = \mathbf{U}, A2 = G\}, \text{ and}\\
&\text{Diagnosis 3:} && \{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = \mathbf{U}\}.
\end{aligned}$$

These correspond, respectively to only O1 being broken, only $A1$ being broken, and both $O2$ and $A2$ being broken.

2.4 Conflict-directed A Jumps Over Conflicts*

Next we use Boolean polycell to demonstrate how conflict-directed A* *jumps over most of the leading candidates that are inconsistent,* while guaranteeing optimality. As a point of reference, first suppose we generate the leading diagnoses by generating each candidate in best first order, and by testing its consistency against the model Φ and observations OBS. Generating the three single and double fault polycell diagnoses, given above, would require testing roughly $5^2 = 25$ candidates, triple faults would take $5^3 = 125$, and n faults would take $O(5^n)$. This cost is prohibitive for many systems, such as those that solve Optimal CSPs within the sense/act loop of an embedded system. In contrast, conflict-directed A* is able to find all single and double faults while only visiting *two* inconsistent states.

The top-level procedure for conflict-directed A* is shown in Figure 8. Conflict-directed A* searches candidates in decreasing order of utility. It tests each candidate S for consistency against the CSP using function Consistent?. When S tests inconsistent, Extract-Conflicts generalizes the inconsistency to one or

```
function Conflict-directed-A*(OCSP)
  returns the leading minimal cost solutions.
  Conflicts[OCSP] ← {}
  OCSP ← Initialize-Best-Kernels(OCSP)
  Solutions[OCSP] ← {}
  loop do
     decision-state ← 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 ← Extract-Conflicts(CSP[OCSP], decision-state)
     Conflicts[OCSP] ←
       Eliminate-Redundant-Conflicts(Conflicts[OCSP] ∪ new-conflicts)
  end
```

Fig. 8. Top level loop of Conflict-directed A*.

more conflicts, each denoting states that are inconsistent in a manner similar to S. Conflict-directed A* keeps track of all discovered conflicts as a record of known inconsistent states, while using Eliminate-Redundant-Conflicts to remove conflicts that offer no additional information. Conflict-directed A* then uses Next-Best-State-Resolving-Conflicts to jump over these states, and generates the next best state S' that *resolves all conflicts discovered thus far.* This process repeats until the desired set of leading solutions is found, as determined by Terminate?, or all states are eliminated. We next walk demonstrate the execution of the outer loop of conflict-directed A* for Boolean polycell.

2.4.0.1 First Candidate – All Components Okay: When conflict-directed A* starts, no conflicts have been discovered, hence all states are under consideration. Conflict-directed A* generates the most likely candidate,

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

which specifies that all components are working correctly. This candidate has probability $.99 \times .99 \times .99 \times .995 \times .995 = .961$.

Next Candidate 1 is tested for consistency against the model and observations (Figure 9). Constraint solvers typically identify inconsistencies through a combination of backtrack search and constraint propagation. For example, the DPLL algorithm uses search and unit propagation to test the satisfiability

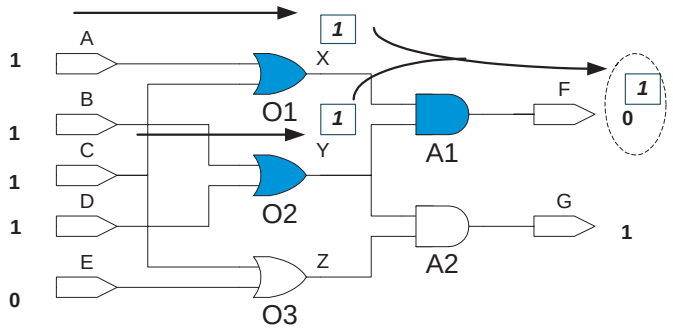

Fig. 9. Candidate 1, $\{O1 = G, O2 = G, O3 = G, A1 = G, A2 = G\}$, tests inconsistent. Predicted values are placed in boxes, and the discrepancy between predicted and observed values for F is circled. Dependencies between predicted values are denoted with arrows. Conflict-directed A* extracts Conflict 1, $\{O1 = G, O2 = G, A1 = G\}$. The components in the conflict are highlighted, and are extracted from the indicated dependencies.

of propositional constraints[35]. More specifically, given that $O1$ and $O2$ are good, propagation would conclude 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 the observation $F = 0$, hence Candidate 1 is eliminated as a solution. The results of these propagations are shown in Figure 9, including the values predicted and dependencies recorded between these predictions.

Next this inconsistency is generalized to a *conflict*. A *conflict of a candidate* is a subset of the candidate's assignments that is sufficient to produce an inconsistency with the constraints. A conflict is extracted from Candidate 1 using *reductio ad absurdum*. In particular, we concluded above from $O1 = G$, $O2 = G$ and $A1 = G$, that $F = 1$, which conflicts with observation $F = 0$. Hence our first conflict is the partial assignment

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

This conflict may be extracted by tracing backwards through the dependencies that were generated by propagation, and accumulating the assignments to $\mathbf{y}$ (the mode assignments) that appear along the path.

2.4.0.2 Jump to Second Candidate – OR Gate O2 broken: Next, conflict-directed A* jumps over any leading candidates that contain a known conflict as a subset, and jumps down to the next best candidate that resolves all known conflicts. A candidate is said to *resolve a conflict* if it does not

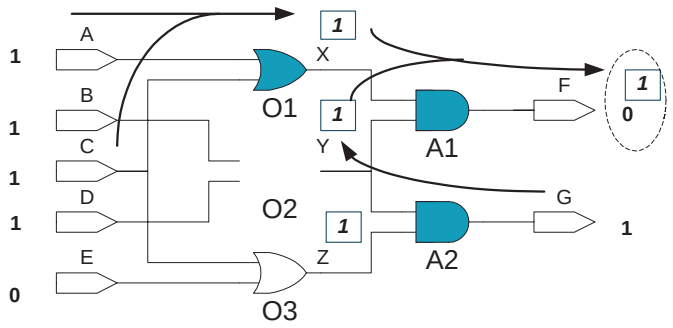

Fig. 10. Candidate 2, $\{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = G\}$, tests inconsistent. Conflict-directed A* extracts Conflict 2, $\{O1 = G, A1 = G, A2 = G\}$, which is the set of mode assignments used to detect the symptom. This figure is annotated similar to Figure 9.

contain that 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. To this end, conflict-directed A* jumps over state

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

since it contains Conflict 1 as a subset. It generates the next best state as,

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

with probability $.99 \times .01 \times .99 \times .995 \times .995 = .0097$.

This candidate resolves

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

by changing $O2 = G$ to $O2 = \mathbf{U}$. We discuss the process of generating candidates from conflicts in the next subsection.

Conflict-directed A* then tests Candidate 2 against the constraints, proving it inconsistent (see Figure 10). The constraint solver uses input $A = 1$ and $O1 = G$ to conclude that X is 1. In addition, output $G = 1$ and $A2 = G$ are used to conclude that A2's input, Y, is 1. Finally, $A = 1$ and $A1 = G$ are used to conclude that output F is 1, which is inconsistent with observation $F = 0$. This eliminates Candidate 2 and produces the conflict,

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

The predicted values and dependencies are shown in Figure 10.

2.4.0.3 The Third Candidate is a Diagnosis – OR Gate O1 broken: As the third candidate, conflict-directed A* selects the next consecutive candidate,

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

which has probability .99 × .01 × .99 × .995 × .995 = .0097, the same as Candidate 2.

This candidate resolves both Conflict 1 and Conflict 2,

$$\text{Conflict 1: } \{\boxed{\text{O1 = G}}, O2 = G, A1 = G\}$$
$$\text{Conflict 2: } \{\boxed{\text{O1 = G}}, A1 = G, O2 = G\},$$

by changing assignment $O1 = G$ to $O1 = \mathbf{U}$. Candidate 3 is tested, as depicted earlier in Figure 7, and proves consistent. Hence, Candidate 3 is our best diagnosis. No conflict is extracted.

2.4.0.4 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 particular, 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*. The next two candidates generated by conflict-directed A* are the two remaining diagnoses listed at the end of Section 2.3,

$$\text{Diagnosis 2: } \{O1 = G, O2 = G, O3 = G, A1 = \mathbf{U}, A2 = G\} \text{ and}$$
$$\text{Diagnosis 3: } \{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = \mathbf{U}\}.$$

These two diagnoses are depicted in Figure 11.

To summarize, the three leading diagnoses were generated by jumping over 19 inconsistent candidates and by explicitly considering only two inconsistent

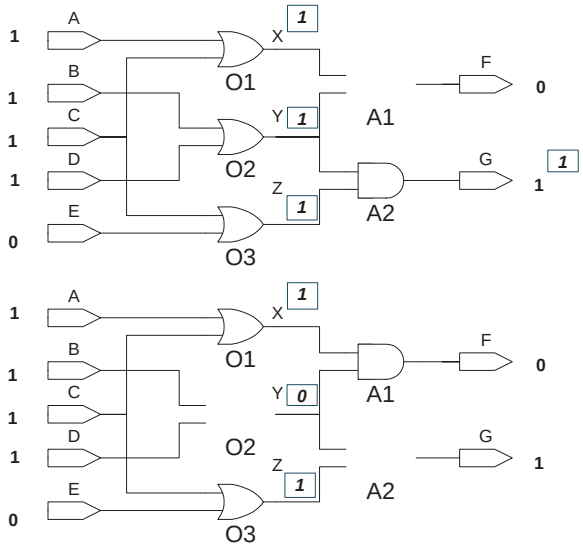

Fig. 11. Candidates 4 and 5 correspond to Diagnoses 2 and 3, respectively. a) Diagnosis 2: $\{O1 = G, O2 = G, O3 = G, A1 = \mathbf{U}, A2 = G\}$. b) Diagnosis 3: $\{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = \mathbf{U}\}$. No conflicts are detected. Unknown modes are depicted by removing corresponding components, and predicted values are placed in boxes.

candidates. If we measure *search efficiency* as

$$\frac{\text{Solutions Found}}{\text{Candidates Tested}}$$

then traditional A* has efficiency $\frac{3}{21} = 14\%$, while conflict-directed A* has efficiency $\frac{3}{5} = 60\%$. Even for this simple example the improvement is dramatic.

2.5 Generating the Best Non-Conflicting Candidate

The key to conflict-directed A* is the ability to efficiently generate, at each iteration, the next best candidate that resolves all known conflicts. This involves combining conflict-directed divide and conquer, described in the introduction, together with A* search. We provide the intuition behind this generation process in this section, offering a more rigorous development in Sections 4, 5 and 6.

The first idea behind candidate generation is to map known conflicts, describing inconsistent states, to partial assignments, called *kernels*. Each kernel

describes a set of states that resolve the known conflicts.[2] The best cost state is extracted from these kernels. Our mapping from conflicts to kernels is closely related to the candidate generation algorithm introduced within the General Diagnostic Engine [1].

For example, recall that we discovered two conflicts for Boolean polycell. A kernel for these two conflicts is

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

Assignment $O1 = \mathbf{U}$ is sufficient to resolve both Conflict 1 and 2, since it eliminates the possibility for $O1 = G$, which appears in both conflicts,

Conflict 1: $\{\boxed{\text{O1} = \text{G}}, O2 = G, A1 = G\}$ and
Conflict 2: $\{\boxed{\text{O1} = \text{G}}, A1 = G, A2 = G\}$.

