ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO – USP

Departamento de Engenharia Mecatrônica

Fernando Augusto Anastácio — N°usp: 3103786
Rodrigo Chanyon Chen — N°usp: 3129833
Tiago Stegun Vaquero — N°usp: 3632988

nota final 9,5 (nove e meio)

Algoritmo de Planning: Graphplan Modificado

Trabalho de Conclusão de Curso apresentado à Escola Politécnica da Universidade de São Paulo.

Área de concentração:
Engenharia Mecatrônica

Orientador:
Prof. Dr. José Reinaldo Silva

São Paulo, 2003

Dedicamos esta obra ao Prof. Dr. José Reinaldo Silva, que tem sido a grande razão e incentivo de nosso aprendizado e fundamental para a realização e conclusão deste trabalho.

Agradecimentos

Nossos pais e familiares, por quem devemos gratidão eterna, por todos os momentos de felicidades e dificuldades em nossas vidas.

Agradecemos os professores da Escola Politécnica da USP, por todo os momentos de saber e de ensino que nos fazem hoje realizar este trabalho e nos formar como Engenheiros.

Nossos amigos e companheiros, que sempre nos apoiaram em momentos imprevisíveis e de grandes desafios com lealdade e companheirismo.

Agradecemos também, nossos colegas do Laboratório de Design que nos ajudaram em ferramentas e conselhos.

Manifestamos também nossos agradecimentos às pessoas que indiretamente estiveram presentes com ajuda e carinho durante todas as árduas dificuldades.

SUMÁRIO

1 OBJETIVO DO TRABALHO 4

1.1 OBJETIVO GERAL 4

1.2 OBJETIVO ESPECÍFICO 5

2 INTRODUÇÃO E MOTIVAÇÃO 7

3 CONCEITOS E DEFINIÇÕES 10

3.1 INTELIGÊNCIA ARTIFICIAL 10

3.2 PLANNING 11

3.2.2 STRIPS 12

3.2.3 Estratégias para inferir um plano 14

3.2.4 GraphPlan 17

3.2.5 Rede de Petri (RdP) 25

3.2.6 Rede de Petri e Inteligência Artificial 31

3.2.7 Resultados da Rede de Petri no AIPS 33

4 ANÁLISE DO GRAPHPLAN 35

4.1 ESTRUTURA DE ARMAZENAMENTO DE DADOS DE ENTRADA 37

4.1.1 Operadores (ops) 37

4.1.2 Fatos: declaração dos parâmetros, fatos iniciais e objetivos (facts) 38

5 GRAPHPLAN MODIFICADO 40

5.1 MODIFICAÇÕES NA INTERPRETAÇÃO DOS DADOS DE ENTRADA 40

5.1.1 A entrada de Fatos (facts) 40

5.1.2 A entrada de Operadores (ops) 45

5.2 MODIFICAÇÕES NO PLANEJAMENTO 47

6 ANÁLISE DOS RESULTADOS OBTIDOS 49

6.1 DESCRIÇÃO DO PROBLEMA DE TESTE 49

7 CONCLUSÕES 62

8 REFERÊNCIAS BIBLIOGRÁFICAS 63

9 ANEXOS 65

1 OBJETIVO DO TRABALHO

1.1 OBJETIVO GERAL

A primeira premissa de um projeto a ser definida é o objetivo, ou seja, o foco do trabalho. Todos os esforços e conclusões devem tender a um objetivo único. Para se atingir este objetivo, várias etapas devem ser concluídas. Assim, em um projeto, sub-objetivos são propostos.

O objetivo geral e único deste trabalho é o desenvolvimento de um novo algoritmo de planejamento, aplicável às exigências do mundo real, envolvendo teorias de Inteligência Artificial, teoria de rede de Petri e bases não lógicas de informação e outras. Um exemplo de aplicação no qual nos focaremos no projeto é o campo de elevadores inteligentes, onde a necessidades de atender os passageiros em um tempo ótimo é necessário.

O objetivo geral visa a aplicação da Teoria de Rede de Petri para o desenvolvimento de um *planner* (algoritmo Planejador) nomeado de **PlaNets** capaz de: dada um domínio, uma descrição inicial deste, o objetivo a ser atingido, e as ações que podem ser tomadas sobre ele, o algoritmo é capaz de fornecer um plano constituído de uma seqüência de ações a serem tomadas para que possa, à partir da condição inicial, chegar ao objetivo. O importante é que esse plano seja um plano ótimo que leva em consideração as variáveis de importância de cada domínio (ex.: tempo, capacidade, distâncias) e a rede de Petri possui diversas vantagens que nos levam a sua utilização.

