Flexible Planning by Leximin Fuzzy Constraint Satisfaction

Ian Miguel[1], Qiang Shen[2], and Peter Jarvis[3]

[1] Dept. Computer Science, University of York, UK
[2] Division of Informatics, University of Edinburgh, UK
[3] AI Center, SRI International, USA

Abstract. The flexible planning problem extends Boolean planning, incorporating preferences into operators and goals. This can be re-represented as a fuzzy constraint satisfaction problem, solutions to which trade plan length versus compromises made. However, Flexible Graphplan (FGP), a recent extension of the Graphplan planner that performs flexible planning, aggregates satisfaction degrees via the minimum operator. Thus, it suffers from 'drowning', where one poor action swamps all others. We show how leximin ordering, which refines maximin ordering to enable lexicographical comparisons between ordered vectors of satisfaction degrees, can be used to extend FGP. This allows a detailed comparison of candidate plans so that important compromise solutions are not missed. Experimental results demonstrate the success of this research.

1 Introduction

It is well established that classical AI planning can be represented as a type of constraint satisfaction problem (CSP) [5]. Inflexible operators, whose applicability in a given situation is a Boolean issue, and imperative goals, all of which must be satisfied, make translation straightforward. However, this rigidity impedes the application of planning techniques to real problems, many of which contain soft constraints and are inherently open to compromise.

The *flexible planning problem* [21], based on the fuzzy constraints work of Dubois *et al* [6], supports such soft constraints. It extends the typed STRIPS problem [10], using fuzzy relations to express possible compromises in both operators and goals. A range of plans can be produced for a given problem, trading plan length versus compromises made. Flexible Graphplan (FGP, [21]) was developed to solve flexible planning problems. FGP is based on Graphplan [3], using a hierarchical decomposition of the planning graph and restriction/relaxation based dynamic flexible constraint satisfaction [20] for plan synthesis.

The standard *min* operator to order solutions suffers from 'drowning' [24]: one poor action obscures levels of compromise of many others. It also restricts the number of compromise plans FGP can produce, so important compromise plans may be missed. Leximin ordering [7] is described as a remedy and used as the foundation for Leximin Flexible Graphplan (LFGP), a new flexible planner.

2 The Flexible Planning Problem and Flexible Graphplan

A flexible planning problem, Ψ, consists of a 4-tuple, $\langle \Phi,$ O, I, $\Gamma \rangle$, denoting sets of plan objects, flexible operators, initial conditions consisting of flexible propositions, and flexible goal conditions. Boolean propositions are replaced by flexible propositions, $(\rho\ \phi_1, \phi_2, ..., \phi_j\ k_i)$, where $\phi_i \in \Phi$ and k_i is an element of a totally ordered set, K, which denotes the subjective degree of truth of ρ. K is composed of a finite number of membership degrees, $k_\perp, k_1, ..., k_\top$. A flexible proposition is described by a *fuzzy relation* [22], R, defined by a membership function $\mu_R(.) : \Phi_1 \times \Phi_2 \times ... \times \Phi_j \rightarrow K$, where $\Phi_1 \times \Phi_2 \times ... \times \Phi_j$ is the Cartesian product of the subsets of Φ allowable at this place in the proposition. The end points of K capture Boolean propositions, with $k_\perp, k_\top \in K$ indicating total falsehood and total truth. For propositions which only take a truth value of $k_\perp$ or $k_\top$, the Boolean style of $\neg(\rho\ \phi_1, \phi_2, ..., \phi_j)$ and $(\rho\ \phi_1, \phi_2, ..., \phi_j)$ is adopted.

A Flexible operator, $o \in$ O, is described by a fuzzy relation mapping from the precondition space onto a totally ordered *satisfaction scale*, L and a set of flexible effect propositions. L represents the degree of compromise required to apply an operator in the current situation. L is also composed of a finite number of membership degrees, $l_\perp, l_1, ..., l_\top$. The endpoints, $l_\perp \in L$ and $l_\top \in L$ denote a complete lack of satisfaction and complete satisfaction. Each $o \in$ O consists of a set of disjoint conditional clauses, Σ. Each $\sigma \in \Sigma$ is a triple $\langle \Theta,$ E, $k_i \rangle$ denoting a conjunction of flexible preconditions, a conjunction of flexible effect propositions and the satisfaction degree of this operator given these preconditions. Each $\theta \in \Theta$ has the form $(\rho\ \phi_1, \phi_2, ..., \phi_j\ \tau\ \kappa\ )$, where τ is a precondition operator with argument set κ. Allowed precondition operators encompass equality, inequality, ranges and sets of discrete truth degrees. Each σ_i maps a subset of the space of preconditions to a set of effects and a satisfaction degree in L. Again, when dealing with preconditions which only ever take a truth value of $k_\perp$ or $k_\top$, the Boolean style of $\neg(\rho\ \phi_1, \phi_2, ..., \phi_j)$ and $(\rho\ \phi_1, \phi_2, ..., \phi_j)$ is adopted for θ.

A flexible plan goal $\gamma \in \Gamma$ maps from the space of flexible propositions to L. Preconditions are defined exactly as those used in flexible operators. A plan satisfaction degree combines both operator and goal satisfaction degrees. A significant advantage of this formalism is in the over-constrained case: when Boolean planning returns no plan at all, compromise plans may still be constructed.

