MASSACHUSETTS INSTITUTE OF TECHNOLOGY
16.412J / 6.834J – COGNITIVE ROBOTICS
SPRING 2016

CONFLICT-DIRECTED A*

A Gentle Introduction – by Steve Levine

INTRODUCTION

Conflict-directed A* (CDA*) is an algorithm for solving optimal CSP's, or namely constraint satisfaction problems where there's a notion of utility; certain assignments are preferred to others. A number of useful problems can be framed as optimal CSP's, such as most-likely mode estimation for diagnosing problems in engineering systems (as in this week's problem set). The general intuition behind CDA* is that it behaves much like A*, searching in best-first order, but it also learns *conflicts* – assignments to decision variables that cannot all hold true – and then actively guides the search to steer clear of these conflicts. If the conflicts learned during search are quite general, then CDA* can leap over large swaths of the search space and hopefully find a good solution much quicker than a naïve search.

CSPS, SEARCH, AND A*… OH MY!

First, let's review the definition of a Constraint Satisfaction Problem (**CSP**). A CSP is defined as a tuple $<Y, D, C>$ where Y is a set of variables, D is a set of domains for those variables, and C is a set of constraints on those variables. A solution to a CSP is a domain assignment to every variable Y such that all the constraints are satisfied.

Lots of useful things can be framed as CSP's; these include activity planning, the N-queens problem, Sudoku, satisfiability problems (SAT), and a million more. Algorithms for solving CSPs generally involve some combination of state space / graph search and local inference (ex., Backtracking with Forward checking, DPLL for SAT problems, etc*). In such algorithms, the states are partial assignments to variables, and the actions that transition between states add on more assignments to the state. The initial state is the empty assignment (no variables assigned yet), and the goal states are full assignments that are consistent* (i.e., satisfy all the constraints). This is illustrated in the following figure.

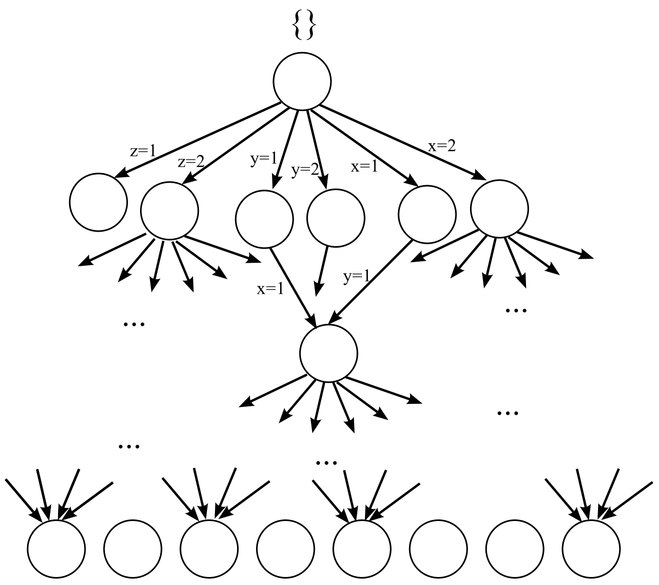
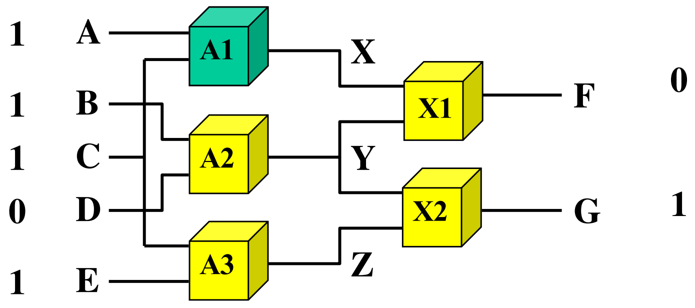


Figure 1: Solving a CSP as state-space search. The initial state is at the top, with no assignments. Actions transitioning between states incrementally add on assignments. The bottom row, which are full assignments to every variable, are goal states if they satisfy all the constraints in the CSP. We can do graph search here to try and find a solution. You might imagine this graph can be quite large… so any techniques that can prune out choices would be very useful.

CSPs are great, but sometimes when there are multiple solutions we want to be able to rank them and only find the best ones. Therefore, we introduce the notion of an **optimal CSP**. An optimal CSP is a normal CSP, except now we have denoted a subset of the variables as *decision variables* that can be ranked with a utility function. We can describe an optimal CSP as a tuple *<X, r(x), CSP>* where *CSP* is a constraint satisfaction problem as above, *X* is a subset of its variables, and *r(x)* is a reward or utility function defined only over *X* (we wish to maximize it). A solution to an optimal CSP is an assignment to all decision variables *X*, such that there exists assignments for all remaining state variables *Y* \ *X* where all the constraints are satisfied. We want to find the assignment to *X* that does this that has the highest possible utility as defined by *r(x)*.