Mas para atingirmos este objetivo é necessário primeiramente provar a necessidade da utilização dessas teorias juntamente com a já existente (Inteligência Artificial). E assim surge a necessidade de uma etapa a ser concluída antes de atingirmos

o objetivo geral. Essa etapa é o que chamamos de objetivo específico que nos ajudará fortemente no desenvolvimento do novo algoritmo.

1.2 Objetivo Específico

No presente trabalho o objetivo específico é a modificação do algoritmo de *Planning* Graphplan criado por Avrim Blum e Merrick Furst em 1995 que teve um enorme destaque na comunidade de Inteligência Artificial e hoje tem diversos descendentes. Faremos com que o Graphplan, que só trabalha com lógica de primeira ordem, aceite e trabalhe com base não lógica de informação (contadores, pilhas), com isso reduzir o volume de informações de entrada e capacitá-lo a lidar com o conceito de capacidade. Estas modificações visam à otimização do algoritmo e a observação da exigência de bases teóricas como a da rede de Petri para aplicações no mundo real, como por exemplo, elevadores inteligentes.

Para efetuarmos a modificação é necessário o completo entendimento do algoritmo Graphplan. Com o entendimento será possível conhecer suas falhas e limitações que trazem a tona às exigências de um novo algoritmo.

Com esse projeto gostaríamos de provar que:

- O Graphplan tem muitas limitações e não lida com diversos problemas do mundo real.

- Mesmo sendo modificado, o Graphplan necessita de novas teorias a serem agregadas a ele.

- A utilização de lógica clássica e base não lógica é realmente eficiente em algoritmos de planejamento.

- A teoria de rede de Petri tem muito a acrescentar para a otimização de planejadores;

- A necessidade do desenvolvimento futuro de um novo algoritmo é real onde poderemos levar em consideração todas as descobertas efetuadas neste projeto.

2 INTRODUÇÃO E MOTIVAÇÃO

Nos últimos dez anos a área de algoritmos de planejamento tem tido um enorme avanço. Um dos propulsores desse avanço é a grande conferência internacional de Inteligência Artificial conhecida como AIPS *(Artificial Intelligence Planning and Scheduling)*, onde muitos pesquisadores se encontram para apresentarem os novos resultados no crescente campo da Inteligência Artificial, refletindo o esforço hoje despendido pela academia nesta área. Naturalmente este esforço se faz presente em novos artefatos que usam AI Planning, tais como os elevadores inteligentes, robôs móveis, automóveis com planejadores embarcados, etc.

Devido a essa grande evolução, muitas empresas e pesquisadores tem se interessado sobre o assunto e estes têm trazido novas idéias e novas aplicações para os algoritmos que surgem e são revelados no AIPS. Algumas empresas buscam soluções de seus problemas de controle (exemplo: controle em Tempo Real) nos conceitos de IA aplicando estes algoritmos de planejamento (*planners*). Empresas como a NASA, IBM, Schindler Lifts Ltd e outras. A maioria, se não a totalidade, desses algoritmos são provenientes de estudos feitos em grandes universidades no mundo. Porém, uma dificuldade encontrada pelos participantes do AIPS é criar *planners* que não sejam específicos demais para cada aplicação no mundo real. Como é possível perceber, muitas idéias são boas, mas sua aplicação no mundo real se torna difícil em função da natureza dos conceitos utilizados pelos pesquisadores para criarem o *planner,* como é o caso do Graphplan. Representações formais como as da lógica clássica e da lógica linear dificultam a representação de domínios que possuem características como: uso de recursos (máquinas, ferramentas, etc.), capacitação (domínios que possuem alguma limitação), conhecimentos que variam no tempo, etc. Temos, portanto, uma dicotomia entre os planejadores globais baseados em algoritmos de *problem solving* e os planejadores para fins específicos, que geralmente atingem um melhor desempenho.

Podemos, portanto, destacar dois pontos de discussão na literatura: i) o uso combinado de algoritmos genéricos e de informação específica sobre o domínio em apreço; ii) a combinação de representações baseadas em lógica e esquemas que melhor representam dinâmica, como as redes de Petri.

Neste trabalho, com continuação em duas dissertações de mestrado, nossa motivação é a criação de um novo algoritmo de planejamento (*planner*) chamado ***PlaNets*** que une conceitos básicos da AI e teorias de Rede de Petri. A formulação desse plano baseia-se na combinação destas duas abordagens para superar problemas encontrados por outros algoritmos nas aplicações práticas no mundo real. A diferença entre a abordagem proposta neste trabalho e outras que constam da literatura (Drumond, Petrix, Hendler, etc, ver Readings on Planning [6]) é que estamos propondo uma representação híbrida onde a parte causal é colocada em lógica e a parte seqüencial em Redes de Petri. Portanto não estamos propondo simular o plano pura e simplesmente. Segundo, o uso das redes de Petri colocado neste trabalho se refere ao modelo dinâmico do domínio e não ao plano. Assim, estamos propondo a análise de domínio ao elaborar o plano e não simplesmente um processo direto de *busca* pelo melhor plano. A verificação do plano poderia ser feita mais pela análise de propriedades da rede do que pelo jogador de marcas.

