ICOT Technical Memorandom: TM-0982

TM-0982

GRAPEの計画問題支援機能
－並列制約論理型言語に基づくアプローチ－

上田　晴康、國藤　進、岩内　雅直、
大津　建太（富士通）

December, 1990

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

GRAPEの計画問題支援機能
— 並列制約論理型言語に基づくアプローチ —

上田晴康 國藤進 (富士通(株)国際情報社会科学研究所)
岩内雅直 大津延太 ((株)富士通ソーシアルサイエンスラボラトリ)

1 はじめに

GRAPEは複数の参加者から知識獲得をし、分類選択型問題から計画型問題までの幅広い知識システムのラピッドプロトタイピングを行なうグループウェアである。

すでに発表したGRAPEの初期知識ベース獲得機能[1, 2]では、分類選択型問題に必要な知識を獲得する。獲得される知識は、選択候補の名前、選択候補の類似性に基づく構造、候補を評価する評価項目、および評価項目ごとの各候補の評価値である。さらに、得られた評価に適当な重み付けをして総合的な評価値も得ることができる。例えば、旅行の行き先に関する総合的な主観的評価を行なうことができる。

計画問題支援機能は、初期知識ベース獲得機能で得られた候補を適当に組み合わせたプランの作成を支援する。例えば、評価の高い行き先を組み合わせて旅行計画を作る。プランを作成する際に初期知識ベース獲得機能で獲得された評価項目と、最終的に獲得された総合的な評価値を用いる。

2 GRAPEの計画問題支援機能

計画問題支援を行なうために、GRAPEでは計画問題向き知識獲得支援モジュール、計画作成支援モジュール、計画修正モジュールの3つのモジュールを用いる。

- 計画問題向き知識獲得支援モジュールは、過去のプランの事例、さまざまな考え得る制約などの入力を促し、また初期知識ベース獲得機能で得られた知識の変形を行なって計画作成に十分な知識を用意する。
- 計画作成支援モジュールは、前モジュールで獲得された知識を用いて、プランを動的に作成する。
- 計画修正モジュールでは、できたプランをユーザに評価してもらい、必要ならばさらに制約や過去のプランの事例を入力をしてもらうことにより修正を行なう。

十分なプランが作成されるまで、これらのモジュールが繰り返し実行される。

本稿では、これらのモジュールのうちで、計画作成の中心的な役割を果たす計画作成支援モジュールと、そこで用いる制約処理系について述べることにする。

2.1 計画作成支援モジュールの概要

計画問題では、さまざまな制約を満足するように、プランの合成を行なう必要がある。特に知識獲得をしながらプラン作成をする場合、単純な記述で十分に良いプランを作る必要がある。

そこでGRAPEでは制約処理系を用い、獲得された制約を用いて順次プランを作成していく。

2.2 計画問題記述の例

GRAPEでは、プランはリスト1のように記述される。この記述言語では、あるプランが適当な制約を満たしているかどうかを調べることと、そのプランがどのくらい良いものであるかを計算することを、KL1[3]風に記述する。

```
importance(I) :- true |
        I = [[time_seq, strong],
             [cost, -40],
             [start_at_late, 1]].

eval_plan(time_seq, Plan, _, Value ) :- true |
        test_time_seq(Plan, Value).
eval_plan(cost, Plan, Data, Value) :- true |
        cost_of_plan(Plan, Data, Value)
eval_plan(start_at_late, Plan, _, Value) :-
    Plan=[{Start,_,_}|_] |
        time_to_minutes("10:00", Compare_time),
        start_late(Start, Compare_time, Value).

test_time_seq([], V) :-  true |
        V = 0.
        % Instantiation of V means good Plan.
test_time_seq([First, Second | Rest], V) :-
    First={Fstart,_,Fend},Second={Sstart,_,_},
    Fstart <= Fend, Fend <= Sstart |
        test_time_seq([Second | Rest] , V).
% Fail means bad Plan.

cost_of_plan([], _, V) :- true |
        V = 0.
cost_of_plan([Place, Trans|Rest], Data, Val) :-
    true |
        fee_at_place(Place, Data, PCost),
        trans_fare(Trans, Data, TCost),
        cost_of_plan(Rest, Data, RCost),
        Val = PCost + TCost + RCost.

start_late(Start, Compare, Value) :-
    Start < Compare |
        Value = 0.
start_late(Start, Compare, Value) :-
    Start >= Compare |
        Value = 1.
```