Figure 1 presents an example. The goals (Figure 2) are to transport the two packages to city c_4 and the guard to c_3. The two packages have different values, described by: $(\textit{valuable}\ \textit{pkg}_1\ k_2)$ and $(\textit{valuable}\ \textit{pkg}_2\ k_1)$. Since $\textit{pkg}_2$ is of a lower value, one option is to choose not to transport it, but at a very low satisfaction degree. A further possible compromise is that the guard may be left at c_2 or c_4. Transportation is by truck using the `Drive-truck` flexible operator. Mountainous roads should be avoided if possible, otherwise satisfaction degree l_1 is assigned. `Load-truck` (Figure 3) and `Unload-truck` consider the value of the package and the presence of a guard. If the package is not valuable, the guard's presence is immaterial - satisfaction degree $l_\top$ is assigned. If the package is valuable, loading/unloading without a guard results in satisfaction degree l_2. Further (Boolean) operators allow the guard to board and leave the truck.

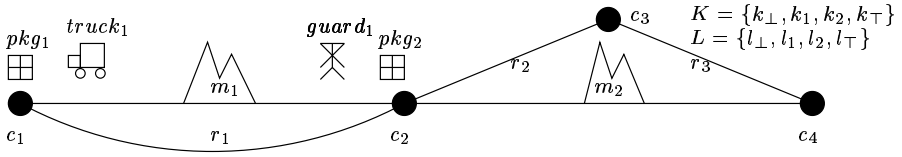

Fig. 1. Example Problem: The r_i, m_i are main/mountainous roads, the c_i are cities.

2.1 Plan Synthesis via Dynamic Fuzzy CSP

FGP aggregates action and goal satisfaction degrees using the idempotent *min* operator [21], which enables classical k-consistency techniques to be straightforwardly extended to fuzzy CSP [6]. Hence, flexible plan *quality* combines satisfaction degree and length, where the shorter of two plans with equal satisfaction degrees is better. Compromise plans with sub-optimal satisfaction degrees trade action and goal satisfaction against reduced length. FGP constructs and analyses a Flexible Planning Graph. It extends the graph until the goals are found, then searches the graph for a plan. A planning graph is divided into a number of *levels*. $Level_i$ contains actions ($actions_i$) and propositions ($propositions_i$). $Level_0$ contains propositions which capture the initial problem state.

Mutual exclusion constraints added to the planning graph greatly reduce search. Exclusive propositions express a different truth degree for the same proposition or all ways of creating one are exclusive of all ways of creating the other. Two actions are mutually exclusive for three reasons as per Graphplan: *Inconsistent Effects* - an effect of one action expresses a different truth degree from an effect of the other for the same proposition; *Interference* - an effect of one action expresses a different truth degree from a precondition of the other for the same proposition; *Competing Needs* - mutually exclusive preconditions. When extending the planning graph to $level_i$ from $level_{i-1}$ (Figure 4), each clause of each operator is instantiated in all possible ways to mutually consistent nodes in $propositions_{i-1}$. An action with the associated satisfaction degree for this instantiation is added to $actions_i$. A `Noop` records the persistence of a proposition.

A node in $propositions_i$ is viewed as a fuzzy CSP variable, its domain comprising the set of nodes in $actions_i$ which assert this proposition as an effect. A unary *preference* constraint is constructed from the domain elements and their associated satisfaction degrees. Consider $level_g$ which contains the goal propositions. Each set of supporting actions specifies (via preconditions) a sub-problem at $level_{g-1}$. Solutions to these sub-problems specify new sub-problems at $level_{g-2}$,

(**goal** pkg_1-destination
(**when** $(at\ pkg_1\ c_4)\ l_\top$))

(**goal** pkg_2-destination
(**when** $(at\ pkg_2\ c_2)\ l_1$)
(**when** $(at\ pkg_2\ c_4)\ l_\top$))

(**goal** $guard_1$-destination
(**when** $(at\ guard_1\ c_2)\ l_2$)
(**when** $(at\ guard_1\ c_4)\ l_2$)
(**when** $(at\ guard_1\ c_3)\ l_\top$))

Fig. 2. Flexible Goals for the Example Problem.

```
(operator Load-truck
 params (?t truck) (?p package) (?d location) (?g guard))
 {when (preconds (at ?t ?l) (at ?p ?l) (valuable ?p <= k1))
  (effects (not (at ?p ?l)) (on ?p ?t)) l⊤}
 {when (preconds (at ?t ?l) (at ?p ?l) (on ?g ?t) (valuable ?p >= k2))
  (effects (not (at ?p ?l)) (on ?p ?t)) l⊤}
 {when (preconds (at ?t ?l) (at ?p ?l) (not (on ?g ?t)) (valuable ?p >= k2))
  (effects (not (at ?p ?l)) (on ?p ?t)) l2})
```

Fig. 3. The `Load-truck` Operator

etc. A sub-problem sequence associated with a graph level is considered as a restriction/relaxation-based dynamic flexible CSP (rrDFCSP) ([18], Figure 5).

FGP solves these sequences with Flexible Local Changes (FLC, [20]), identifying a sub-graph connecting a set of goals to the initial conditions. Assuming that the solutions to the sub-problem at $level_i$ intersect, there will be similarities between sub-problems at $level_{i-1}$. FLC exploits these similarities, re-using effort applied to the previous problem in the sequence when solving the current sub-problem. Also, effort used in levels visited in a failed attempt at plan extraction is re-used when the graph is expanded and a further extraction attempted.