Por exemplo, uma das grandes dificuldades enfrentadas hoje pelos *planners* (por exemplo, o Graphplan) que se apresentaram nos AIPS de 1998, 2000 e 2002 é o tratamento do problema "capacitado" como é o caso dos elevadores. O algoritmo deve inferir um plano capaz de satisfazer todos os pedidos dos passageiros e deve levar em conta a capacidade do elevador. Alguns *planners* abordam este problema sem levar em conta a capacidade, o que implica em destacar o elevador para atender um andar com N pessoas enquanto o elevador tem um número de lugares menor que N. Problemas deste tipo também acontecem no escalonamento de helicópteros para aplicações reais, como a da Bacia de Campos da Petrobrás [16][17].

Nosso algoritmo tem muito a acrescentar propondo uma solução para este problema e outros, com o auxílio de uma base teórica de muito potencial como é o caso da Rede de Petri. Após as modificações no Graphplan e as informações adquiridas através das modificações neste projeto, desenvolveremos o algoritmo visualizando aplicações práticas do mundo real que exijam características que são problemáticas para muitos algoritmos hoje existentes. O objetivo direto seria a participação no AIPS em 2006.

3 CONCEITOS E DEFINIÇÕES

3.1 INTELIGÊNCIA ARTIFICIAL

Inteligência artificial é um dos mais importantes assuntos da ciência de computação moderna. Apesar de inventado em 1955, no MIT (*Massachusetts Institute of Technology*), por John McCarthy, então professor de Matemática, e nascido oficialmente na conferência de Dartmouth, encontra-se apenas no início da sua carreira.

Mas o que significa Inteligência artificial? Não tem uma definição precisa, mas trata-se, naturalmente, de "algo" que tem a ver com a construção de máquinas dotadas de uma inteligência semelhante à humana. Assim, a tecnologia da Inteligência Artificial tem cinco campos básicos:

- Método da representação do conhecimento;
- Método de raciocínio;
- Método do tratamento do conhecimento incerto;
- Método da planificação;
- Método da aprendizagem.

Muitos problemas envolvendo os temas acima podem ser encontrados no mundo que vivemos e por esses motivos podemos encontrar o crescimento da AI em diversos campos da economia, sejam elas manufaturas (robôs) até softwares especiais. Um jogo de xadrez, por exemplo, envolve planejamento que pode ser realizado pelo método da planificação, como citaremos nos próximos itens.

3.2 PLANNING

Por definição, a finalidade de um *planner* consiste em reduzir a própria ação (task) a uma seqüência ótima de ações primitivas, baseando-se no conhecimento das várias ações possíveis e nos seus efeitos. Assim, não existe conhecimento específico do domínio, mas conhecimento genérico das ações que se podem realizar em tal domínio e dos seus efeitos. O objetivo é construir um algoritmo, juntando um certo número dessas ações.

Na abordagem clássica, os principais elementos de uma atividade de *planning* são o operador e o estado. Um operador (a ação) provoca a passagem de um estado para outro (por exemplo, a posição de um robô). Um problema pode sempre ser representado por um conjunto de operadores e de estados. Um operador (ou uma ação) é definido por: um nome, um conjunto de pré-condições (condições sob as quais é capaz de operar) e um conjunto de efeitos ou pós-condições. O *planning* mais simples consiste em aplicar todos os operadores disponíveis: primeiro ao estado inicial, depois a cada um dos estados obtidos e assim sucessivamente. Portanto a abordagem clássica consiste na busca por uma seqüência de ações ótima para levar do estado inicial ao estado final (objetivo).

O objetivo do *planning* é encontrar uma seqüência de operadores que conduza ao *goal* (objetivo) com uma seqüência muito reduzida, isto é, a mais curta possível. O primeiro programa planejador foi o GPS (*General Problem Solver*, realizado por Newell nos anos 60), um solucionador geral de problemas que representavam o mundo em termos de estados e operadores. O GPS constitui-se de dois procedimentos que colaboram na percussão da solução. O primeiro tem como finalidade de busca os estado final a partir do estado inicial. Para isso, o procedimento examina a diferença entre os dois estados e escolhe um operador que opere precisamente sobre tal diferença; depois

informa ao outro procedimento para aplicar o operador à fonte: esta cria um novo estado resultante que se torna à nova fonte.