Mode estimation and model-based diagnosis can be framed as an optimal CSP; consider the Boolean polycell example discussed in class:



In this case, the modes of A1, A2, A3, X1, and X2 are decision variables, each with domain ‘G’ (good / working) or ‘U’ (unknown / broken). Our utility function is the probability of these variables taking their respective values (for simplicity here, we assume they’re all independent). Other, non-decision variables (often called state variables) are A, B, C, D, E, F, G, X, Y, and Z. *The goal is to enumerate the highest* *utility (i.e., most likely) combination of these decision variables, such that all constraints are satisfied*. This is known as the **mode estimation problem** for **model-based diagnosis**.

This Boolean polycell example also illustrates where "decision variables" and "state variables" got their name. In this case, we're trying to "decide" what the most likely mode is of all the components, subject to the other constraints on the system's "state."

CONSTRAINT-BASED A*

So, how can we solve optimal CSP's? One solution, which we will present here first because it's simpler to get our feet wet, is known as constraint-based A* (because we use A* to search for a solution to a constraint system).

Recall that A* is a best-first search algorithm. It searches through a graph, and can find paths of minimum cost or greatest reward.

In constraint-based A*, we use A* to search in the large state space graph shown before, with the only exception that we only search over the decision variables. In other words: *States are partial assignments to decision variables, actions that transition from one state to the next add a single decision variable assignment to the states, the initial state has no assignments, and the goal state is a complete assignment to all decision variables*. Because this is only searching over the decision variables, whenever we reach a full assignment we still need to check the remaining non-decision variables to see if we can come up with an assignment to them that satisfies all of them. We encompass this into a *consistent?(x)* method, and itself often entails further search (over the non-decision variables), and is often implemented as DPLL or back-track search with forward checking.

So voila! We can use A* over this state space, right? Almost. First, we need to define how we measure cost / reward. A* search ranks states in its graph with a function $f(x) = g(x) + h(x)$, where $g(x)$ is the known cost "so far" in the search, and $h(x)$ is a heuristic; an admissible estimate of the cost remaining to the goal. What "admissible" means depends on whether your goal is to maximize or to minimize, but an easy trick is to remember that admissible means optimistic. If you're using A* to find the *minimum* distance from A to B in a graph, the heuristic must *underestimate* the length to the goal. If you're trying to find *maximum* probability assignment to variables, it should *overestimate* the probability. Optimistic in both cases.

What $f(x) = g(x) + h(x)$ should we use for constraint-based A*? Remember our states (x in $f(x)$) are partial assignments to decision variables, and we wish to maximize $r(x)$. For our Boolean polycell example, we can choose $g(x)$ to be the product of probabilities in the assignment. We can choose our heuristic $h(x)$ to be the highest-possible product of each of the remaining unassigned variables. So, we wish to maximize the following utility:

$$f(assignment) = \underbrace{\prod_{\substack{(x_i=v_k)\ in \\ assignment}} P(x_i = v_k)}_{g(x)} \underbrace{\prod_{\substack{x_j\ not\ in \\ assignment}} \max P\big(x_j = v_{max}\big)}_{h(x)}$$

(If you're concerned about why we're multiplying $f(x) = g(x)h(x)$ here instead of adding, good point! Consider what happens if we take the log of this equation, which is a monotonic function and wouldn't change the ordering of nodes picked off the A* queue. Thus, having a product that we can break up in this way is equivalent to using the summation definition $f(x) = g(x) + h(x)$ that is more commonly associated with A*. Expressing as a product is advantageous for us though since it's a probability).

And now we're done. Using the above *f(x)* to rank items in the A* queue, we can search over the decision variable state space and try to find the best-possible assignment. Pseudo code is presented below for constraint-based A*. Note that *Q* is a priority queue, so whenever we add anything to it we implicitly need to compute the *f(x)* above.