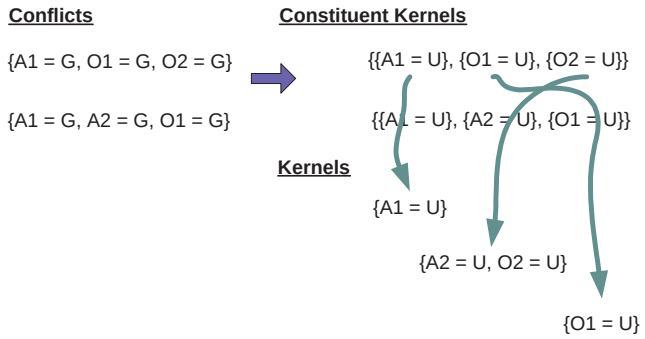

Fig. 12. Conflict-directed A* generates the set of kernels that resolve all conflicts. Each conflict is first mapped to a set of constituent kernels, which resolve that conflict individually. Kernels are generated from the constituents by minimal set covering.

We generate kernels through divide and conquer. 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. Each combined kernel has the property that it contains a constituent kernel for every conflict, hence all conflicts are resolved. For example, the constituent kernels for each of Conflict 1 and 2 are shown at the top of figure 12. The three kernels resolving both conflicts are generated by minimal set covering, and are shown at the bottom of the figure.

[2] The concept of kernel is generalized from kernel diagnosis, introduced in [2].

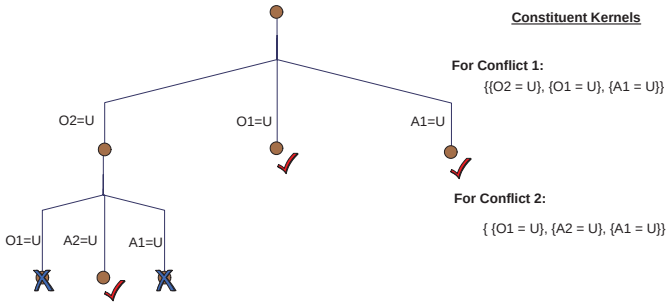

Fig. 13. A search tree created by Conflict-directed A* to identify all kernels. Nodes crossed off are visited nodes that are not kernels, either because they contain too many assignments or inconsistent assignments. Nodes that cover the kernels of both conflicts are check marked. Note that the two nodes to the right are not extended after covering Conflict 1, since their one assignment is sufficient to cover Conflict 2.

The problem with the approach of generating all kernels is that an exponential number of kernels may be involved. However, we are only interested in the kernel containing the best utility state. Hence, as we search for the kernel containing the best utility state, we want to explicitly enumerate as few kernels as possible. To accomplish this we frame kernel generation as best-first search, using A* to focus the search towards the kernel that contains the best assignment. This search combines minimal set covering with a constraint-based variant of A* search that we will introduce in Section 4.

A search tree for Boolean polycell is shown in Figure 13. The leaves of the tree are kernels which represent minimal coverings of the constituent kernels, and intermediate nodes represent partial coverings. For example, the bottom left leaf denotes kernel $\{O1 = \mathbf{U}, O2 = \mathbf{U}\}$ and its parent denotes $\{O2 = \mathbf{U}\}$. A search tree node is expanded by selecting the constituent kernels of a conflict that has not already been resolved by that node, and then creating a child node 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 node in Figure 13 are the constituent kernels of Conflict 1, which are $\{O2 = \mathbf{U}\}$, $\{O1 = \mathbf{U}\}$ and $\{A1 = \mathbf{U}\}$. The first and third leaves on the bottom left are eliminated as non-minimal, since they are supersets of the kernels $\{O1 = \mathbf{U}\}$ and $\{A1 = \mathbf{U}\}$, respectively.

Next consider the portion of the Boolean polycell search tree expanded after the generation of each candidate, Candidates 1, 2 and 3. Details of the algorithm that guides this search is developed in Section 6.

Figure 14 shows the tree when Candidate 1 is generated. At this point there are no conflicts and one kernel, {}, indicated by an arrow. We generate a candidate by assigning the remaining unassigned variables. To accomplish

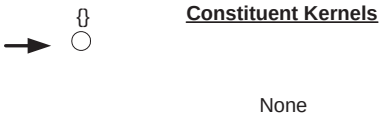

Fig. 14. Conflict-directed A* tree expansion corresponding to Candidate 1. No conflicts exist, hence the root node, denoting the empty assignment, resolves all conflicts. An open circle indicates an unexpanded node.

this we exploit a property called *mutual, preferential independence (MPI).* MPI says that to find the best candidate we may assign each variable the value with the best utility in its domain, independent of the values assigned to the other variables. This property is developed in depth in Section 3.2. The best candidate contained by the kernel $\{\}$ assigns the most likely value, G, to each of the unassigned variables, resulting in a combined probability of .961. Hence Candidate 1 has all components working,

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

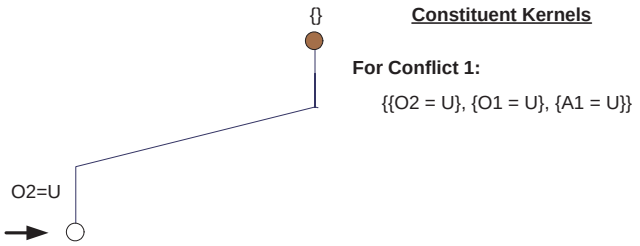

Fig. 15. Conflict-directed A* tree expansion corresponding to Candidate 2. The arrow indicates the node that is the kernel of Candidate 2. A closed/open circle indicates an expanded/unexpanded node. Note that only the best valued child of the root is expanded, not all children.

Figure 15 shows the tree when Candidate 2 is generated. At this point only Conflict 1 has been discovered, hence the set of kernels correspond to the constituent kernels of Conflict 1. An arrow indicates the kernel containing the most likely candidate, which is $\{O2 = \mathbf{U}\}$. The probability of $\{O2 = \mathbf{U}\}$ is .01 and the estimated best probability for the unassigned variables is $.99 \times .99 \times .995 \times .995 = .97$, resulting in a combined probability of .0097 for the best candidate of the kernel. Note that this tree only expands the best valued child of $\{\}$, which is $\{O2 = \mathbf{U}\}$. This is 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 $\{O1 = \mathbf{U}\}$. $\{O2 = \mathbf{U}\}$ is a kernel, since it resolves all known conflicts. The best state that expands this

kernel has only $O2$ broken; the remaining components are assigned G.

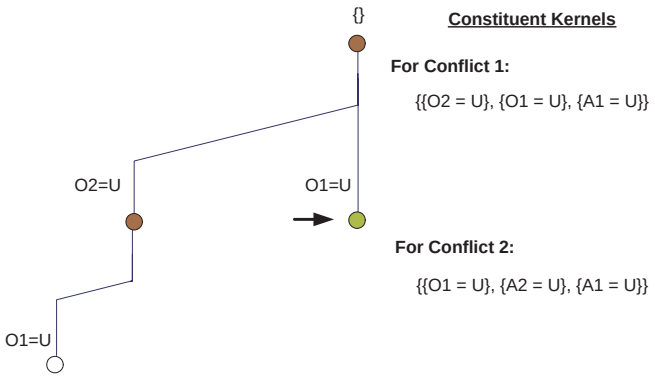

Fig. 16. Conflict-directed A* tree expansion corresponding to Candidate 3. The arrow indicates the node that is the kernel of Candidate 3. A closed/open circle indicates an expanded/unexpanded node. Note that when node $O2 = \mathbf{U}$ is expanded, its best child and its next best sibling are created.

Figure 16 shows the tree when Candidate 3 is generated. At this point Conflict 1 and 2 have been discovered. The node $\{O2 = \mathbf{U}\}$ does not resolve Conflict 2, hence, it must be expanded. We do this by creating the best child of $\{O2 = \mathbf{U}\}$, which is $\{O2 = \mathbf{U}, O1 = \mathbf{U}\}$. The probability for the best state of this kernel is $.01 \times .01 \times .99 \times .995 \times .995 = .00098$.

In addition, now that Conflict 2 has restricted the set of candidates considered under $\{O2 = \mathbf{U}\}$, we no longer are guaranteed that the node $\{O2 = \mathbf{U}\}$ contains a state that is as good as its siblings. Hence we also expand its next best sibling, which is $\{O1 = \mathbf{U}\}$. The probability of the best candidate under $\{O1 = \mathbf{U}\}$ is $.01 \times .99 \times .99 \times .995 \times .995 = .0097$. This has a higher probability than $\{O2 = \mathbf{U}, O1 = \mathbf{U}\}$, which has probability .00098. This makes intuitive sense, since single faults are more likely than double faults. Hence we choose $\{O1 = \mathbf{U}\}$ for further exploration. Next, note that node $\{O1 = \mathbf{U}\}$ resolves both Conflict 1 and 2, hence it is a kernel. We generate the best candidate of this kernel by assigning G to the remaining variables, resulting in the single fault candidate,

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

This candidate proves consistent, completing the search for the most likely diagnosis.

Note that the distinctive pattern of this search strategy is to expand 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. This strategy is embodied in the function Next-Best-Kernel, developed in Section 5.

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. 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.

In addition to its role within conflict-directed A*, constraint-based A* offers a point of comparison, by providing a method for solving optimal CSPs that exploits preferential independence, but not conflicts. Next-Best-Kernel also 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.

3 Optimal Constraint Satisfaction Problems

This paper is about the solution of optimal CSPs, that is, combinatorial optimization problems with constraints expressed as CSPs. An example is the model-based diagnosis problem framed in the previous section. We focus on the pervasive family of optimal CSPs in which the utilities of the decision variables are *preferentially independent,* as defined in this section. In addition, in Section 2.2 we described practical applications of optimal CSPs in which the constraints are encoded as propositional satisfiability and unsatisfiability constraints. We refer to these as *OpSat problems*. The concepts of optimal CSP, preferential independence and OpSat were introduced in Section 2. In this section we provide definitions, illustrated with Boolean polycell.

3.1 Defining Optimal CSPs

We begin with the familiar definition of a CSP,

Definition 1 We denote a *constraint satisfaction problem* as a triple $\mathrm{P} = \langle \mathbf{x}, \mathbf{D}_{\mathbf{x}}, C_{\mathbf{x}} \rangle$. $\mathbf{x}$ is a set of *variables*, where each variable $x_i \in \mathbf{x}$ ranges over a corresponding *domain* D_{x_i} in $\mathbf{D}_{\mathbf{x}}$. A variable assignment is an expression $x_i = v_{ij}$, where $x_i \in \mathbf{x}$ is a variable and $v_{ij} \in D_{x_i}$. A *state* of P is an assignment to vector $\mathbf{x}$, represented as a set containing exactly one assignment for every

$x_i \in \mathbf{x}$. The set of all assignments to $\mathbf{x}$ is denoted $\mathbb{S}$, and is called the *state space* of P. A constraint $C_\mathbf{x}$ denotes a subset of state space $\mathbb{S}$. A state $\alpha \in \mathbb{S}$ *satisfies* $C_\mathbf{x}$ when $\alpha \in C_\mathbf{x}$, in this case α is called a *solution to* P. If α satisfies $C_\mathbf{x}$, then $C_\mathbf{x}(\alpha)$ is said to be *consistent;* otherwise, $C_\mathbf{x}(\alpha)$ is *inconsistent.*

For example, we can use a CSP to frame the problem of identifying a set of component modes *and* variable values for Boolean polycell (Figure 6) that are consistent with a model of Boolean polycell and observations OBS. Viewed as a CSP, Boolean polycell consists of state variables

$$\mathbf{x} = \{A, B, C, D, E, F, G, X, Y, Z, O1, O2, O3, A1, A2\}.$$

A - Z are Boolean variables, each with domain $\{1, 0\}$. A - E are inputs, F and G are outputs, and X - Z are hidden variables. $O1$ - $A2$ are mode variables, each with domain $\{\mathbf{U}, G\}$.