リスト1: プランに関する制約の例

importance 述語は、どんな評価項目があるか、それをどのくらい重要視するかを記述するものである。

importance 述語で与えられる重要度は以下のような意味を持つ。

• 重要度が strong の評価項目は、eval_plan で得られる評価値は無視して良いことを示している。この評価項目には後述の強い制約が記述される。

• 重要度が数値の評価項目では、重要度が eval_plan で得られる評価値の係数となる。すなわち各係数を評価値に掛けて総和をとったものがプランの評価になる。この評価値が大きいものほど良いプランであり、相対的に早く作成されるようになる。この評価項目には後述の弱い制約と強い制約の両方が記述される。

この例では、cost という評価項目が、start_at_late という評価項目に比べて 40 倍重視されることを示している。

eval_plan 述語は、評価項目名、評価すべきプラン、大域的データ (時刻表や料金表) から、評価値を求めるものである。各評価項目の評価の方法として任意の KL1 [3] コードが書ける。eval_plan 述語は普通の KL1 と異なり、失敗しても良い。eval_plan 述語の失敗は、評価対象のプランが評価できないほど悪いことを示す。重要度が strong の評価項目は、この失敗するかどうかだけを評価の基準として用いる。

リスト 1 の記述から、以下のような制約が得られる。

• `Plan = [Place1, Trans1, Place2, Trans2, ...]` (time_seq, cost から)

• `Place = {Time1,..,Time2}` (cost から)

• `Trans = {Time3,..,Time4}` (cost から)

• Plan は行き先と交通手段を時間順に並べたリストで、それぞれに関して時間の重複はない。(time_seq から)

• cost の値は、Place の料金と Trans の料金の総和である。(cost から)

• start_time は、出発時刻の評価で、10 時以降だと評価が良い。(start_time から)

3 制約処理系

GRAPE においては、プランのテスト用プログラムを KL1 で書くと、それがそのままプラン作成用のプログラムを記述したことになる。

本節では、なぜテスト用のプログラムを書いたことが制約を表現したことになるのか、このテスト用プログラムをどう処理してプランを作るのか、またその中で制約はどう扱われるのかの三点について述べる。

3.1 制約処理系と Generate & Test

一般の制約処理系で与えられる制約は、プランの満たすべき必要条件である。そして制約処理系に要求されるのは、効率良くこの必要条件を満たしたプランを作り出すことである。

例えば、必要条件を満たしたプランを作るもののうち、最も単純だが効率の悪いものに generate & test がある。generator は手あたりしだいにプランを作り、tester はそれを条件に照らして、条件を満たしたものだけを正しいプランとして残す。

これに対し、この最も効率の良いものが制約処理系であると考える。制約処理系では、test として与えられた条件を解析し、それを満足するプランだけを作成しようとする。すなわち generate & test から test & generate へと発想の転換を行なったのである。

GRAPE の制約処理系はこの考え方に基づき、test 用の KL1 プログラムを解釈して generate を行なう。

3.2 GRAPE の制約処理

GRAPE で用いる制約処理系は、以下の特徴を持つ。

• 処理する制約を二つに分け、最低限満たさなくてはならない種類の制約 (強い制約と呼ぶ) と、強い制約を満たした上でさらに満たした方が良い数値的な制約 (弱い制約) とする。eval_plan 述語が失敗しないことが強い制約であり、eval_plan 述語で計算される評価値を大きくすることが弱い制約である。

• 強い制約にしたがってプランを作るために、制約伝播を用いる。この時使える制約は、言語作成上の都合から、線形 1 次方程式、不等式、複数候補からの一要素の選択の 3 種に制限する。

• プランの generator は、弱い制約にしたがってプランの評価値の大きいものからプランをつくり出す。これを行なうために、これまでの制約処理系と異なり、制約伝播では解けない部分の解を求めるのに generate & test ではなく、制約変数の分割による divide & conquer を用いる。