2.2 Example Problem: Plans Synthesised by FGP

FGP synthesises three plans for the example. The first, with four steps and a satisfaction degree of l_1, transports pkg_1 to city 4, ignoring pkg_2 and the guard:

1. LOAD-TRUCK pkg_1 $truck_1$ (l_2)
2. DRIVE-TRUCK $truck_1$ c_1 to c_2 via r_1 ($l_\top$)
3. DRIVE-TRUCK $truck_1$ c_2 to c_4 via m_2 (l_1)
4. UNLOAD-TRUCK pkg_1 $truck_1$ (l_2)

The second (6 steps, l_2) transports both packages, avoiding mountainous roads.

1. LOAD-TRUCK pkg_1 $truck_1$ (l_2)
2. DRIVE-TRUCK $truck_1$ c_1 to c_2 via r_1 ($l_\top$)
3. LOAD-TRUCK pkg_2 $truck_1$ ($l_\top$)
4. DRIVE-TRUCK $truck_1$ c_2 to c_3 via r_2 ($l_\top$)
5. DRIVE-TRUCK $truck_1$ c_3 to c_4 via r_3 ($l_\top$)
6. UNLOAD-TRUCK pkg_1 $truck_1$ ($l_\top$)
   UNLOAD-TRUCK pkg_2 $truck_1$ ($l_\top$)

The compromise-free plan has ten steps. It first fetches the guard before loading pkg_1, then uses the major roads to deliver both packages and the guard.

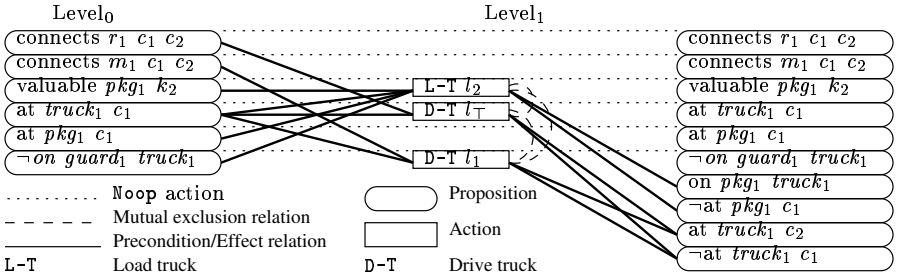

Fig. 4. Partial Flexible Planning Graph for the example. Only mutual exclusion relations between non-`Noop` actions are shown.

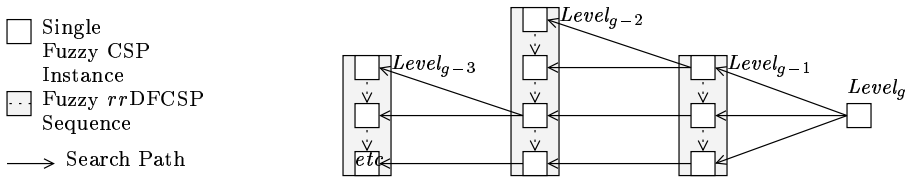

Fig. 5. Plan Extraction as a Hierarchy of Fuzzy rrDFCSPs.

3 The Drowning Effect and Leximin Ordering

Min aggregation suffers from *drowning* [24]: a low satisfaction degree from one assignment 'drowns' several others whose satisfaction degrees are not reflected in the overall value. The 4-step plan in section 2.2 travels from c_1 to c_2 via main road r_1, but FGP cannot distinguish this from a plan which chooses mountainous road m_1. Driving across the mountainous road m_2 (step 3, l_1), drowns this decision. FGP attempts to avoid drowning by selecting goals and actions with the highest satisfaction degree first [21]. This works here since a single non-`Noop` action per step is required. Generally, an optimal *set* of actions must be chosen at each step to avoid drowning. A *min*-based planner cannot guarantee this.

The *min* operator also limits the number of compromise plans FGP can produce. Given a satisfaction scale with ℓ elements, a *min* planner can produce *at most* $\ell - 1$ plans, one per satisfaction degree above $l_\perp$. Consider the following:

1. `LOAD-TRUCK` pkg_1 $truck_1$ (l_2)
2. `DRIVE-TRUCK` $truck_1$ c_1 to c_2 via r_1 ($l_\top$)
3. `LOAD-TRUCK` pkg_2 $truck_1$ (l_3)
4. `DRIVE-TRUCK` $truck_1$ c_2 to c_4 via m_2 (l_1)
5. `UNLOAD-TRUCK` pkg_1 $truck_1$ ($l_\top$)

   `UNLOAD-TRUCK` pkg_2 $truck_1$ ($l_\top$)

FGP discards this plan because it has the same satisfaction degree as the 4-step plan in section 2.2. Intuitively, however, it is a significant improvement since pkg_2 is delivered. An 8-step plan can improve on FGP's 6-step plan by collecting $guard_1$ and taking him to c_3 after unloading the packages.

Given two vectors of satisfaction degrees from two plans, *leximin* ordering is a lexicographic ordering between the two vectors sorted in ascending order [7]. Hence, $\boldsymbol{v}_1 = \{l_1, l_2, l_2\}$ is greater (better) than $\boldsymbol{v}_2 = \{l_1, l_1, l_2\}$ ($\boldsymbol{v}_1 >_{lex} \boldsymbol{v}_2$), but $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are indistinguishable by *min*. Flexible plan quality now combines the associated vector of satisfaction degrees with its length. The shorter of two plans with equivalent satisfaction degree vectors is better. Unequal length vectors are compared by extending the shorter with $l_\top$ entries, since only satisfaction degrees below $l_\top$ represent compromises. The refined notion of plan quality supports our intuition that the 5-step plan above is better than FGP's 4-step plan. Compare:

4-step: $\{l_1$ (pkg_2 goal), l_1 (`DRIVE-TRUCK`), l_2 ($guard_1$ goal), l_2(`LOAD-TRUCK`), l_2(`UNLOAD-TRUCK`)$\}$
5-step: $\{l_1$ (`DRIVE-TRUCK`), l_2 ($guard_1$ goal), l_2 (`LOAD-TRUCK`), l_2(`UNLOAD-TRUCK`)$\}$

Clearly, the 5-step plan is better by the leximin ordering, hence a planner based on this ordering would be able to present it as a possible compromise.