CONSTRAINT-BASED A*

```
Q = [] # priority queue
Add {} to Q
expanded = []
while Q isn't empty:
    assignment ← pop best from Q
    Add assignment to expanded # Makes sure we don't repeat visits
    if assignment is full assignment to decision variables: # potential goal
        if consistent?(assignment): # searches over non-decision variables and checks constraints
            return assignment

    else: # partial assignment to decision variables
        $x_i$ = some decision variable not assigned in assignment
        neighbors = split_on_variable(assignment, $x_i$) # split search on different assignments to $x_i$
        Add each $x_k$ in neighbors to Q if not in expanded
# If we get here, we've expanded all possibilities to no avail!
return NOSOLUTION
```

Again, note that this algorithm searches over the decision variables. We need to make sure that these values to the decision variables are also consistent with the non-decision variables though, which is accomplished via the *consistent?*(…) method above. This method may involve performing further search, such as backtrack search with forward checking or DPLL – taking into account the decision variables' assignments.

The search tree is "split" on variables; this is a fancy way of saying that we choose our successor states as extensions of the current state by picking some variable that's not yet assigned, and assigning it all possible values in its domain. Those resulting states become our successor states, and are added to *Q*.

split_on_variable(assignment, x_i)

```
return [(assignment ∪ {$x_i = d_j$}) for each $d_j$ in $x_i$'s domain]
```

Constraint-based A* will solve the problem at hand. It does however suffer from an important drawback: it can repeatedly (and unintelligently…) keep searching over problematic areas of our state space. Consider the following figure of a search tree generated from constraint-based A*. Suppose that we know that *x=1* can never hold. The search doesn't care about this; it'll keep searching the *x=1* subtree until completion, before finally moving onto the *x=2* subtree. For big state spaces, that can add up to a lot of wasted time.

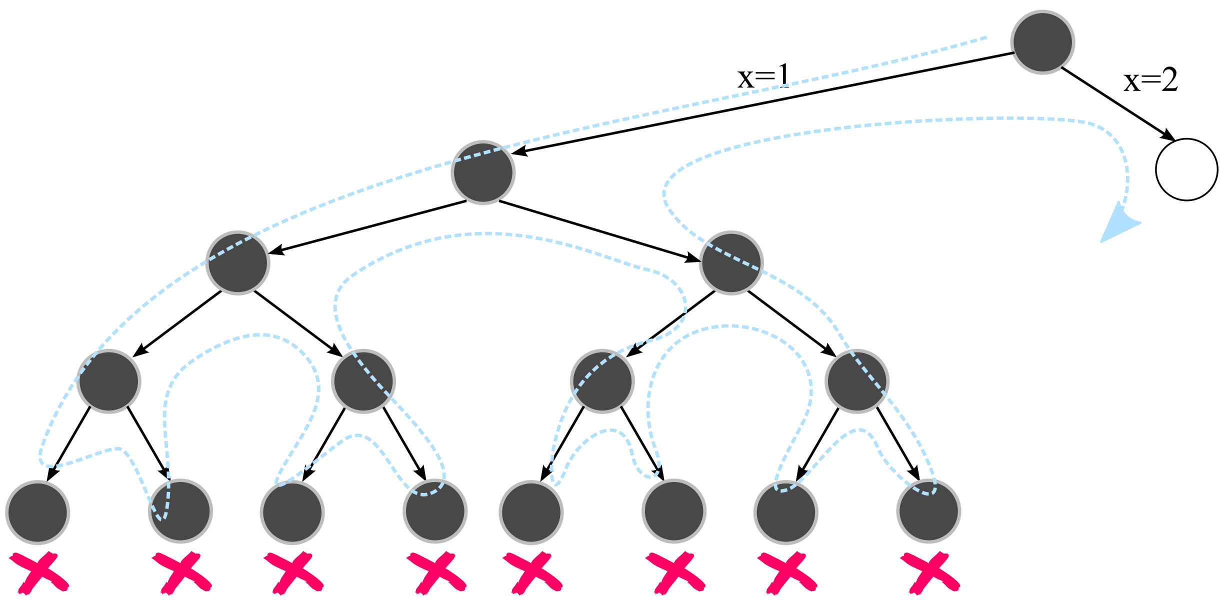


Figure 2: Constraint-based A* inefficiently searching. The blue dotted arrow shows the order of search, which is depth-first in nature. Darkened nodes have been expanded, while unshaded ones are unexpanded but in the queue. The red x's indicate that those full assignments failed the *consistent?(…)* check. The algorithm will search the entire *x=1* subtree. If we know that *x=1* can never hold however, this wastes a lot of time, and causes a lot of calls to the *consistent?(…)* method.

Can we do better? Yes we can!

INTRODUCING: CONFLICT-DIRECTED A*

Conflict-directed A* aims to solve the above problem, by not repeatedly exploring states that all won't work for the same reason. The notion of "states that won't work for the same reason" is captured by a **conflict**. Intuitively, a conflict is a partial set of assignments to the decision variables, not all of which can hold at once. We often denote a conflict as $\{x_i = v_i, \ldots\}$.