The constraint $C_\mathbf{x} \equiv \Phi \cap \text{OBS}$ denotes the set of all variable assignments that are consistent with a model for Boolean polycell, Φ, and the set of observations, OBS. Following the description of Section 2.3, Φ specifies a behavior for each component when it is in the good mode (G), and imposes no constraint when a component is in the unknown mode ($\mathbf{U}$). Φ is defined by the conjunction of five constraints:

$$O1 = G \Rightarrow X = A \text{ OR } B$$
$$A1 = G \Rightarrow F = X \text{ AND } Y$$
$$O2 = G \Rightarrow Y = B \text{ OR } D$$
$$A2 = G \Rightarrow G = Y \text{ AND } Z$$
$$O3 = G \Rightarrow Z = C \text{ OR } E$$

where "AND" and "OR" are the standard Boolean operators on 1 and 0. Finally, observations OBS is defined by the conjunction of seven constraints: $A = 3$, $E = 3$, $B = 2$, $F = 10$, $C = 2$, $G = 12$ and $D = 3$.

Next, an optimal CSP extends a CSP to include a set of decision variables and a utility function. More specifically,

Definition 2 An *optimal constraint satisfaction problem (P)* is a triple $P = \langle \mathbf{y}, g, \text{CSP} \rangle$. $\mathbf{y}$ is a set of *decision variables*, the *utility function* g maps assignments of $\mathbf{y}$ to the Reals $\Re$, and $\text{CSP} = \langle \mathbf{x}, \mathbf{D}_\mathbf{x}, C_\mathbf{x} \rangle$ is a constraint satisfaction problem, where $\mathbf{y}$ is a subset of $\mathbf{x}$.

An assignment to $\mathbf{y}$ is called a *decision state* of P, and the domain of all possible assignments to $\mathbf{y}$, denoted $\mathbf{D}_\mathbf{y}$, is called the *decision space* of P. We

use $\mathbf{z}$ to denote the state variables of $\mathbf{x}$ not contained in $\mathbf{y}$. The domain of $\mathbf{z}$ is denoted $\mathbf{D_z}$.

We define constraint $C_\mathbf{y}$ as the projection of $C_\mathbf{x}$ on to decision variables $\mathbf{y}$, such that $C_\mathbf{y}(\mathbf{y})$ is consistent if and only if $\exists \mathbf{z} \in \mathbf{D_z}.C_\mathbf{x}(\mathbf{y}, \mathbf{z})$ is consistent.

A *solution to* P is the decision state [3]

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

For example, the diagnosis of Boolean polycell is encoded as an optimal CSP, where the decision variables of the OCSP is the set of component mode variables,

$$\mathbf{y} = \langle O1, O2, O3, A1, A2 \rangle.$$

The utility function g on the decision variables is the probability of the mode assignment,

$$g = \prod_i P_i(y_i).$$

The CSP is the set of state variables $\mathbf{x}$ and constraints $C_\mathbf{x} \equiv \Phi \cap \text{OBS}$, specified earlier. $C_\mathbf{y}(\mathbf{y}) \equiv \exists \mathbf{z} \in \mathbf{D_z}.\Phi(\mathbf{y}; \mathbf{z}) \cap \text{OBS}(\mathbf{y}; \mathbf{z})$ is consistent.

The space of candidates for Boolean polycell is equivalent to the decision space of this OCSP. The three leading solutions to the OCSP are the three diagnoses given in the preceding section,

$$\begin{aligned}
&\text{Diagnosis 1:} \quad \{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\}, \\
&\text{Diagnosis 2:} \quad \{O1 = G, O2 = G, O3 = G, A1 = \mathbf{U}, A2 = G\}, \text{ and} \\
&\text{Diagnosis 3:} \quad \{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = \mathbf{U}\}.
\end{aligned}$$

The first diagnosis, $\{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\}$, for example, satisfies $C_\mathbf{y}$ with assignments $X = 0$, $Y = 1$ and $Z = 1$, as shown in Figure 7.

[3] In this paper we use the convention that g is a utility to be maximized. It is straightforward to reformulate the approach such that g is a cost to be minimized.

This section introduces a key structural property shared by most OCSPs, called *mutual preferential independence*. We view an optimal CSP as a *multi-attribute decision problem*. The attributes are the decision variables of the CSP, each decision variable y_i has an associated *attribute utility* $g_i(y_i)$, and the function g describes the *multi-attribute utility*. We further restrict ourselves to OCSPs in which the attributes are *(mutually) preferentially independent.* Informally, this means that an assignment to $\mathbf{y}$ maximizes utility, g, by maximizing the attribute utility, g_i, of each y_i separately. Preferential independence is key to how conflict-directed A* searches through candidates in best first order. The remainder of this section makes these concepts precise.

To facilitate the solution of multi-attribute decision problems, utility theory exploits common structural properties to determine the preference of one set of assignments over another. Preference corresponds to better utility:

Definition 3 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 $\text{g}(\delta_1) > g(\delta_2)$.

Mutual preferential independence is one of the most commonly occurring structural properties of practical, multi-attribute decision problems:

Definition 4 Given an optimal CSP, $\langle \mathbf{y}, g, \text{CSP} \rangle$, then variables $\mathbf{y}$ are *mutually preferentially independent* (MPI) if for any subset of the decision variables $\mathbf{w} \subset \mathbf{y}$, preferences between assignments to $\mathbf{w}$ are independent of the particular assignments to the remaining decision variables, $\mathbf{y} - \mathbf{w}$.

Conflict-directed A* is restricted to the family of optimal CSPs that are MPI.

Economist Debreu [36] showed that any decision problem that is MPI may be captured as maximizing a utility function of the form

$$g(\mathbf{y}) = \sum_i g_i(y_i).$$

Consequently, the utility of an assignment to decision variables $\mathbf{y}$ is maximized by maximizing the utility of the assignment to each decision variable y_i, separately; that is,

$$\max g(\mathbf{y}) = \max(\sum_i g_i(y_i)) = (\sum_i \max(g_i(y_i))).$$

In Sections 4.4 and 5.3 we describe how algorithms constraint-based A* and conflict-directed A* exploit this property to efficiently enumerate assignments

in best first order.

For an optimal CSP that is MPI we encode the utility g using a single attribute utility, g_i, for each decision variable y_i, and a binary function G, used to compose the single attribute utilities into a total utility. We assume that G is associative, commutative, and has identity I_G. G applied to the utility of n attributes is defined recursively in the standard manner,

$$G(u_1, u_2, \ldots u_n) = G(u_n, G(u_1, u_2, \ldots u_{n-1}))$$
$$G(u_1) = G(u_1, I_G).$$

Total utility is then

$$g(\mathbf{y}) = G(g_1(y_1), g_2(y_2), \ldots g_n(y_n)).$$

G being MPI corresponds to the property:

If $u > v$ then $G(u, w) > G(v, w)$, for all w.

To summarize,

Definition 5 The utility function, g, of an Optimal CSP that is MPI is specified as a triple $g = \langle \mathbf{g}, G, I_G \rangle$, where each attribute utility $g_i \in \mathbf{g}$ maps D_{y_i} to $\Re$, G is a binary function from $\Re \times \Re$ to $\Re$ that is associative, commutative, and mutually preferential independent, and I_G is the identity of G.

For Boolean polycell, the utility function, $g = \prod_i P_i(y_i)$, is encoded as $g = \langle \mathbf{P}, \times, 1 \rangle$. Note that $\times$ satisfies the condition of MPI for non-zero probabilities, since if $u > v$, then $u \times w > v \times w$.

3.3 OpSat

The applications of conflict-directed A* discussed in Section 2.2 involve constraint systems described in propositional state logic. We refer to these as *OpSat problems*. For propositional logic, recall that a proposition is a boolean variable that may be assigned true or false. A literal is a proposition or its negation, the former being a positive literal and the later being a negative literal. A positive literal is true if and only if its proposition is true, and a negative literal is true if and only if its proposition is false. A clause is a disjunction of literals and is satisfied if at least one of its literals is true. A theory is a set of clauses, and is satisfied exactly when all of the clauses in the set are satisfied.

Propositional state logic is a propositional logic in which every proposition is a variable/value assignment. In addition to a set of clauses, a description in propositional state logic specifies a set of variables $\mathbf{x}$ ranging over domain $\mathbf{D_x}$, which defines the allowed variable/value assignments. An additional constraint enforced by the logic is that each variable must take on exactly one value in its domain.

In Section 2.2 we saw that several OpSat problems require the decision variables to satisfy conditions of entailment, as well as consistency. We re-encode each entailment condition ($\Theta \models \phi$) as an unsatisfiability condition ($\Theta \wedge \neg\phi$ is unsatisfiable). The general form of an OpSat problem is then a set of satisfiability conditions and unsatisfiability conditions that any solution must achieve.

Definition 6 An *OpSat problem (P)* is a triple $P = \langle \mathbf{y}, g, \mathrm{SAT} \rangle$. $\mathbf{y}$ is a set of *decision variables*, the *utility function* g maps assignments of $\mathbf{y}$ to the Reals $\Re$, and $\mathrm{SAT} = \langle \mathbf{x}, \mathbf{D_x}, C_S, C_U \rangle$ specifies the (un)satisfiability constraint, where $\mathbf{y}$ is a subset of $\mathbf{x}$. C_S and C_U are both conjunctions of clauses in propositional state logic with variables $\mathbf{x}$, ranging over domain $\mathbf{D_x}$.

A *solution to* P is the decision state

$$\begin{aligned} \delta = & \mathit{argmax}_{\mathbf{v} \in \mathbf{D_y}} g(\mathbf{v}) \\ & \text{such that } C_S(\mathbf{v}) \text{ is satisfiable and} \\ & \qquad C_U(\mathbf{v}) \text{ is unsatisfiable.} \end{aligned}$$

For example, for Boolean polycell the assignment to the mode variables must be consistent with the constraints describing the model and observations. These constraints are easily expressed in propositional state logic. Each AND gate (AND_n) for Boolean polycell is described by the conjunction of three clauses

$$\begin{aligned} AND_n = U \vee In_1 = 0 \vee In_3 = 0 \vee Out = 1 \\ AND_n = U \vee Out = 0 \vee In_1 = 1 \\ AND_n = U \vee Out = 0 \vee In_2 = 1, \end{aligned}$$

and each OR gate is described by the conjunction of clauses

$$\begin{aligned} OR_n = U \vee In_1 = 1 \vee In_3 = 1 \vee Out = 0 \\ OR_n = U \vee Out = 1 \vee In_1 = 0 \\ OR_n = U \vee Out = 1 \vee In_2 = 0. \end{aligned}$$

The primary subtlety around extending conflict-directed A* to OpSat problems is the definition and efficient extraction of conflicts for the unsat condition. Intuitively, a conflict for the unsat condition, C_U, is a minimal partial assignment to $\mathbf{y}$ that guarantees that C_U is satisfiable, for all possible assignments to the remaining variables in $\mathbf{y}$. In the remainder of this paper we focus on the general problem of solving Optimal CSPs, rather than those aspects that are peculiar to implementing OpSat in particular.

4 Constraint-based A*

In this section we explore the use of mutual preferential independence (MPI) to solve optimal CSPs. Constraint-based A* uses MPI to develop a heuristic function for A* search that guides its exploration through the space of consistent partial assignments. MPI also allows conflict-directed A* to expand a smaller portion of the search tree than traditionally performed by A* search. In subsequent sections we explore the use of conflicts.