A leximin-optimal solution to a fuzzy CSP violates the fewest constraints [9], translating to the fewest compromises in a plan. Hence, leximin ordering removes drowning so that the satisfaction degree of every action is important. It also ensures that optimal action combinations are selected for each sub-problem.

a)

```
1.  Procedure counterCheck(counters, entry)
2.    counters[entry]--
3.    If counters[entry] = -1
4.      For i = 1 To entry - 1
5.        If counters[i] != 0 Return true
6.    Else If counters[entry] = 0
7.      For i = entry + 1 To l⊤
8.        If counters[i] > 0 Return true
9.        If counters[i] < 0 Break
10.     For i = 1 To entry - 1
11.       If counters[i] != 0 Return true
12.   Return false
```

b)

```
1.  Procedure forwardCheck(x_i, d_i, c, counters)
2.    wipeout ← true
3.    D_j ← getDomain(otherVar(c, x_i))
4.    Foreach d_j ∈ D_j
5.      l_j ← getSatDegree(d_j)
6.      If Not counterCheck(counters, l_j) Prune d_j
7.      Else If test(c, d_i, d_j)
8.        wipeout ← false
9.      Else Prune d_j
10.   Return wipeout
```

Fig. 6. The `counterCheck()` and `forwardCheck()` procedures.

3.1 Solving Leximin Fuzzy CSPs

We solve Leximin Fuzzy CSP sequences via a combination of branch and bound and forward checking [12]. Efficient backtracking means tracking when the satisfaction degree vector for the current assignment, $\boldsymbol{v}_\beta$ is lexicographically less than or equal to that of the best assignment, $\boldsymbol{v}_\alpha$. We use an array of *counters*, with one entry per element of L. Each element records the difference between the number occurrences of the associated satisfaction degree in $\boldsymbol{v}_\alpha$ and $\boldsymbol{v}_\beta$. The array is initialised by counting the occurrences of each satisfaction degree in $\boldsymbol{v}_\alpha$, then decrementing each entry for each occurrence of the corresponding satisfaction degree in $\boldsymbol{v}_\beta$. Hence, if $\boldsymbol{v}_\alpha = \{l_2, l_3, l_\top\}$ and $\boldsymbol{v}_\beta = \{l_2\}$, *counters* would become {0, 0, 0, 1, 1}. The first entry is always 0, since $l_\bot$ is never tolerated.

As assignments are made, `counterCheck()` (Figure 6*a*) updates *counters* and provides an incremental test for the need to backtrack. The test, also used during initialisation of *counters*, is as follows: A negative count (line 3) means that more entries of the corresponding satisfaction degree exist in $\boldsymbol{v}_\beta$ than in $\boldsymbol{v}_\alpha$. This is significant if all counts for lower satisfaction degrees are 0. A 0 count (line 6) indicates an equal number of entries of the corresponding satisfaction degree in $\boldsymbol{v}_\alpha$ and $\boldsymbol{v}_\beta$. This is significant if counts for lower satisfaction degrees are all 0 and the first non-zero count for a higher satisfaction degree is negative. This scheme is improved by ignoring $l_\top$. Since $\boldsymbol{v}_\alpha$ represents a complete assignment, it is maximally sized. If $\boldsymbol{v}_\alpha$ and $\boldsymbol{v}_\beta$ cannot be distinguished via the lower satisfaction degrees, then $\boldsymbol{v}_\alpha$ must have a greater or equal number of $l_\top$ entries.

Figure 7 presents `BBFCLex`. Throughout, $assignment_\alpha$ (best known solution) and $assignment_\beta$ (current variable assignment) have global scope. Note that only the unary constraints are fuzzy. Generally, forward checking with fuzzy binary constraints generates satisfaction degrees according to how well each potential assignment satisfies the constraint. This information must be stored and added to $\boldsymbol{v}_\beta$ when the associated domain element is eventually assigned. A Boolean constraint check results in either $l_\bot$ (element pruned), or $l_\top$. Hence, the only information recorded, beyond the static consistency degree of a domain element, is whether or not it is currently pruned. This results in space and time savings.