An example of a conflict may be that "you cannot have your cake and eat it too." You can have cake, you can eat it, but you can't do both. So, it would be futile for a search algorithm to examine such states as "has cake, eats cake, likes chocolate" or "has cake, eats cake, likes to bake"– because these states *manifest* the conflict. Some states, such as "doesn't have cake, likes ice cream" or "doesn't eat cake, eats cereal" – provably *resolve* the conflict by avoiding it. If one of these states holds, it guarantees the conflict doesn't hold. Other partial states such as "has chocolate" neither manifests the conflict nor resolves it. However in our search, we can be clever and choose successor states such as "has chocolate, doesn't have cake" and "has chocolate, doesn't eat cake" to guarantee that these successors will resolve the conflict. This uses the notion of *constituent kernels* (We'll elaborate all of this a bit shortly).

In our example above where constraint-based A* was repeatedly exploring problematic areas, our conflict was that *x=1*; in other words, *x*=1 doesn't hold in any solution. Suppose we can discover this conflict early in the search, say, after we reach the first full assignment:

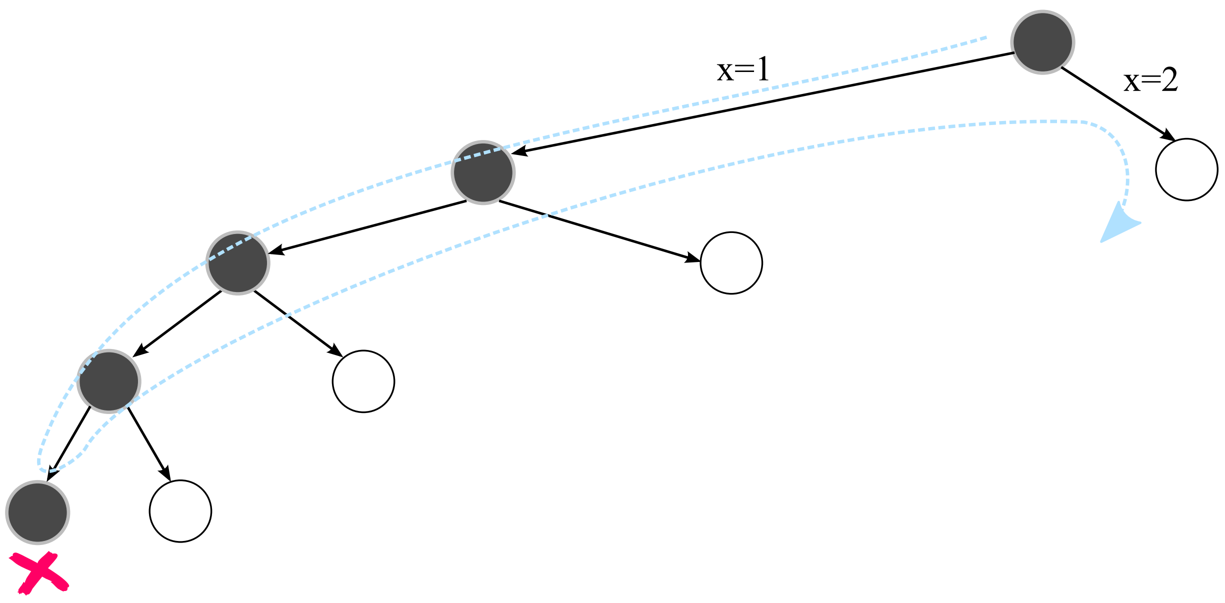


Figure 3: Conflict-directed A* discovers a conflict that *x=1* cannot hold early in the search, after the first consistency check. Because of this, the algorithm "leaps" over the rest of the *x=1* subtree, knowing that any state there will be problematic, and immediately moves on to the *x=2* subtree. This can save a lot of time.

If we can do that, then we can immediately leap over any problematic subtrees in our search that exhibit *x=1*. Why waste time there if we know there can't be any solutions? This is the intuition behind conflict-directed A*.

But how do we get these conflicts in the first place? Technically, that's not the job of conflict-directed A*. We assume they're given to us on a silver platter by the consistency checker. So we modify the *consistent?(…)* function above. Instead of just returning True / False, we now expect it to return a tuple (*is_consistent, conflict)*, where *is_consistent* is True or False as before, and *conflict* is a conflict. In practice, this can be done in a number of ways; SAT-based consistency checkers that use unit propagation can trace back support to get conflicts. Oftentimes, these conflicts can be extracted efficiently with little overhead.