4.1 Review of A Search*

We start by reviewing A* for state space search problems.

Definition 7 A *state space search problem* is defined by a quintuple

$\langle \Sigma, \Theta, \text{Ops}, \text{Goal-Test}, g \rangle$. Σ is the *state space* of the search problem. $\Theta \in \Sigma$ is the problem *initial state*. Ops is a set of search operators, $op : \Sigma \rightarrow \Sigma$, each of which maps a state in Σ to a next state, in Σ. Goal-Test: $\Sigma \rightarrow \{\text{True}, \text{False}\}$ specifies for each state whether or not it satisfies the problem goals. The utility function g maps a sequence of operators to $\Re$, and represents the utility of applying the operator sequence, starting in initial state Θ.

A *candidate solution* to a search problem is a sequence of operators that maps initial state Θ to a state in Σ. A *feasible solution* is a candidate that maps the initial state to a state that satisfies Goal-Test. An *optimal solution* is a feasible solution that maximizes utility g.

A version of the A* search algorithm is shown in Figure 17. A* search incrementally expands a search tree, rooted in the problem's initial state. Each tree node is labeled with a search state and each branch is labeled with an operator. The children of each node are states that are generated by applying each search problem operator to the node's search state. This is performed by the function Expand(*search-state*, *problem*).

```
function Initialize-A*-Search(problem)
  returns Problem with its search-tree initialized.
  Nodes[problem] ←
      Make-Queue(Make-Search-Tree-Node(Θ[problem],NoParent))
  Expanded[problem] ← {}
  return problem

function A*-Search(problem, h)
  returns the next best solution or failure.
  f(x) ← g[problem](x) + h(x)
  loop do
     if Nodes[problem] is empty then return failure
     node ← Remove-Best(Nodes[problem], f)
     remove any n from Nodes[problem] such that State(n) = State(node)
     Expanded[problem] ← Expanded[problem] ∪ {State(node)}
     new-nodes ← Expand(node, problem)
     for each new-node in new-nodes
        unless State(new-node) is in Expanded[problem]
           then Nodes[problem] ← Enqueue(Nodes[problem], new-node, f)
     if Goal-Test[problem] applied to State(node) succeeds
        then return node
  end
```

Fig. 17. A* Search. *Problem* is a state space search problem with utility function g, initial state Θ, node expansion function Expand, and a *Goal-Test* for identifying states that are solutions. h is an admissible heuristic for *problem*. Variations of this algorithm are used by Constraint-based-A* (Figure 19), Next-Best-Kernel (Figure 31) and Conflict-Directed-A* (Figure **??**).

A path through the tree from the root to a leaf node constitutes a candidate solution, consisting of the operators on the branches along the path. The feasible solutions are the leaf nodes whose state satisfy the goal test. An optimal solution is one of the leaves that maximizes utility g.[4]

A* search expands search tree nodes in order of *estimated utility* f,

$$f(n) = g(n) + h(n)$$

where n is a node, $g(n)$ is the utility of the path from the root to that node, and $h(n)$ is a heuristic that estimates the maximum utility gained by reaching a state that satisfies goal test. An *admissible* heuristic $h(n)$ never underestimates

[4] A* search is typically framed as finding a minimal cost solution; however, without loss of generality, we frame it as maximizing utility to maintain consistency with the development of optimal CSPs.

the utility of reaching the goal. The set of k leading solutions are found by first calling Initialize-A*-Search, followed by k successive calls to A*-Search.

4.2 The Dynamic Programming Principle

Dynamic programming is an important element of standard A* search. A* may be viewed as a search for the best utility path from Θ to one of the goal states. Multiple paths may exist to any intermediate state within the state space. Nodes on the A* search queue represent different partial paths from Θ to intermediate states, and are waiting to be extended in best first order. Exploring all paths that go through each intermediate state to ensure optimality can be quite costly.

The dynamic programming principle tells us that the best utility path that goes from the root to a goal state through a node s, will always contain, as a prefix, the shortest path p from the root to s. The key consequence is that, without sacrificing optimality, A* search may remove from the search queue the remaining, sub-optimal paths to s, without further exploration.

The dynamic programming principle also tells us that, for each intermediate state s, A* will always remove from the queue the node representing the best utility path p to s first. That is, it occurs before removing those nodes representing suboptimal paths to s. To remove nodes representing suboptimal paths, whenever a node n is *taken off* the queue, A* removes any node on the queue that has the same search state as n. This test must be performed when a node is removed from the queue, rather than when it is added. This ensures that the best cost path to the intermediate state has been found. In addition, a list of already expanded nodes is maintained (called Expanded[problem]), and used to check if newly created nodes refer to an already expanded state.

*4.3 Optimality of A**

Given that h is admissible, A* has three desirable features: It is *correct*, that is, the value it returns is guaranteed to be the global optimum. This is in contrast to local search methods that can become stuck in a local optimum. A* is *complete* for locally finite graphs, that is, if a feasible solution exists, then A* is guaranteed to return an optimal feasible solution. Finally, A* is *efficient*, that is, it explores no node with heuristic utility h less than the optimum.

A* is characterized as *optimally efficient*[37]. For example, Russell and Norvig [38] observe 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 beyond a generic state space search problem. In particular, for optimal CSPs, states are factored into variable assignments, and the constraint $C_{\mathbf{x}}$ is factored into a conjunction of constraints. Like A*, conflict-directed A* must preserve efficiency, that is, it should not explicitly consider any state whose $g(n)$ is worse than the optimal solution. For correctness it must also rule out any state whose $g(n)$ is better than the optimum. However, while A* rules out these states *explicitly*, conflict-directed A* rules out many of these states *implicitly.*

4.4 The Constraint-based A Algorithm*

In this section we develop constraint-based A*, a variant of A* that solves optimal CSPs by exploiting MPI, but not conflicts. Examples of partially expanded search trees for constraint-based A* are shown in Figures 25 - 28. For a CSP 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 order in which these assignments is made is irrelevant. The search state of a leaf node is one of the states of the CSP. The search states of the set of all leaf nodes is equivalent to the state space of the CSP. We refer to this tree as an *assignment tree.* Functions that support this definition of search tree for CSPs are shown in Figure 18.

Given an OCSP, constraint-based A* (Figure 19) guides the selection of variable assignments towards the optimal decision state that satisfies $C_{\mathbf{y}}$, (according to function Goal-Test-State). We extend A* search to constraint systems by performing A* search using specialized versions of A* functions Goal-Test, g and h, and Expand that operate on assignment trees.

Recall that dynamic programming is an important element of standard A* search, since in general multiple paths may exist to any intermediate state within the state space. The dynamic programming principle avoids further expansion of those paths to an intermediate state that are sub-optimal. Constraint-based A* has the property, shared with most CSP algorithms, that it does not generate multiple paths to the same partial assignment. As a consequence, the dynamic programming principle is not needed for constraint-based A*. Note, however, that we will need to reintroduce a variant of the principle when we

```
function Make-Tree-Node(assignment, parent)
  return ⟨assignment, parent⟩

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] ∪ State[Parent[node]]
  end

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

Fig. 18. Functions for constructing and examining a search tree for a CSP, called an assignment tree. These functions are used by Constraint-Based-A* (Figure 19), Next-Best-Kernel (Figure 31) and Next-Best-State-Resolving-Conflicts of Conflict-Directed-A* (Figure 35).

incorporate conflicts into the search.

The utility $f(n)$ of node n is an upper bound on the utility of all states of the leaf nodes appearing below node n. More specifically, $g(n)$ is the utility of the assignments along the path from the root to n, while $h(n)$ is an upper bound estimate on the utility of assigning the remaining variables. Definitions for g and h are shown in Figure 20.

MPI is the key to defining a heuristic evaluation function that may be computed efficiently. Heuristic function h is defined by exploiting the property of mutual preferential independence (MPI). Recall that, since utility function g is MPI, it follows that, if $u \geq v$, then $G(u, w) \geq G(v, w)$. Hence, the utility of a decision state is maximized by maximizing the utility 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 maximum utility completion

```
function Constraint-based-A*(OCSP)
  returns the leading minimal cost solutions of OCSP.
  f(x) ← G[problem](g[problem](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. 19. Constraint-based A*.

of $\mathbf{z}$ is [5]

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

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

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

Utility is maximized by maximizing probability, and the probability of each component is maximized if it is in the "Good" mode, hence,

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

Note that the definition of $h(\mathbf{z})$ is an optimistic estimate on the utility of all extensions, h is admissible, hence, constraint-based A* is guaranteed to come

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

up with an optimal solution. h is only an estimate since, although a state must exist with utility $h(n)$, that state may be inconsistent with $C_{\mathbf{y}}$.

---

function **g**[*problem*](*node*)
  returns the utility 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 utility to complete *node*'s state.
  unassigned ← all variables not assigned in State[*node*]
  return G_{max}[*problem*](*unassigned*)

function G_{max}[*problem*](*variables*)
  returns the maximum utility 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_{max}[*problem*](y_i), G_{max}[*problem*](*remaining*))

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

Fig. 20. Utility g and heuristic utility h for an optimal CSP. These functions are used by Constraint-Based-A* (Figure 19), Next-Best-Kernel (Figure 31) and Next-Best-State-Resolving-Conflicts of Conflict-Directed-A* (Figure 35).

Finally, consider the expansion function, Expand-Variable. Given a node n, a straight forward implementation of function Expand-Variable would 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 expanded at each step.

A key consequence of mutual preferential independence is that it enables Constraint-Based-A* 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 $g_i(v_{ij}) \geq g_i(v_{ik})$. Then

there exists a leaf node $l1$ under $c1$ such that for all leaf nodes $l2$ under $c2$, $g(\text{State}[l1]) \geq g(\text{State}[l2])$.

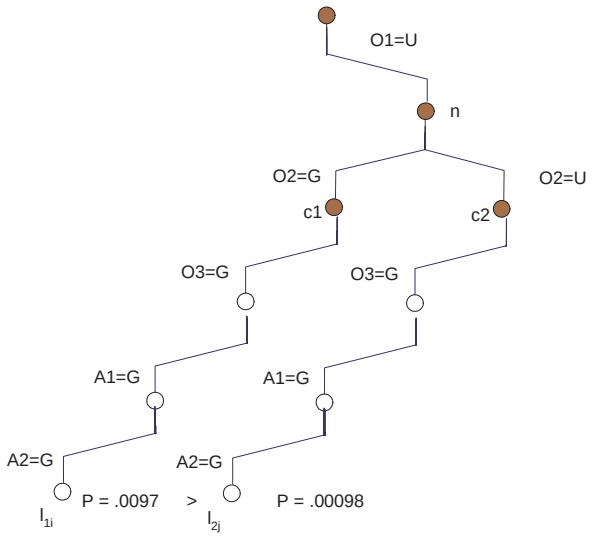

Fig. 21. An example demonstrating that, given two children, $c1$ and $c2$, the child with the better utility, $c1$, always contains a state l_{1i} with a better utility than the best state, l_{2j}, of $c2$.