The satisfaction degree of the selected domain element is used to update *counters* (lines 7-9). If adding this satisfaction degree to $\boldsymbol{v}_\beta$ does not cause

```
1.  Procedure BBFCLex(X, D, C)
2.   If X = ∅ assignment_α ← assignment_β, Return
3.   counters←initCounters( assignment_α, assignment_β)
4.   Select and Remove x_i ∈ X, D_i ∈ D
5.   subC ← getConnected(x_i, X, C)
6.   Foreach d_i ∈ D_i
7.    dCounters ← copy(counters)
8.    l_i ← getSatDegree(d_i)
9.    If counterCheck(dCounters, l_i) And
          ∀c ∈ subC forwardCheck (x_i, d_i, c, copy(dCounters))
10.    Assign x_i ← d_i
11.    BBFCLex(X, D, C)
```

Fig. 7. The `BBFCLex()` Procedure

backtracking, $x_i \leftarrow d_i$ is propagated via `forwardCheck()` (Figure 6*b*) through a constraint to the domain, D_j, of an unassigned variable. It is first determined whether the unary consistency, l_j, of $d_j \in D_j$ in combination with l_i and $\boldsymbol{v}_\beta$ is feasible (lines 5 - 6). If so, $\{x_i \leftarrow d_i, x_j \leftarrow d_j\}$ is evaluated (line 7). If the pair is excluded, d_j is pruned. This process is repeated for all variables connected to x_i by a constraint. If all elements of any D_j are pruned $x_i \leftarrow d_i$ is infeasible. Otherwise the search proceeds (`BBFCLex`, lines 10-11).

The description of `BBFCLex` contains two simplifications. First, a backtracking mechanism is necessary to undo the propagation made by `forwardCheck()`. This is implemented efficiently by associating pruning with a search depth, and restoring domain elements as backtracking occurs. Second, before a first solution is found all assignments except those with a satisfaction degree of $l_\perp$ are permitted to obtain an initial solution to use as a bound.

BBFCLex solves leximin *rr*DFCSPs effectively by re-using information from the solutions to the previous problems in a dynamic sequence. This is achieved cheaply and effectively by using previous solutions as a heuristic [18]. Given alternative assignments with equivalent satisfaction degrees, the heuristic selects the one which agrees with the previous solution, if possible.

4 Leximin Flexible Graphplan

Leximin Flexible Graphplan (LFGP) retains the basic 2-phase structure of plan synthesis. Indeed, the process of flexible graph expansion is unchanged. Hence, despite the extra flexible ability, the size of the graph remains tractable [18]. Plan extraction also operates in the same basic way as described in section 2, as the solution of a hierarchy of *rr*DFCSPs, each now solved using BBFCLex.

Leximin ordering is also used to determine when to abandon the current plan, $plan_\beta$ as necessarily no better than the best plan known, $plan_\alpha$. Vectors v_α and v_β, introduced in section 3.1, now represent the satisfaction degree vectors of $plan_\alpha$ and $plan_\beta$. We include the satisfaction degrees of assignments made during the solution of a sub-problem in v_β. Hence, the usage of v_α and v_β remains the same in BBFCLex. Backtracking is now when the current partial *plan* cannot better $plan_\alpha$, rather than with respect to the best known local solution.

```
1.  Procedure LFGP(ψ)
2.    graph ← graph-initialise(ψ)
3.    lvl ← 0
4.    While Not compromise-free(plan_α)
5.      lvl ← lvl+1
6.      extend(graph)
7.      goalSets ← getGoalSets (graph, ψ)
8.      sort(goalSets)
9.      Foreach goalSet ∈ goalSets
10.       If satVector(goalSet) >_lex v_α
11.         extract(lvl, goalSet)

1.  Procedure extract(lvl, goalSet)
2.    plan_β ← plan-init(lvl, satVector(goalSet))
3.    extractLevel(lvl, getPropNodes(goalSet))

1.  Procedure extractLevel(lvl, subGoals)
2.    If lvl = 0 And matchInitConds(subGoals)
3.      plan_α ← plan_β
4.      Return ∅
5.    memoset ← getMemoset(graph, lvl, subGoals)
6.    If memoset ≠ ∅ Return memoset
7.    BBFCLex_l ← getSolver(graph, lvl)
8.    subProb ← translate(subGoals)
9.    memoset ← solve(BBFCLex_l, subProb)
10.   If memoset ≠ ∅ recordMemo(memoset, lvl)
11.   Return memoset
```

Fig. 8. The LFGP(), extract() and extract-level() Procedures

Figure 8 presents the top level of LFGP. The planning graph (*graph*), $plan_\alpha$ and $plan_\beta$ have global scope. We extend the planning graph and search for plans until a compromise-free plan is encountered. At each level, the *goalSets* of mutually consistent proposition nodes which match the plan goals are determined (line 7). Since flexible goals associate a satisfaction degree with different propositions, depending on their degree of satisfaction, a vector of satisfaction degrees is generated for each *goalSet*. The *goalSets* are sorted in descending lexicographic order (line 8) before attempting to find a plan containing them (line 10). Sorting increases the likelihood of finding a good plan early, to bound future search.

The extract() and extractLevel() procedures (Figure 8) control the solution of the hierarchy of rrDFCSPs created from the planning graph. The former initialises a partial plan with the satisfaction vector of the current *goalSet* (line 2) before calling the latter. The base case is when $level_0$ of the planning graph is reached (lines 2-4). If the subgoals now match the initial conditions (line 2), a new best plan has been found and becomes a new bound (line 3).

At all other levels, we first check whether a previously recorded memoset (a type of nogood, see section 4.1) shows that the current branch cannot lead to a new best plan. If so, this branch of the search is pruned, and the memoset is returned to the level above for propagation (line 6). Otherwise, a new sub-problem in the rrDFCSP for the current level is generated and solved, returning a new memoset to be recorded and propagated (lines 7-11). BBFCLex() and extractLevel() are mutually recursive: once the former has found a solution, it invokes the latter to search the next level down.