Once we have conflicts, how can we use them to guide our search? In a few ways. The most obvious way is that, once we discover a conflict, remove any states in *Q* that manifest the conflict. That guarantees we won't waste time searching over future enqueued nodes that are definitely problematic.

A second clever technique used by conflict-directed A* is to "leap over" the conflicts. This is accomplished by converting the conflict into what is known as a **constituent kernel**. Suppose we have three decision variables, x_1 and x_2 and x_3, each with domain {1, 2, 3}. Now suppose our *consistent?*(…) method discovers the conflict {x_1=2, x_2=3}. Logically, this means:

$$\neg(x_1 = 2 \wedge x_2 = 3)$$

If we apply De Morgan's law, we get:

$$\neg(x_1 = 2) \vee \neg(x_2 = 3)$$
$$(x_1 \neq 2) \vee (x_2 \neq 3)$$
$$(x_1 = 1 \vee x_1 = 3) \vee (x_2 = 1 \vee x_2 = 2)$$

and finally

$$x_1 = 1 \vee x_1 = 3 \vee x_2 = 1 \vee x_2 = 2$$

This last line represents what is known as a *constituent kernel*. It's a disjunction (an "or" of assignments). Each assignment in the constituent kernel is guaranteed to resolve the conflict. In other words, if $x_1=1$, it is guaranteed that $\neg(x_1 = 2 \land x_2 = 3)$ is satisfied. For the conflict to be resolved, at least one of the constituent kernels must hold.

Suppose in our search, we encounter a state like $x_3 = 1$ that neither manifests the above conflict, nor resolves it. What can we do? Well, we can guarantee that any successor states stemming from this state will resolve the conflict using its constituent kernel. We can conjoin this state with each of the assignments from the constituent kernel, and be guaranteed that the resulting state resolves the conflict. For example, one such successor state could be $\{x_3 = 1, x_1 = 1\}$ – which is guaranteed to avoid the conflict. By selecting the neighbors / successors of this state in our search in this way, we can "leap" over conflicts – our search is guaranteed to avoid them.

This is exactly what the following split on conflict function does:

```
split_on_conflict(assignment, $\gamma$)
return [(assignment $\cup$ $c_i$) for $c_i$ in constituent_kernel($\gamma$), if it's self-consistent]
```

(The "self consistent" part of the above just makes sure we don't have anything silly, like $\{x_1=2, x_1=3\}$).

Putting all these ideas together, here is some pseudo code for conflict-directed A*:

```
CONFLICT-DIRECTED A*
Q = [] # priority queue
Add {} to Q
$\Gamma$ = [] # conflicts
expanded = []
while Q isn't empty:
    assignment ← pop best from Q
    Add assignment to expanded
    if assignment is full assignment to decision variables: # potential goal
        is_consistent, conflict = consistent?(assignment): # searches over non-decision variables
        if is_consistent:
            return assignment
        else:
            Add conflict to $\Gamma$ # some clever tricks here to save space
            Remove anything from Q that manifests conflict

    else: # partial assignment to decision variables
        if assignment resolves all conflicts in $\Gamma$:
            $x_i$ = some decision variable not assigned in assignment
            neighbors = split_on_variable(assignment, $x_i$)
        else if assignment doesn't resolve some conflict $\gamma$ in $\Gamma$:
            neighbors = split_on_conflict(assignment, $\gamma$)
        Add each $x_k$ in neighbors to Q if not in expanded
# If we get here, we've expanded all possibilities to no avail!
return NOSOLUTION
```

In a nutshell, this algorithm behaves just like constraint-based A*, except that it gets conflicts from the consistency checker and stores them in a list Γ. Whenever it learns a conflict, it removes from Q any state manifesting it. When expanding a state and getting its neighbors / successors, we always guarantee that the generated neighbors resolve conflicts.

How well does this work in practice? It depends on a few things. If we can extract better, more minimal conflicts that are smaller in size – we can leap over larger and larger sections of state space and possibly gain huge advantages. If, however, the cost to extract a conflict is very high, the performance could suffer. These need to be traded off.

TIPS & TRICKS

There are plenty of tips & tricks for conflict directed A* that aren't discuss here. For instance, we only want to keep the "minimal" conflicts in Γ, such that no conflict in it is a subset of any other conflict in it. That would be redundant information, and we can save space (and efficiency) by only storing the smallest conflicts.

There are also other tricks we can play to minimize the queue size of conflict-directed A* in circumstances where we have a **mutually preferentially-independent** (MPI) reward function (which we actually do have). Please read the full paper about conflict-directed A* for details.