For example, suppose we have a node n with state $\{O1 = \mathbf{U}\}$ (Figure 21). Furthermore, suppose we expand n using $O2$, hence, n has a child $c1$ for $\{O2 = G\}$ and a child $c2$ for $\{O2 = \mathbf{U}\}$. $g_i(O2 = G) = .99$, while $g_i(O2 = \mathbf{U}) = .01$, hence, $c1$ has a leaf that is $\geq$ all the leaves of $c2$. In particular, by MPI the best leaf, l_{1i}, of $c1$ is $\{O1 = \mathbf{U}, \boxed{\text{O2=G,}}\ O3 = G, A1 = G, A2 = G\}$, with probability $.01 \times .99 \times .99 \times .995 \times .995 = .0097$. This is better than the best leaf, l_{2j}, of $c2$, which by MPI is $\{O1 = \mathbf{U}, \boxed{O2 = \mathbf{U},}\ O3 = G, A1 = G, A2 = G\}$, with probability $.01 \times .01 \times .99 \times .995 \times .995 = .00098$. We note that these two best children only differ by the assignments to $O2$, indicated by boxes. This is a consequence of MPI.

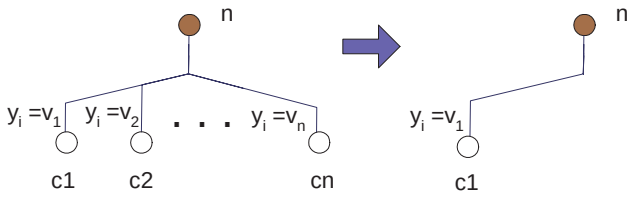

Fig. 22. Due to MPI, only the child of a node with the best cost assignment needs to be expanded (right), rather than all children (left).

Now consider a node n that is to be expanded using an unassigned variable y_i. We rank the values of D_i in decreasing order of utility g_i. We use v_1 to denote the value that maximizes g_i, we use v_2 to denote the 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 22). This follows because some leaf, l_{1n}, of $c1$ must exist whose utility is greater than or equal to all leaves of the siblings of $c1$. Hence the best utility 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 23. [6]

---

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 upper bound utility 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 $g_{y_i}[problem](v_{ij}) \geq g_{y_i}[problem](v_{ik})$

Fig. 23. Expanding the best child for Constraint-Based-A*. Expand-Domain is also used by Next-Best-State-Resolving-Conflicts of Conflict-directed-A* (Figure 35)

Node $c1$ is guaranteed to contain the best state *only until one or more of the states of* $c1$ *have been eliminated,* for example, by eliminating one of $c1$'s leaf nodes. At this point we may have eliminated $l1_n$, in which case the best leaf node of $c2$ may have a greater utility than the remaining unexplored leaves of $c1$. Hence once a leaf of a node c_n is eliminated, we must create a node for its