4.1 Nogood Recording in a Hierarchy of Leximin Fuzzy rrDFCSPs

In order to increase efficiency, it is necessary to prune search branches that necessarily lead to failure. A mutually-consistent set of nodes in $propositions_i$ is *conjunctively unsupportable* if all combinations of nodes in $actions_i$ that assert these propositions are either directly inconsistent or their own preconditions are unsupportable. *Memoisation* is the process of recording this type of nogood,

known as a *memoset* [3]. If a memoset (or a superset) is encountered again at the level at which it was recorded, the current search branch is pruned immediately.

Naively, we can record the entire set of nodes currently considered at $propositions_i$ once the fact that they are unsupportable has been established. However, it is likely in general that certain propositions are irrelevant to the failure. As described in [21], FGP employs more informed methods of memoisation to improve plan extraction significantly. Most importantly, memoset information is propagated back up through the planning graph such that the memosets of the children of a sub-problem are combined to generate the memoset for their parent. We describe how this scheme can be extended to LFGP.

Leximin Memoset: A set of propositions, p_m, a satisfaction degree vector, v_m, and an associated graph *level*. v_m records a lexicographic upper bound on the satisfaction degree vector of a sub-plan which includes p_m at *level*.

When `extractLevel()` is invoked to try and support a set of sub-goals at $level_i$, we take the subset of the memosets recorded at $level_i$ whose proposition sets are in turn a subset of the sub-goals. For each such memoset, we append its associated satisfaction degree vector to v_β. From the definition above, if the (sorted) resulting vector is lexicographically less than v_α, then $plan_\beta$ cannot be extended to a plan lexicographically better than $plan_\alpha$.

Leximin memosets are generated both locally and by propagation. Locally, BBFCLex is modified to record a *conflict set* at a dead end in the search. This consists of the variable whose domain elements have all been pruned and the set of variables on which this pruning *depends*:

Dependency: A *direct* dependency from x_a to x_b, denoted $\mathrm{dep}(x_a, x_b)$, exists if an assignment to x_b causes the pruning of one or more elements of D_a. An *indirect* dependency exists from x_a to x_c if a chain of variables $y_0, ..., y_k$ exist such that: $\mathrm{dep}(y_0, x_c) \wedge \forall i\ \mathrm{dep}(y_{i+1}, y_i) \wedge \mathrm{dep}(x_a, y_k)$.

If a sub-problem has no feasible solution, a leximin memoset is generated from the union of all such recorded conflict sets. v_m is found by re-solving the sub-problem without pruning based on v_β. This is feasible because sub-problems are typically small and easy to solve. Removing the effects of v_β is necessary since v_β may (lexicographically) *increase* as well as decrease before the next visit to this level. Domain elements formerly pruned with respect to a lower version of v_β are then reinstated and could form part of a local solution.

Memosets are propagated up the level hierarchy via precondition/effect relations in the planning graph. For each memoset we propagate from $level_i$ to $level_{i+1}$ we identify the set of actions, a_c, at $level_{i+1}$ that, via their preconditions, required the memoset propositions to be supported at $level_i$. Given a solution to a sub-problem at $level_{i+1}$, leading to a conflict and therefore a memoset at $level_i$, the associated *culprit set* of propositions is the subset of the sub-problem variables whose assignments form the set a_c. A *propagated conflict set* is generated from the union of the culprit set and the dependencies of all of its elements. An

associated satisfaction degree vector is formed by appending the vector of satisfaction degrees associated with the elements of a_c to the vector v_m associated with the memoset propagated to find the culprits.

A leximin memoset is generated from such propagation as follows. p_m is the union of all propagated conflict sets discovered during the solution of the current sub-problem. v_m is the lexicographically greatest of two components. The first is the set of vectors associated with each propagated conflict set. The second is found by re-solving the sub-problem without pruning based on $plan_\beta$, such that the solution contains at least one assignment that would be pruned with respect to $plan_\beta$. This ensures that the vector returned by re-solution is not simply a sub-vector of a vector associated with a propagated conflict set.

5 Experimental Results

To the best of our knowledge, FGP and LFGP are the only planners capable of solving the flexible planning problems defined in section 2. Therefore, we compare their performance on a set of problems. Since LFGP performs a superset of the work done by FGP, we cannot expect to see improved run-times. Rather, we must weigh what we gain, the discovery of important compromise plans that might be missed by FGP, against the increased search effort. We also test LFGP*, a version of LFGP without the ability to propagate memosets up the level hierarchy, in order to test the efficacy of more informed memoisation. Results with no memoisation at all were relatively so poor that they are not included.

Fifteen logistics problems of varying complexity were tested. The length of the compromise free plan and the number of satisfaction degrees used were varied to gauge their effects on the relative performance of the planners. Table 1 summarises the results. It is immediately noticeable that, via leximin ordering, LFGP finds a substantial number of plans that the minimum-based FGP misses. Early in the search the extra plans can be synthesised with minimal extra effort. Hence, LFGP further enhances the capability of FGP of producing short, compromise plans very quickly. This provides an attractive 'anytime' behaviour, where the best plan found so far can be returned given a strict time limit.

The behaviour of the three planners diverges at the latter stages of the search. On the harder instances in particular, LFGP typically requires more time than FGP to return the final compromise-free plan. This is unsurprising, given the relative complexity of the search that LFGP must perform. The balance is redressed to a large extent through the use of memoset propagation, which works to produce a dramatic reduction in run-times from LFGP* to LFGP.

These results are indicative of the importance of informed memoisation. The arity of a memoset is crucial to its pruning power. A high arity memoset has less chance of matching a set of sub-goals, often adding overhead. This problem is exacerbated via memoset propagation up the level hierarchy. Hence, we believe that effort devoted to recording minimal-sized memosets would not be wasted.