これらの特徴を満たすために、GRAPE の制約処理系の実行は以下のアルゴリズムにしたがって行なわれる。

Step 1　最も抽象的な、いいかえれば何も決まっていないプランから始める。

Step 2　一つのプランの一部を instantiate して、or 並列に計算できる複数のプランを作るため、どこを instantiate するかを決める。

Step 3　複数のプランを計算する or 並列なプロセスを作る。この時分かれた複数のプロセスは、互い

に通信しないで計算を行なう。

Step 3.1　それぞれのプランの優先度を計算する。

Step 3.2　制約変数とその依存関係をコピー(プロセスの fork)し優先度を割り当てる。

Step 3.3　必要ならリダクションの処理をする。

Step 3.3.1　新たに必要な制約変数の作成と依存関係の設定を行なう。

Step 3.3.2　リダクションされた節のボディ部を次の or 並列プラン作成の候補に加える。

Step 4　sup-inf 法を用いて制約を伝播する。
伝播中に見つけられた分割可能な変数は or 並列プラン作成の候補に加える。

Step 5　伝播の終了を待つ。

Step 6　全ての変数が instantiate されていれば
得られたプランとその評価値を出力し、
そうでなければ Step 2 に戻って繰り返す。

以下では、これらの特徴を持った制約処理言語が、どのように制約を処理するかを述べる。

3.3　or 並列による複数プランの作成

本来制約処理言語では、変数の取り得る複数の値は一度に考慮され、制約を加えてその候補を徐々に減らして行く。そのため、全てのプランの候補を別々に考慮しないですむ。

一方変数の値の取り方によって、プランの評価値はまちまちになる。GRAPE では評価値の大きいプランから作成するように制御を行なうため、評価値の違うプランは別々に考慮する必要がある。

GRAPE は、一つのプランの一部を instantiate して評価値が大きく異なる複数のプランを作る時、すなわち、(1) リダクションで複数の節の選択が起きる時と、(2)divide & conquer によって一部の変数の値の異なる複数のプランを考える時に、プランを or 並列なプロセスに分けて計算する。

しかし、このように or 並列に分けるべき点は多数存在し、そのうちのどれを選択すれば最も効率良くプランを作ることができるかを判断するのは難しい問題である。そこで、ヒューリスティクスを用いて判断を行なう。

一般的な基準として以下の三点が挙げられる。これらの基準は前から順に重視される度合が大きい。

1. プランの評価値が大きくなるように、プランの変数の値を決められること。

2. 探索範囲を狭められるように、他の多くの変数から依存されている変数の値を決められること。

3. プランの自由度が下がらないこと。

これらのヒューリスティクスを計算するために、テスト用プログラムの中の評価値に関する制約が解析され、複数の or 並列なプランにした時に評価値が元のプランのものよりどのくらい大きくなるかが調べられる。

3.4　or 並列プランの制御

弱い制約は、相対的な制約である。このため、各プランの評価値を比較せずに探索の枝刈りをすることはできない。例えば α-β 枝刈りでは、すでに計算したプランの評価値を元にして枝刈りを行なう。

しかし並列処理言語系の上でそれぞれの or 並列プランの比較を行なうと、通信量がかさんだり待ち合わせが起きて、十分な並列性が生まれない。

そこで、並列処理言語の優先度制御機能を用いて、評価値の低いプランはゆっくりと計算し、評価値の相対的に高いプランから計算をする。そのために or 並列に動くプランの優先度は以下の二つの基準にしたがって決められる。

- 評価値の高いプランほど優先度が高い。
- 決まった部分に比べ変数部分の多いプランほど優先度が高い。

この二つの基準は、変数部分を含んだプランでは正確に評価が行なえないため、変数部分の不確定要素の分だけ優先度をあげて計算を行なうことに相当する。

3.5　制約伝播

前節で述べたように or 並列なプランを作った後は、制約伝播を行なうことで、プランの中の各変数の取り得る値を減らして行くことができる。これにより、効率良くプランを作成することができる。

GRAPE で用いる制約伝播は、比較的単純な方法の sup-inf 法[4] を線形一次不等式の他に、線形一次等式、複数候補からの一要素の選択も制約として扱えるよう拡張したものである。

sup-inf 法は、制約に沿って変数同士が情報交換することで制約伝播を行ない、変数の取り得る値の範囲を減らして行く。sup-inf 法では制約は線形一次不等式のみで表現されるので、制約変数の取り得る値は一つながりの区間となる。この区間は一つだけの制約変数に関連する制約であり、その区間の上限と下限の二つの数値 ($\pm\infty$ を許す) を変数同士で情報交換することで、制約伝播を行なう。

GRAPE では、制約の種類を拡張して、線形一次方程式、複数候補からの一要素の選択の制約も使えるようにする。この様な拡張をしても一つの制約変数についての制約は、$[1 \leq X \leq 2, X \in \{4,5,6\}]$ のようにコンパクトに表