[6] For simplicity of presentation, in the figure, Expand-Domain orders the constituent kernels by utility when the node is created. For efficiency, the implementation performs this ordering when the constituents are created.

```
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. 24. Expanding the best sibling for constraint-based A*. Expand-Next-Best-Sibling is also used for expansion by Next-Best-Kernel (Figure 31) and Next-Best-State-Resolving-Conflicts (Figure 35).

next best sibling c_{n+1}. This sibling is created using the function Expand-Next-Best-Sibling, shown in Figure 24. 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 24. In this approach also note that a node is only expanded when its partial assignment proves consistent.

A* traditionally expands all children of a node, producing at most b nodes, where b is the maximimum 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. 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.

4.5 Constraint-based A for Boolean Polycell*

Consider the results of node expansion when Constraint-based-A* is applied to Boolean polycell. On the first iteration of Constraint-based-A*, search begins with only the root node on the search queue. This is node n_1 of Figure 25. The root is taken off the queue and its best child (n_2) is expanded, by selecting $O3$ as an unassigned variable and assigning it its best assignment, G. Next, n_2 is taken off the queue, and its best child, n_3 is generated. A similar process generates n_4, n_5 and finally n_6, which is the best state,

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

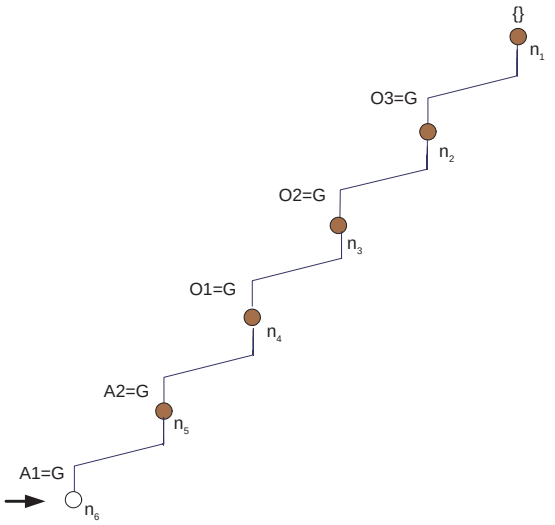

Fig. 25. Search tree created by Constraint-based-A* to generate the best utility state. The best state, Candidate 1, is $\{O1 = G, O2 = G, O3 = G, A1 = G, A2 = G\}$, and is indicated by an arrow.

Node n_6 is a leaf node, hence when it is removed from the search queue, Expand-Variable generates the next best sibling of that node and all its ancestors, producing n_7 – n_{11} in Figure 26.

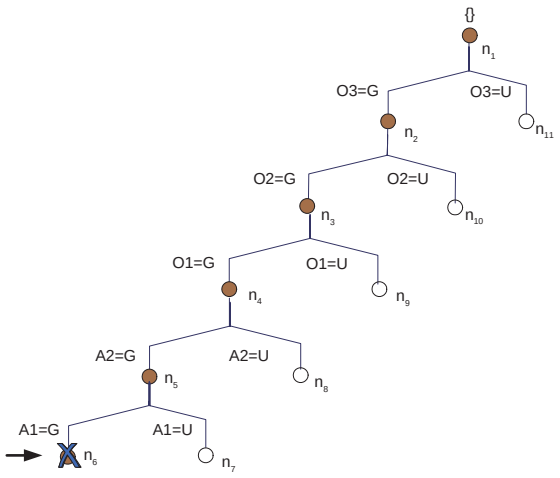

Fig. 26. When leaf node n_6 is expanded, the best sibling of n_6 and all its ancestors are created.

Constraint-Based-A* uses Goal-Test-State to test Candidate 1 for consistency and proves it inconsistent. Hence the search continues with the current search agenda to find the next best utility state. Nodes $n_9 - n_{11}$, which are at the front of the search queue, all have the same utility. We will assume that n_{11} is taken off the search queue for expansion. Expand-Variable repeatedly generates the best descendants of n_{11}, which are $n_{12} - n_{15}$, shown in Figure 27. n_{15} is a complete assignment, and is returned as a candidate,

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

When leaf node n_{15} is removed from the search queue, Expand-Variable generates the next best sibling of n_{15} and its ancestors, which are nodes $n_{16} - n_{19}$ in Figure 28. Expand-Variable does not generate a next best sibling for n_1, because the domain of O_3 has been exhausted.

Constraint-Based-A* also determines that n_{15} is inconsistent, and the search is continued for a third round, with n_{10} at the top of the queue. As before, its best descendants are expanded depth first, by selecting the best value of its unassigned variables, generating nodes $n_{20} - n_{22}$ (Figure 28), and candidate

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

This candidate is also inconsistent, hence, the process repeats until the desired set of best consistent candidates has been found.

This completes our development of basic constraint-based A*. To summarize, constraint-based A* introduces three key concepts. First, an OCSP may be

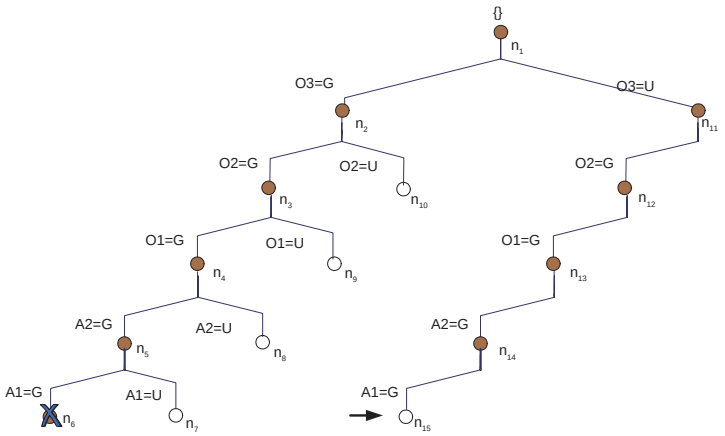

Fig. 27. The best descendants of n_{11} are expanded to produce the second best utility state, $\{O1 = G, O2 = G, O3 = \mathbf{U}, A1 = G, A2 = G\}$. This state is indicated by an arrow.

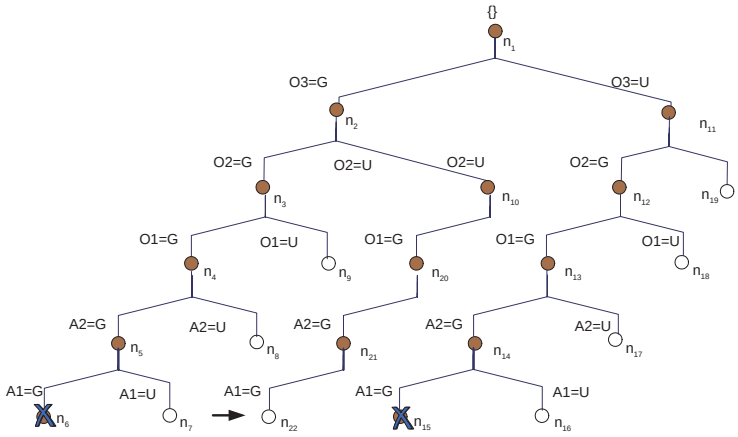

Fig. 28. After eliminating n_{15}, the next best siblings of its ancestors are expanded. Next the best descendants of n_{10} are expanded to produce the third best utility state, $\{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = G\}$. This state is indicated by an arrow.

solved by performing an A* search on an assignment tree, representing the space of all partial assignments. 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 utility for each unassigned variable. Finally, queue growth is reduced by only expanding, for each node, waiting until one of the child's states is eliminated, before expanding its next best sibling.

the best child that has no eliminated states.

5 Generating the Best Kernel

In Section 2.5 we saw how conflict-directed A* jumps over the leading set of conflicting states by generating the kernel containing the best state. In this section we develop a function, called Next-Best-Kernel, that generates this best kernel. In Section 6 we use this result to develop conflict-directed A*.

More generally, repeated invocations of Next-Best-Kernel provides an effective algorithm for generating a compact description of most or all "good" solutions. Early diagnostic approaches [1,15,2,17] generated a complete description of the diagnostic space by generating all kernels from 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, for example, when the generated kernels cover most of the probability density space of valid diagnoses. This approach is particularly effective for cases where a small collection of kernels covers the majority of the diagnoses, and where the remaining, exponential number of kernels collectively cover a small portion of the probability density space. The approach offers an any-time, any-space algorithm, which increases its coverage of the solution space as additional time and memory permits.

5.1 Definitions

In Section 2 we introduced several terms, such as *conflict* and *kernel*, in order to describe conflict-directed A*. This section defines these terms formally, generalizing from concepts introduced in model-based diagnosis, particularly [1,2,15].

Recall that 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 we can think of a conflict as denoting an inconsistent subset of the state space. Any state contained by a conflict is said to *manifest* that conflict. Any state not contained by a conflict is said to *resolve* that conflict.

Definition 8 Let $\mathbf{y}$ be a set of variables with domain $\mathbb{D}_{\mathbf{y}}$ and state space $\mathbb{S}_{\mathbf{y}}$. A *partial assignment, α, to* $\mathbf{y}$ is a set of assignments to a subset of $\mathbf{y}$ that contains *at most one* assignment for every $y_i \in \mathbf{y}$. The set of all partial assignments is denoted $\mathbb{P}_{\mathbf{y}}$. A partial assignment $\beta \in \mathbb{P}_{\mathbf{y}}$ *extends* α if $\alpha \subset \beta$. The *states of a partial assignment* α, denoted States(α), is the set of all states, $s \in \mathbb{S}_{\mathbf{y}}$, that extend α.

For example,

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

is a partial assignment for Boolean polycell that leaves A2 unassigned. It denotes a subspace consisting of two states,

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

Definition 9 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 $C_{\mathbf{y}}$. A *minimal conflict* of $C_{\mathbf{y}}$ is a conflict of $C_{\mathbf{y}}$, no proper subset of which is a conflict. 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. If s manifests α, then α is called a *state conflict* of s.

For Boolean polycell, the state

$$s_1 : \{O1 = G, O2 = G, O3 = G, A1 = G, A2 = G\}$$

contains two minimal state conflicts,

Conflict 1: $\{O1 = G, O2 = G, A1 = G\}$ and
Conflict 2: $\{O1 = G, A1 = G, A2 = G\}$,

which are the two conflicts identified in Section 2.4. Hence, s_1 manifests Conflicts 1 and 2. State

$$s_2 : \{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\}$$

resolves Conflicts 1 and 2, since neither conflict is a subset of s_2.

Recall that the task of conflict-directed A* is to jump over subspaces known to be inconsistent, without further considering any of the states within those subspaces explicitly. The states jumped over are the states of the known conflicts. To accomplish this we *invert* the known conflicts, by generating descriptions of all subsets of the state space that resolve these conflicts. Subspaces of states that resolve a set of conflicts are described by partial assignments called *kernels*. Every state contained by a kernel resolves every known conflict. Conversely, 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.

Definition 10 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 $\{\beta \in \mathbb{P}_{\mathbf{y}} | \beta \text{ is a kernel of } \Gamma\}$.

A kernel for Conflict 1 and 2 is

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

In particular, no state of Kernel 1 can manifest Conflict 1, since $O1 = \mathbf{U}$ guarantees that the assignment $O1 = G$ of Conflict 1 will not occur. The same holds for Conflict 2.

Two consequences of these definitions are important. First, the complete set of kernels characterize all states that are not ruled out by one of the known conflicts.

Proposition 2 Every state that resolves a set of conflicts Γ is a member of at least one kernel of Γ. Conversely, every member of the kernels of Γ resolves Γ.

In addition, conflicts and kernels are related through prime implicants.[7]

Proposition 3 Given a set of conflicts Γ, let Δ denote the conjunction

$$\bigwedge_{\gamma \in \Gamma} \neg(\bigwedge_{(x_i = v_{ij}) \in \gamma} x_i = v_{ij}).$$

Then the kernels of Γ are the prime implicants of Δ.

These two propositions follow by direct analogy to Theorem 3 of [2].

The complete set of kernels for Conflict 1 and 2 consists of

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

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

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

[7] An implicant of a theory Θ is a conjunction of literals that entail Θ. A prime implicant ϕ of Θ is an implicant of Θ such that no proper subset of the literals of ϕ form an implicant of Θ.

Our concept of kernel is similar to that of kernel diagnosis in [2]. Beyond the generalization of kernel diagnosis to CSPs, the primary difference is that a kernel resolves the *known* conflicts, while a kernel diagnosis resolve *all* conflicts.

Finally, in Section 2.5 we generated the kernels of conflicts Γ by first generating the kernels of each conflict separately. We call these the *constituent kernels* of Γ.

Definition 11 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 $\{\text{kernels of } \gamma | \gamma \in \Gamma\}$.

For example, given conflicts Γ,

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

the corresponding set of constituent kernels is

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

We use constituent kernels in Section 5.2 to define a procedure for mapping conflicts to kernels.

Note that previous approaches, such as [1,2] blur the distinction between what we call conflicts, and constituent kernels. However, these distinctions are conceptually important. A conflict is a set of assignments that entail a particular set of "conflicting" values. The constituent kernels remove the contribution of the conflict.

5.2 Mapping Conflicts to All Kernels

Recall for conflict-directed A*, that the function Next-Best-State-Resolving-Conflicts uses the set of known conflicts to generate a kernel that contains the best cost state resolving the known conflicts. As we walked through the Boolean polycell example in Section 2.5, we generated the kernels of a set of conflicts, Γ, by first generating the constituent kernels of Γ, and then combining them to create the kernels. In this section we state the algorithm for generating *all kernels of* Γ. This algorithm generalizes the candidate generation algorithm introduced in the GDE system [1], and is a stepping stone to the generation of the best kernel, presented in the next subsection.

First consider the generation of the constituent kernels of Γ. Note that a state that manifests a conflict $\gamma \in \Gamma$ is one that entails the conjunction

$$\bigwedge_{x_i = v_{ij} \in \gamma} x_i = v_{ij}.$$

For example, if a state manifests Conflict 1, $\{O1 = G, O2 = G, A1 = G\}$, then it entails

$$O1 = G \wedge O2 = G \wedge A1 = G.$$

A state that resolves γ is one that entails its negation,

$$\neg \left(\bigwedge_{x_i = v_{ij} \in \gamma} x_i = v_{ij} \right),$$

or equivalently

$$\bigvee_{x_i = v_{ij} \in \gamma} x_i \neq v_{ij}.$$

This clause specifies that at least one assignment of γ *does not* hold in any state s that resolves γ. For example, the clause for Conflict 1 is

$$O1 \neq G \vee O2 \neq G \vee A1 \neq G.$$

This clause holds in state s exactly when at least one variable of an assignment in γ is assigned a different element of its domain in state s.

Proposition 4 *The set of (constituent) kernels of conflict γ is*

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

For example, in Conflict 1 the three variables - $O1$, $O2$ and $A1$ - have domain $\{G, \mathbf{U}\}$. We create the complete set of constituent kernels for Conflict 1 by replacing each assignment of Conflict 1 with its alternative domain assignments,

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

Likewise, for Conflict 2,

$$\{O1 = G, A1 = G, A2 = G\},$$

the complete set of constituent kernels is

$$\{\{O1 = \mathbf{U}\}\{A1 = \mathbf{U}\}\{A2 = \mathbf{U}\}\}.$$

The procedure for generating Constituent-Kernels of a set of conflicts, Γ, is provided in Figure 29, and directly follows from Proposition 4. Function Constituent-Kernels incurs negligible computational cost; its worst case computational complexity is on the order of $\sum_{D_{x_i} \in \mathbf{D_x}} |D_{x_i}|$.

---

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* $\cup \{S\}$

Fig. 29. Constituent-Kernels generates the kernels for each conflict $\gamma \in \Gamma$ separately. Add-To-Minimal-Sets adds set S to Set, while eliminating any element that is a superset of another. Constituent-Kernels is used by functions Kernels (Figure 30), Next-Best-Kernel (Figure 31) and Next-Best-State-Resolving-Conflicts (Figure 35).

Next we generate the kernels of Γ from its constituent kernels, by exploiting the following proposition.

Proposition 5 *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. For example, combining $\{O2 = \mathbf{U}\}$ from Conflict 1 with $\{A1 = \mathbf{U}\}$ from Conflict 2 produces $\{O2 = \mathbf{U}, A1 = \mathbf{U}\}$, which is subsumed by combining $\{A1 = \mathbf{U}\}$ with $\{A1 = \mathbf{U}\}$, producing $\{A1 = \mathbf{U}\}$. Finally, 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. For example, suppose we had constituent kernel $\{A1 = G\}$ for a third conflict, which we combined with $\{A1 = \mathbf{U}\}$, taken from constituents for Conflict 1 and Conflict 2. This would produce $\{A1 = G, A1 = \mathbf{U}\}$, which is inconsistent and hence eliminated. The remaining unions represent the complete set of kernels. For Conflicts 1 and 2 these are $\{O2 = \mathbf{U}, A2 = \mathbf{U}\}$, $\{A1 = \mathbf{U}\}$ and $\{O1 = \mathbf{U}\}$.

The corresponding procedure, called Kernels, is given in Figure 30. Kernels is analogous to the candidate generation algorithm used in the GDE system[1], whose soundness and completeness was demonstrated by Corollary 1 of [2].

---

function Kernels(Γ)
  returns A set of all kernels that resolve conflicts Γ.
  kernels $\leftarrow \{\{\}\}$
  augmented-kernels $\leftarrow \{\}$
  for K_γ **in** Constituent-Kernels(Γ)
    for K **in** *kernels*
      if $\exists\{x_i = v_{ij}\} \in K_\gamma$ such that $x_i = v_{ij} \in K$
        then *augmented-kernels* $\leftarrow$
          Add-To-Minimal-Sets(*augmented-kernels*, K)
        else
          for $\{x_i = v_{ij}\}$ **in** K_γ
            if x_i is not mentioned in any assignment of K
            then *augmented-kernels* $\leftarrow$