Prob.	Sat Degrees	Goals	Max Length	Solver	Plan number 1	2	3	4	5	6	7	8	9
Logs1	4	2	7	FGP	0.08	-	0.13	0.23					
				LFGP	0.08	0.11	0.13	0.23					
				LFGP*	0.08	0.11	0.13	0.33					
Logs2	4	3	10	FGP	0.1	-	0.12	-	-	0.41			
				LFGP	0.11	0.14	0.25	0.3	0.51	0.58			
				LFGP*	0.11	0.13	0.22	0.28	0.4	1.5			
Logs3	4	4	12	FGP	0.1	-	-	-	-	0.66	-	1.2	
				LFGP	0.11	0.17	0.3	0.4	0.78	1.0	1.3	2.5	
				LFGP*	0.11	0.16	0.31	0.44	0.93	1.1	1.4	18.77	
Logs4	4	4	14	FGP	0.19	-	0.64	-	-	-	1.22		
				LFGP	0.29	0.4	0.6	0.83	1.65	2.22	2.44		
				LFGP*	0.29	0.4	0.65	1.3	2.64	3.45	16.3		
Logs5	4	5	16	FGP	0.55	-	-	2.23	-	4.6			
				LFGP	0.57	1.0	1.8	5.4	6.2	11.5			
				LFGP*	0.56	1.2	2.56	8.9	9.1	60.21			
Logs6	5	2	7	FGP	0.1	0.13	-	0.27	0.3				
				LFGP	0.1	0.14	0.2	0.32	0.44				
				LFGP*	0.1	0.14	0.19	0.45	1.36				
Logs7	5	2	9	FGP	0.1	-	-	0.25	0.32	0.44			
				LFGP	0.1	0.14	0.2	0.34	0.45	1.2			
				LFGP*	0.1	0.14	0.21	0.48	0.73	14.54			
Logs8	5	3	10	FGP	0.1	-	-	0.27	0.33	-	0.5		
				LFGP	0.1	0.22	0.36	0.56	0.65	2.0	2.1		
				LFGP*	0.13	0.23	0.4	1.17	1.44	64.3	77.62		
Logs9	5	3	11	FGP	0.1	-	-	0.27	0.33	-	0.52		
				LFGP	0.13	0.23	0.34	0.54	0.62	1.37	1.53		
				LFGP*	0.13	0.24	0.4	1.12	1.31	53.7	181.46		
Logs10	5	3	11	FGP	0.1	-	-	0.31	-	0.45	0.6		
				LFGP	0.13	0.18	0.31	0.52	0.6	0.99	7.1		
				LFGP*	0.13	0.18	0.36	0.78	0.9	14.75	549.5		
Logs11	6	3	9	FGP	0.08	0.10	0.12	-	0.3	-	0.41		
				LFGP	0.09	0.11	0.14	0.19	0.31	0.43	0.48		
				LFGP*	0.08	0.11	0.13	0.21	0.28	0.6	0.83		
Logs12	6	3	11	FGP	0.09	0.11	0.14	-	0.32	0.4			
				LFGP	0.09	0.12	0.15	0.18	0.33	0.41			
				LFGP*	0.09	0.11	0.14	0.18	0.53	1.04			
Logs13	6	4	14	FGP	0.09	0.13	-	-	-	0.71	-	-	1.1
				LFGP	0.09	0.13	0.18	0.22	0.46	0.72	0.8	1	1.3
				LFGP*	0.09	0.13	0.18	0.24	1.1	1.24	1.32	1.56	5.26
Logs14	6	4	15	FGP	0.09	0.13	-	-	-	0.65	-	0.8	1.1
				LFGP	0.1	0.14	0.18	0.23	0.5	0.66	0.8	0.92	1.3
				LFGP*	0.09	0.13	0.18	0.24	1.13	1.37	1.44	3.34	12.92
Logs15	6	5	19	FGP	0.2	0.33	-	-	-	2.2	-	4.6	7.2
				LFGP	0.3	0.51	0.68	1.05	1.94	2.41	4.33	7.15	9.9
				LFGP*	0.31	0.43	0.65	2.23	39.1	52.14	100.23	197.95	284.92

Table 1. Cumulative run-times (PIII 750Mhz 256Mb). '-': plan missed by FGP.

6 Related Work

Systematic algorithms for the optimal solution of leximin FCSPs have been investigated previously [8, 17], as has transforming leximin FCSPs into equivalent weighted MAX-CSPs [24]. It may be possible to improve the novel forward checking algorithm presented herein, specialised to the sub-class of leximin FCSPs where the fuzzy constraints are unary, by combining it with these established methods. Recently, classical local consistency algorithms have been extended to non-idempotent operators, such as leximin FCSP [2, 23]. We expect that further efficiency gains can be realised by embedding such techniques in LFGP's search.

Previously, Williamson and Hanks [27] developed the Pyrrhus planner, which replaces simple goal formulae with utility models to introduce a measure of plan quality. However, utilities are not associated with performing individual actions. Other related work [15, 26] incorporates numerically weighted constraints into the planning problem to provide a quantitative means of differentiating plans. Investigating the combination of this with our qualitative method may prove beneficial and remains an important item of future work.