現することができる。制約関係のある制約変数同士でこれらの表現をやりとりすることにより、冗長な制約を取り除くことができる。

3.6 変数値の分割による探索

3.3節でも述べた通り、変数の値の取り方によってはプランの評価値はまちまちになってしまう。しかし、全てのプランを別々に考慮したのでは、強い制約による探索範囲の枝刈りができず効率が悪い。

そこでGRAPEでは適当なタイミングで変数の値を分割し、それぞれの値を持つプランをor並列に計算する。

変数をどう分割すれば最も効率的にプランの探索範囲を狭めることができるかは、以下の基準にしたがって判断する。

- その変数の値の候補が数個しかないときは、その変数にそれぞれの値を入れた数個のプランを作る。
- 多くの制約によって、制約された変数を適当な区間に分割して、それぞれの区間を持つプランを作る。

これらの分割は、リダクションと同様にor並列分割の候補として考慮され、3.3 節で述べたヒューリスティクスで選び出された後、or並列のプランとして分割される。

4 他の制約論理型言語処理系との比較

他の有名な制約論理型言語[5] として、 Prolog III[6], CLP(R)[7], CAL[8] が挙げられる。これらの処理系はいずれも強い制約のみを扱っており、制約を満たした解は全て同程度に望ましいものとされる。

しかし計画問題のように、強い制約を満たした解の間に相対的な評価ができ、それらのうち最も評価値の高い解を求めたい場合は、強い制約の範囲内でさらに generate & test をしなくてはならない。

また、強い制約の間に一定の優先順位を与え、優先順位の低い制約を無視することによって同様の効果をもたらそうとする研究もある[9, 10]。

しかし、コストと質などのトレードオフは、優先順位をつけた制約でも解消することができない。なぜならば、どちらかの制約があらかじめ決められた優先順位にしたがって先に満たされてしまうためである。

GRAPEでは、弱い制約と強い制約を交互に適用する。また弱い制約の解消に当たっては、より効率の良い divide & conquer 戦略をとる。これにより、計画型問題で必要となる相対的に評価の高い解をより効率的に作成することができる。

5 終りに

GRAPEの計画問題支援機能について述べた。この機能のために専用の制約指向並列論理型言語を設計した。

今後の課題としては、この言語のKL1 上での実現および探索の効率化の二点が挙げられる。

謝辞 本研究の一部は第五世代コンピュータプロジェクトの一環として行なわれました。

また、制約指向言語については当研究所の平石邦彦研究員に貴重な助言をいただきました。

参考文献

[1] 國藤進他. グループ知識獲得支援システムGRAPEにおける初期知識ベース獲得機能. 人工知能学会研究会資料 SIG-HISG-8903-5, アルカディア市ヶ谷私学会館, December 1989.

[2] 上田晴康, 國藤進, 須永知之, 岩内雅直. 知識獲得支援のためのグループウェア GRAPE とその実現について. 人工知能学会全国大会 (第4回) 論文集, 学習院大学, July 1990.

[3] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the parallel inference machine operation system(PIMOS). In *Proceedings of FGCS '88*, volume 1, pp. 230–251, 1988.

[4] 川岸太郎, 坂根清和, 生駒憲治. 線形不等式を解く制約ソルバーの並列計算について. 情報処理学会研究報告, Vol. ソフトウエア基礎論, No. 89-SF-33, pp. 1–7, 1989.

[5] Jacques Cohen. Constraint logic programming languages. *CACM*, Vol. 33, No. 7, pp. 52–68, July 1990.

[6] A. Colmerauer. Prolog III 入門. 制約論理プログラミング, pp. 51–74. 共立出版, 1989.

[7] J. Jaffar and J-L Lassez. 単一化から制約へ. 制約論理プログラミング, pp. 31–50. 共立出版, 1989.

[8] A Aiba and et al. Constraint logic programming language cal. *International Conference on Fifth Generation Computer Systems*, pp. 263–276, 1988.

[9] 佐藤健. 解釈の順序による柔らかい制約の定式化. TR 502, ICOT, 1989.

[10] A. Borning, M. Maher, A Martindale, and M. Wilson. Constraint hierarchies and logic programming. Technical Report 88–11–10, Computer Science Department, Univ. of Waxhington, November 1988.