              Add-To-Minimal-Sets(*augmented-kernels,* $K \cup \{x_i = v_{ij}\}$)
    kernels $\leftarrow$ *augmented-kernels*
    augmented-kernels $\leftarrow \{\}$
  return *kernels*

Fig. 30. Procedure for generating the kernels of a set of conflicts Γ.

Consider the application of Kernels to Conflict 1 and 2 for Boolean polycell.

Kernels first generates the set of constituent kernels,

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

The first run through the outer loop assigns K_γ to the first set of constituent kernels, and then transfers this to variable *kernels*,

$$\textit{kernels:}\ \{\{O1 = \mathbf{U}\}\{O2 = \mathbf{U}\}\{A1 = \mathbf{U}\}\}.$$

During the second run through the outer loop, K_γ is assigned the second set of constituent kernels

$$K_\gamma : \{\{O1 = \mathbf{U}\}\{A1 = \mathbf{U}\}\{A2 = \mathbf{U}\}\}.$$

During the second run, the inner loop uses K to iterate over *kernels*, first assigning $\{O1 = \mathbf{U}\}$ to K. It detects that K already contains one of the elements of K_γ, and hence adds K to *augmented-kernels* without extension,

$$\textit{augmented-kernels:}\{\{O1 = \mathbf{U}\}\}.$$

Next the inner loop assigns $\{O2 = \mathbf{U}\}$ to K. This does not contain an element of K_γ, hence it tries to extend K with each element of K_γ. It starts with $\{O1 = \mathbf{U}\}$ producing $\{O1 = \mathbf{U}, O2 = \mathbf{U}\}$. This is added to *augmented-kernels* using Add-Minimal-Sets. However, $\{O1 = \mathbf{U}, O2 = \mathbf{U}\}$ is a superset of $\{\mathrm{O1} = \mathbf{U}\}$, which is already a member of *augmented-kernels,* hence $\{O1 = \mathbf{U}, O2 = \mathbf{U}\}$ is eliminated as not minimal. Next, $\{A1 = \mathbf{U}\}$ is added to K, creating $\{O2 = \mathbf{U}, A1 = \mathbf{U}\}$, and is successfully added to *augmented-kernels.* Finally, $\{A2 = \mathbf{U}\}$ is added to K, creating $\{O2 = \mathbf{U}, A2 = \mathbf{U}\}$, which is also added, producing

$$\textit{augmented-kernels:}\{\{O1 = \mathbf{U}\}\{O2 = \mathbf{U}, A1 = \mathbf{U}\}\{O2 = \mathbf{U}, A2 = \mathbf{U}\}\}.$$

Finally, the inner loop assigns $\{A1 = \mathbf{U}\}$ to K. This contains an element of K_γ, hence it tries to add $\{A1 = \mathbf{U}\}$ to *augmented-kernels* without extension. Minimal-Sets determines that *augmented-kernels* has an element $\{O2 = \mathbf{U}, A1 = \mathbf{U}\}$, which is a superset of $\{A1 = \mathbf{U}\}$, hence this superset is removed

and $\{A1 = \mathbf{U}\}$ is added to *augmented-kernels*, producing

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

This is the familiar set of kernels, previously shown in Figure 12.

Note that the function Kernels is worst case exponential in the number of conflicts. Generating the set of kernels is equivalent to minimal set covering, and is NP Hard.

5.3 Generating the Best Kernel

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 function Kernels, introduced in the last section, can be viewed as an uninformed, breadth first search through a space of partial kernels. At each iteration these partial kernels are 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.

In this section we replace this search strategy with an informed search for the best kernel, implemented by the function Next-Best-Kernel in Figure 31. As we will see, this search has strong similarities to constraint-based A*, introduced in the preceding section.

The search tree for the kernels of Boolean polycell was shown in Figure 13. Each node has an associated conflict and branches to the children of the node are labeled with the constituent kernels of that conflict. The partial kernel associated with a node is the set of variable/value assignments along the path from the root of the tree to that node. Kernels are check marked, while eliminated nodes are crossed out. For example, the check marked node at the bottom left of the tree in Figure 13 corresponds to the kernel

$$\{O2 = \mathbf{U}, A2 = \mathbf{U}\},$$

while the crossed off node to the far left corresponds to

$$\{O2 = \mathbf{U}, O1 = \mathbf{U}\}.$$

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

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

Fig. 31. 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 32 and 33. Functions Make-Tree-Node, Root?, State and Theta are the same as for Constraint-Based-A*, and were given in Figure 18. g, h, G_{min} and g_{min} are also the same, and were given in Figure 20.

This second node is not a kernel, even though it resolves all conflicts, because it is not minimal.

This search tree is closely related to the search tree constructed by constraint-based A*. The edges of both trees are labeled with assignments, the trees are both rooted in the empty set, and in both cases the search state of each node is the set of assignments along the path from the root to that node. Given this similarity, Next-Best-Kernel is able to use the same functions for creating and

examining trees - Make-Tree-Node, Root?, State and Theta - as were used in constraint-based A* (Figure 18).

One difference 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 constitutent kernels, and hence all conflicts have been resolved (Figure 32).

---

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

Function g and h for evaluating node utility are the same as for constraint-based A*, and were given in Figure 18. The reason for this is that the partial kernels and partial states of the two search trees are both partial assignments, and the utility in both cases is the best utility extension of the partial assignment. Note that one difference in behavior is that the value of h for a goal-node in constraint-based A* will be 0, since the node is a full assignment, while the value of h for a goal-node for Next-Best-Kernel will typically be non-zero, since a kernel is a partial assignment.

The remaining difference between the algorithms is the set of children generated by Next-Best-Kernel versus constraint-based A*. For constraint-based A*, each child selects a domain element of an unassigned variable. For Next-Best-Kernel, each child selects a constituent kernel of an unresolved conflict. This difference requires modification to the node expansion function, as shown in Figure 33. Recall that the node expansion function of constraint-based A* (Figures 23 and 24) selects an unassigned variable and creates a child for each element of its domain. To generate best kernels, we create an expansion function, called Expand-Conflict, that 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, {}, in Figure 13 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.

For constraint-based A*, recall that a consequence of mutual preferential independence is Proposition 1, which allows us to only expand the best child of a node, rather than all children. This expansion involves ordering the assign-

ments of the selected variable's domain based on the assignment's utility, and selecting the assignment with the best utility as the best child.

Like constraint-based A*, Next-Best-Kernel also exploits mutual preferential independence to expand only the best child of a node; however, the criteria used by Next-Best-Kernel to determine the best child is more complex. The reason is that Proposition 1, used by constraint-based A*, requires that all the assignments being considered must refer to the same variable. However, the assignments in the constituent kernels of a conflict typically refer to different variables. Hence in the case where the two children being compared have different variables, Proposition 1 does not apply. To address this we introduce an additional proposition that establishes a criteria for ordering the assignments of children involving different variables, and we exploit this criteria to expand only the best child.

Proposition 6 Let $c1$ and $c2$ be sibling nodes with parent n, where $c1$ is labeled with assignment $y_i = v_{ij}$, $c2$ is labeled with $y_k = v_{kl}$, $y_i \neq y_k$, and neither y_i nor y_k appear in State[n]. Let $g_{y_i}^{max}$ and $g_{y_k}^{max}$ denote the best attribute utilities of y_i and y_k, respectively. If $G(g_{y_i}(v_{ij}), g_{y_k}^{max}) \geq G(g_{y_i}^{max}, g_{y_k}(v_{kl}))$, then there exists a leaf node $l1$ under $c1$ such that for all leaf nodes $l2$ under $c2$, $g(\text{State[l1]}) \geq g(\text{State[l2]})$.

The key difference from Proposition 1 is that, during the comparison, the utility of each child's assignment is weighted by the best assignment utility for the other child's variable. This is because $c1$ 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 must be performed under the assumption that the two children take on best utility values for their sibling's variable.

For example, consider the node labeled $O2 = \mathbf{U}$ in Figure 13. 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 $c2$ if

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

Simplification demonstrates that the relation is satisfied,

$$\begin{aligned} P(O1 = \mathbf{U}) \times P(A2 = G) &\geq P(A2 = \mathbf{U}) \times P(O1 = G), \\ .01 \times .995 &\geq .005 \times .99, \\ .00995 &\geq .00495. \end{aligned}$$

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 33. Let k_n denote the nth kernel in this ordering, and c_n denote the corresponding child. It follows from Proposition 6 that only the first child, c_1, needs to be expanded. This is performed by function Expand-Conflict-Best-Child in Figure 33.

---

function Expand-Conflict(*node, problem*)
  returns the best nodes expanded from *node*.
  return Expand-Conflict-Best-Child(*node, problem*) ∪
        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 upper bound utility 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}) $\geq g_{y_k}$[*problem*](v_{kl})
    else return G[*problem*](g_{y_i}[*problem*](v_{ij}), $g_{max}(y_k, problem)$)
        $\geq$ G[*problem*]($g_{max}(y_i, problem)$, g_{y_k}[*problem*](v_{kl}))

Fig. 33. 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 24. g, h, G_{min} and g_{min} are also the same, and are shown in Figure 20. Likewise, Make-Tree-Node, Root?, State and Theta are shown in Figure 18.

Proposition 6 only holds until one or more of the states of a child c_n has been eliminated. Unlike constraint-based A*, 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 33) and Expand-Next-Best-Sibling (Figure 24). This approach is in contrast to constraint-based A*, which waits until a leaf node of c_n is expanded, before expanding its next best sibling. An example of the execution of Next-Best-Kernel has already been given in Section 2.5, and depicted in Figures 14 through 16.

5.4 The Dynamic Programming Principle Revisited

Recall from Section 4.1 that standard A* search may encounter multiple paths to the same search state. A* uses the dynamic programming principle to avoid expanding the sub-optimal paths of each search state. At the end of Section 4.4 we discussed how constraint-based A* did not need to incorporate this principle, since it generates only one path to each partial assignment. Next-Best-Kernel, however, may produce multiple paths to the same partial assignment. Hence, to improve the efficiency of Next-Best-Kernel we incorporate a variant of the dynamic programming principle.

The cost of a partial assignment is independent of the path by which it is reached, since the utility function G is associative and commutative. As a result, all search nodes with the same state must have the same cost, and Next-Best-Kernel does not need to search for the path to a state with the best utility. Rather, Next-Best-Kernel is free to expand the first node to each search state that is entered onto the queue.

Next-Best-Kernel does need 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 so, then the node is ignored.

Note that this approach is more efficient than generic A* search. The test for Next-Best-Kernel is performed when a node is added to the queue, rather than when it is removed, thus reducing the overall growth in search queue size. The dynamic programming principle was not needed for the simple Boolean polycell sequence, given in Section 2.5. However, it has a substantial impact on our performance experiments, discussed in Section 7.

To summarize the results of this section (Section 5), 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*

This section develops Conflict-Directed-A* and its key supporting function, Next-Best-State-Resolving-Conflicts. We begin by formally specifying the interaction between the generator that produces the best non-conflicting states, and the CSP algorithm that tests the consistency of these states. This allows conflict-directed A* to use a wide range of CSP representations and CSP algorithms. Next we develop conflict-directed A* for the case where we are only interested in the single best solution, building off of the function Next-Best-Kernel (Section 5). This case corresponds to the algorithm demonstrated at the beginning of the paper (Section 2). Finally, we generalize conflict-directed A* to find 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, developed in Sections 4 and 5.

6.1 Conflict-directed Generate and Test

The top-level procedure of conflict-directed A* was introduced and demonstrated in Section 2 and is shown in Figure 8. In this section we discuss the properties of each of its subroutines. Recall that conflict-directed A* repeatedly performs a generate and test loop, where generation is focussed by the set of known conflicts. The loop first uses Next-Best-State-Resolving-Conflicts to find the best decision state, according to f, that resolves all known conflicts. It then uses Consistent? to test the decision state against the CSP to determine consistency. If the decision state is inconsistent, it then generalizes the state to one or more conflicts using Extract-State-Conflicts. Finally, these new conflicts are added to the set of known conflicts, and conflicts that do not offer additional information are removed using Eliminate-Redundant-Conflicts. This loop terminates if no decision states remains or when the desired set of

solutions are found, as determined by Terminate?. [8]

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

Definition 12 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 $g(\beta) > g(\alpha)$.

Our development of Conflict-directed-A* does not commit to a specific constraint system or implementation of *Consistent?*, *Extract-State-Conflicts* and *Eliminate-Redundant-Conflicts*.

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[35]. Local search methods, such as Min-Conflict [20] or GSAT[21], 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.

[8] 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.

The most common way to extract a conflict, as mentioned in Section 2.4, 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 9 and 10 show 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 [35] that uses Boolean Constraint Propagation (BCP) [39–41,33] 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[42], 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.

6.2 Conflict-directed A: One Solution*

To complete our development of conflict-directed A*, we need to define the function Next-Best-State-Resolving-Conflicts. We consider here the case where we are interested only in the single best solution to 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 (top, Figure

34).

To extract the best state of a kernel K, let $\mathbf{z}$ be the set of variables not assigned in kernel 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,

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

This corresponds to Function Kernel-Best-State, shown in Figure 34. This case was demonstrated in detail at the beginning of the paper (Section 2).

---

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 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 $\{y_i = v_{max}[problem](y_i)\}$∪ 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. 34. Support functions for Conflict-directed-A* for the case of generating a single best solution.