Work on conformant planners considers the possible initial worlds in which a plan can be executed together with the possible outcomes of each action's execution [4, 16]. While such planners model uncertainty as a distribution over the effects of a given operator, the propositions themselves are exclusively *true* or *false*. In our approach, flexible propositions are assigned subjective truth degrees in a formal framework and operators map *deterministically* from the space of flexible preconditions to a set of flexible effects and a satisfaction degree.

Plan synthesis as the solution of a hierarchy of leximin fuzzy rrDFCSPs is an alternative to the compilation of the planning graph to either a single CSP [5] or SAT problem [14]. We have made a preliminary investigation into translating the flexible planning problem into SAT [13]. A full comparison of the relative merits of all these approaches forms a further important area for future research.

7 Conclusions

Leximin Flexible Graphplan is a new version of Flexible Graphplan, based on leximin fuzzy CSP. Experimental results confirm that LFGP provides a more effective means of generating a range of compromise solutions from a given flexible planning problem than FGP. In particular, it removes 'drowning' by a single particularly poor action and its finer grain of comparison allows candidate plans to be generated that would have been missed by FGP. We also introduced the BBFCLex algorithm, an efficient means of solving the restricted class of leximin fuzzy CSPs generated from the flexible planning graph.

Our immediate focus is on improving efficiency, building on the propagation of memosets discussed in section 4.1. This may be achieved through stronger local consistency enforcing, improved nogood recording, and the exploitation of symmetry inherent in the problem structure [19]. In addition, the recent epistemological focus of the planning community centres around time [11, 25] and resources [28]. Flexible planning is complementary. An integrated representation would allow the preferences between resources and deadlines to be expressed.

Acknowledgements The first author is supported by EPSRC Grant GR/N16129. The third author is funded under DARPA's Active Templates program, contracts F30602-97-C-0067 and F30602-00-C-0058 under the supervision of Air Force Research Lab - Rome.

References

1. J. Allen, J. Hendler, and A. Tate. *Readings in Planning*. Morgan Kaufmann, 1990.
2. S. Bistarelli, R. Gennari, and F. Rossi. Constraint propagation for soft constraints: Generalization and termination conditions. In *Proc. 6th CP*, pages 83–97, 2000.
3. A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.
4. A. Blum and J. Langford. Probabilistic planning in the graphplan framework. In *Proc. 5th European Conference on Planning Systems, Durham, UK*, pages 320–332, 1999.
5. M.B. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132:151–182, 2001.
6. D. Dubois, H. Fargier, and H. Prade. Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty. *Applied Intelligence*, 6:287–309, 1996.

7. D. Dubois, H. Fargier, and H. Prade. Refinements of the maximin approach to decision making in a fuzzy environment. *Fuzzy Sets and Systems*, 81:103–122, 1996.
8. D. Dubois and P. Fortemps. Computing improved optimal solutions to max-min flexible constraint satisfaction problems. *European Journal of Operational Research*, 118:95–126, 1999.
9. H. Fargier. *Problèmes de satisfaction de constraintes flexibles: application à l'ordonnancement de production.* PhD thesis, Université Paul Sabatier, Toulouse, 1994.
10. R. Fikes and N. Nilsson. A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.
11. M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Technical report, University of Durham, 2001.
12. R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
13. P. Jarvis, I. Miguel, and Q. Shen. Flexible blackbox: Preliminary results. *Proc. AAAI-2000 Workshop on Representational Issues for Real-World Planning Systems*, pages 43–49, 2000.
14. H. Kautz and B. Selman. Blackbox: Unifying sat-based and graph-based planning. In *Proc. 16th IJCAI*, pages 318–325, 1999.
15. H. Kautz and J. Walser. State-space planning by integer optimization. In *Proc. 16th AAAI*, pages 526–533, 1999.
16. N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, 1995.
17. P. Mesegure and J. Larrosa Solving Fuzzy Constraint Satisfaction Problems. *Proc. IEEE-FUZZ'97*, pages 1233–1238, 1997.
18. I. Miguel. *Dynamic Flexible Constraint Satisfaction and its Application to AI Planning.* PhD thesis, Edinburgh University, 2001.
19. I. Miguel. Symmetry-breaking in planning: Schematic constraints. In *Proc. CP'01 Workshop on Symmetry in Constraints*, pages 17–24, 2001.
20. I. Miguel and Q. Shen. Dynamic flexible constraint satisfaction. *Applied Intelligence*, 13(3):231–245, 2000.
21. I. Miguel, Q. Shen, and P. Jarvis. Efficient flexible planning via dynamic flexible constraint satisfaction. *Engineering Applications of Artificial Intelligence*, 14(3):301–327, 2001.
22. W. Pedrycz and F. Gomide. *An Introduction to Fuzzy Sets: Analysis and Design.* MIT Press, 1999.
23. T. Schiex. Arc consistency for soft constraints. In *Proc. 6th CP*, pages 411–424, 2000.
24. T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proc. 14th IJCAI*, pages 631–637, 1995.
25. D. Smith and D. Weld. Temporal graphplan with mutual exclusion reasoning. In *Proc. 16th IJCAI*, pages 326–333, 1999.
26. T. Vossen, M. Ball, A. Lotem, and D. Nau. Applying integer programming to AI planning. *Knowledge Engineering Review*, 16:85–100, 2001.
27. M. Williamson and S. Hanks. Optimal planning with a goal-directed utility model. In *Proc. 2nd AIPS*, pages 176–181, 1994.
28. S. Wolfman and D. Weld. The lpsat engine and its application to resource planning. In *Proc. 16th IJCAI*, pages 310–317, 1999.