```
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
```

Fig. 35. Support functions for Conflict-directed-A* for the case of generating multiple solutions. Combines expansion functions for Next-Best-Kernel (Figure 33) and Constraint-based-A* (Figure 35). Terminate? is application specific and is not supplied.

Next we extend conflict-directed A* to generate n leading solutions, where $n > 1$. To accomplish this we introduce a version of Next-Best-State-Resolving-Conflicts that is able to enumerate, in best first order, several non-conflicting states of one or more kernels. This is in contrast to the function of the preceding section, which is only able to enumerate the single best state of each kernel.

Next-Best-State-Resolving-Conflicts, defined in Figure 35, generates kernels similar to Next-Best-Kernel (Figure 33), and enumerates the states of these kernels, similar to Constraint-Based-A* (Figure 35). 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 type, using Expand-Conflict to expand partial kernels and Expand-Variable to expand kernels to states. The goal-test function, Goal-Test-State-Resolves-Conflicts, returns true when a search state is a complete assignment and resolves all conflicts. The application of the dynamic programming principle is the same as outlined in Section 5.4 for Next-Best-Kernel.

6.4 Applying Full Conflict-directed A to Boolean Polycell*

Consider the results of repeatedly invoking the multiple solution version of Next-Best-State-Resolving-Conflicts (Figure 35), when Conflict-directed-A* is applied to Boolean polycell. On the first iteration of Conflict-directed-A*, Next-Best-State-Resolving-Conflicts is called with the root node placed on the search queue (node n_1 of Figure 36) and with no known conflicts. Next-Best-State-Resolving-Conflicts starts by taking n_1 off the queue. Since there are no conflicts to be resolved, the expansion of n_1 and its children is the same as for constraint-based A*, given in Section 4.5. The best descendants of n_1 are generated in a depth first manner (nodes $n_2 - n_6$ in Figure 36), producing the best state,

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

Node n_6 is a leaf node, hence when it is removed from the search queue, Expand-State-Resolving-Conflicts generates the next best sibling of that node and all its ancestors. This is the same as for Constraint-based-A*, and is indicated on Figure 37 as nodes n_7 - n_{11}.

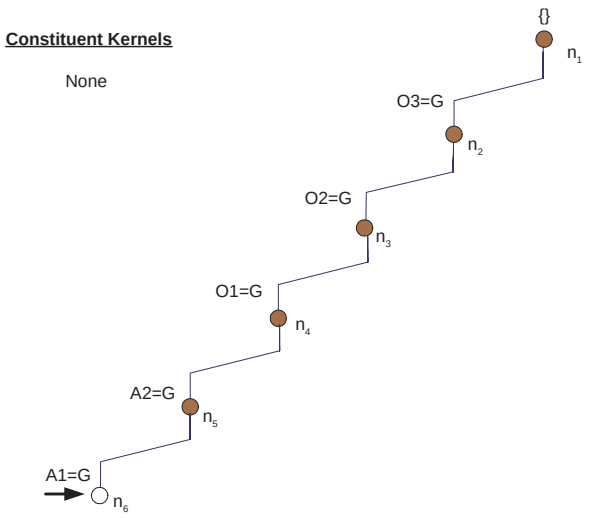

Fig. 36. Search tree created by Next-Best-State-Resolving-Conflicts to generate the best utility state, given no conflicts. The best state, Candidate 1, is $\{O1 = G, O2 = G, O3 = G, A1 = G, A2 = G\}$, and is indicated by an arrow. A closed/open circle indicates an expanded/unexpanded node.

Candidate 1 is returned to Conflict-Directed-A* and tested for consistency. It proves 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. Node n_{11}, shown in Figure 37, is taken off the search queue for expansion. Note that $n_9 - n_{11}$ all have the same utility, hence any of these nodes can be taken from the queue. n_{11} does not resolve Conflict 1, hence a best child (n_{12}) is generated for n_{11} that selects the best utility constituent kernel, $\{O2 = \mathbf{U}\}$, for Conflict 1. Note that this kernel adds an additional failure (O2 broken) and hence the utility of n_{12} is about an order of magnitude lower than that of n_{11}.

The next best node taken off the search queue is n_{10}, which has the same utility as n_{11}. This node already resolves Conflict 1, hence the node is recursively expanded to its best state by selecting an unassigned variable and assigning it its best utility value. The nodes generated are $n_{13} - n_{15}$ shown in Figure 37. n_{15} is the best state that resolves all conflicts, and hence is returned as

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

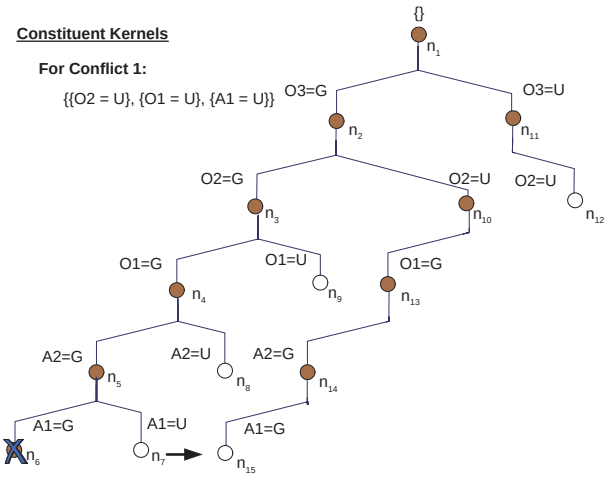

Fig. 37. Given conflict $\{O1 = G, O2 = G, A1 = G\}$, Next-Best-State-Resolving-Conflicts extends the search tree to generate the best utility state, $\{O1 = G, O2 = \mathbf{U}, O3 = G, A1 = G, A2 = G\}$. This state is indicated by an arrow. A closed/open circle indicates an expanded/unexpanded node.

n_{15} is a leaf node, hence, when it is removed from the search queue, Expand-State-Resolving-Conflicts generates the next best sibling of n_{15} and all its ancestors. These are nodes $n_{16} - n_{18}$ in Figure 38. Note that Expand-State-Resolving-Conflicts does not generate a next best sibling for n_2, since it was already generated as n_{11} when leaf node n_6 was expanded.

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

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

This conflict is added and Next-Best-State-Resolving-Conflicts is invoked for a third round. At this point node n_9 is at the top of the queue. It resolves both Conflict 1 and Conflict 2, hence this node is repeatedly expanded by selecting the best value of its unassigned variables, generating nodes $n_{19} - n_{20}$ (Figure 38), and

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

This candidate is consistent, hence providing 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.

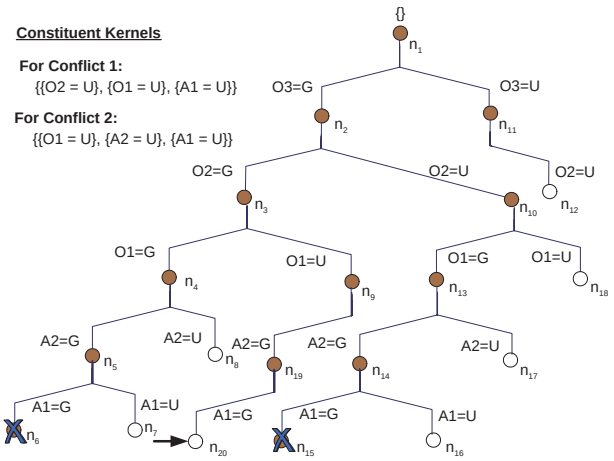

Fig. 38. Given a second conflict, $\{O1 = G, A1 = G, O2 = G\}$, Next-Best-State-Resolving-Conflicts extends the search tree to generate the best utility state, $\{O1 = \mathbf{U}, O2 = G, O3 = G, A1 = G, A2 = G\}$. This state is indicated by an arrow. A closed/open circle indicates an expanded/unexpanded node.

7 Experimental Results

We evaluated the performance of constraint-based and conflict-directed A* both on applications to real world space systems and on randomly generated problems. 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[27] and Titan[10]. These have been or are being demonstrated on several space systems, including NASA's Deep Space One probe, the Air Force TechSat 21 cluster, NASA's Messenger mission, NASA's ST-7 concept mission, and a simulated version of the Cassini Saturn space probe. The performance of an earlier variant of conflict-directed A* for the Cassini scenario was reported in [8,33].

The Cassini scenario, discussed in Section 2.2, provides a representative case study of a complex embedded system. The scenario consists of roughly 80 components, which corresponds 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* 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 of constraint-based A* was very sensitive to variable ordering. For comparison with conflict-directed A*, we consider the most 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 for constraint-based A*, as well as conflict-directed A*. For these recoveries, constraint-based A* 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.

Turning to randomized experiments, we verified the performance improve-

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 results on randomly-generated problems

Fig. 39. Average performance of Constraint-based A* and Conflict-directed A* on randomly-generated problems.

ments discussed above through a series of experiments on randomly generated problems. For these experiments each randomized data set was generated based on five parameters, which characterize 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.

Constraint-based A* and conflict-directed A* were applied to the sets of randomly generated problems, and rated, similar to above, based on total number of nodes expanded and maximum search queue length. The results of these experiments are shown in Figure 39. Once again the data shows a significant improvement in performance for conflict-directed A* over constraint-based A* across the range of problems tested. The degree of improvement varies depending on how constrained the problem is and the difficulty of the optimization problem.

The data suggests that the performance benefit of conflict-directed A* over constraint-based 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 are explored by constraint-based A*.

Conflict-directed A* also performs well for problems that are lightly-constrained. Conflict-directed A* performs well 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 result for lighty constrained problems is less significant, simple because these problems are more easily solved in general.

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 Summary

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.

In Section 3 we formalized 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, and in Section 2.2 we demonstrated that the solution to optimal CSPS with preferential independence is central to a new generation of highly robust, model-based, embedded systems.

The remainder of the paper introduced three new algorithms for tackling optimal CSPS by extending A* search – Constraint-based A*, Conflict-directed A* and Next-Best-Kernel. Traditional A* search guarantees optimality by visiting all infeasible states whose cost is better than that of the optimal feasible solution. Each of the algorithms introduced is able to reason about subsets of these infeasible states implicitly, by exploiting the structure of the CSPs and the source of conflicts.

Constraint-based 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. Constraint-based A* also exploits MPI to reduce node expansion to a single best child and sibling. Constraint-based A* easily augments existing CSP algorithms. It demonstrates promising performance both for the Cassini spacecraft scenario, and for randomly generated problems, as discussed in Section 7.

Conflict-directed A*, the central result of this paper, accelerates best first search by identifying the sources of conflict within each inconsistent candidate,

and uses this information to jump over related candidates in the sequence. 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 constraint-based A*.

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 exists 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[1,15,17] 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, depending on time and space limitations.

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. The extension of these methods to Optimal CSPS is a rich area for future research.

9 Acknowledgments

This research has gained invaluable insights through current and past research with a range of researchers. Early research along this line was pursued jointly with Johan de Kleer on assumption-based dependency-directed backtracking, and focused search algorithms for finding most likely diagnoses within the Sherlock diagnostic system. We would also like to thank Pandu Nayak, who helped refine and drive these algorithms into Deep Space through the Livingstone and Remote Agent systems. Finally, we would like to thank Seung Chung, Samidh Chakrabarti, Mitch Ingham, John Van Eepoel and Andreas Wehowsky, for their insights and for their extensive effort towards demonstrating OpSat on the Air Force Tech Sat 21 spacecraft cluster mission and NASA Messenger and ST-7 missions.

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

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

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

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

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

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

[20] 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.

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

[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 DX-90, 1990.

[26] 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.

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

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

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

[30] 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.

[31] B. C. Williams, Doing time: Putting qualitative reasoning on firmer ground, in: Proceedings of AAAI-86, 1986.

[32] P. Kim, B. C. Williams, M. Abramson, Executing reactive, model-based programs through graph-based temporal planning, in: Proceedings of IJCAI-01, 2001.

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

[34] S. C. et. al., The techsat-21 autonomous sciencecraft constellation, in: Proceedings of ISAIRAS-01, 2001.

[35] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Journal of the ACM 5 (7).

[36] G. Debreu, Topological methods in cardinal utility theory, in: K. Arrow, S. Karlin, P. Suppes (Eds.), Mathematical Methods in the Social Sciences, Stanford University Press, Stanford, California, 1959.

[37] 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.

[38] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 1995.

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

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

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

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