O segundo procedimento tem como tarefa aplicar um operador a um estado. Para isso este chama o primeiro procedimento para que ele transforme o estado de entrada em um outro ao qual se possa aplicar o operador e, uma vez realizada a operação, produz-se o estado resultante. Portanto o modelo proposto pelo GPS também pode ser considerado um algoritmo de busca.

3.2.2 STRIPS

Na realidade, a capacidade de planificação do *General Problem Solver* era muito limitada. Um avanço decisivo foi o realizado pelo STRIPS, desenvolvido em 1969, no *SRI International*, por Richard Fikes e Nils Nilsson. Este programa adota a mesma técnica do *General Problem Solver*, ou seja, escolhe o operador a aplicar baseando-se na diferença entre o estado do mundo atual e o objetivo a atingir (chamado *means-end analysis*).

Assim, STRIPS é um algoritmo para resolver problemas que operam com um modelo de mundo representado por um conjunto de fórmulas em lógica de primeira ordem. Um sistema de STRIPS descreve o efeito de uma ação através de regras que definem como o modelo de mundo atual deveria ser modificado.

Um sistema de STRIPS é definido por um modelo de mundo inicial e por um conjunto de operadores, que correspondem às ações de mudança do estado atual. Assim, o STRIPS tenta achar uma seqüência de operadores para transformar o modelo de mundo inicial em um outro modelo que satisfaz uma dada fórmula em lógica de primeira ordem que representam o estado final proposto.

A descrição de cada operador consiste em definir sua *pré-condição* (*precondition*) (a condição de aplicabilidade, expressada por uma fórmula de primeira ordem), sua *add list* (a lista de fórmulas que precisam ser adicionadas ao modelo de mundo atual) e sua *delete list* (a lista de fórmulas que não serão mais verdade após a aplicação do operador e então precisam ser deletadas) [9].

São duas as novidades importantes do STRIPS:

1. STRIPS leva em consideração uma situação completa, não somente o estado isoladamente, mas também o contexto no qual o estado se encontra se encontra.

2. STRIPS utiliza a dedução matemática para decidir os seus próprios passos.

Apesar das várias qualidades do STRIPS, ele não era capaz de resolver muitos problemas por algumas falhas na forma com que planeja. Algumas vezes, na obtenção de um plano, alguns subproblemas podem ser resolvidos independentemente, não importando a ordem de execução. Nestes casos, a abordagem linear (atingir cada sub-objetivo independentemente) funciona muito bem. Algumas vezes os passos de resolução do problema vão interagir uns com os outros e nestes casos particulares nós devemos saber se eles exigem algum tipo de plano não linear, como por exemplo, um plano ordenado que não trará cada passo como sendo independente um do outro.

Para simbolizar bem esse conceito tomemos como exemplo um pintor que deseja pintar o forro de uma sala. É necessário que o pintor esteja ao mesmo tempo em cima da escada e com a lata de tinta em suas mão para pintá-lo. Imaginemos que a lata de tinta esteja no chão e que a escada já esteja posicionada corretamente. O pintor pode então pegar a lata e subir a escada, o que funciona muito bem para alcançar o objetivo, ou ele pode subir a escada e pagar a lata, o que não funciona, pois uma vez em cima da escada, não se tem acesso à lata de tinta. Esta ultima estratégia simboliza a falha da teoria linear onde se tenta alcançar os objetivos independentemente, ou seja, sem nenhuma ordem.

Assim a ordem de cada sub-objetivo ('pegar a lata de tinta' e 'subir a escada') é muito importante nesse caso e em muitos outros em nosso mundo.

Assim possível perceber que a ordem é muito importante. Para esse caso específico a resolução é muito simples, mas seria impossível analisarmos caso a caso onde isso ocorre na resolução de cada problema. No STRIPS isso também ocorre e não é capaz de resolver muitos deles. Sussman apresenta diversos exemplos onde o STRIPS não é capaz de resolver problemas de não linearidade [4].

3.2.3 Estratégias para inferir um plano

Uma das características importantes que determinam a eficiência do algoritmo de planejamento, *planner,* é como ele lida com as informações que são fornecidas. Para inferir um plano o planejador deve, com as informações que lhe foram dadas sobre o domínio, aplicar os operadores, ou seja, aplicar as ações possíveis sobre aquele domínio e tentar achar um caminho que leve da condição inicial ao objetivo ou vice-versa. Podemos determinar o caminho, plano, de duas maneiras: a primeira consiste em partir das condições iniciais e tentar atingir o objetivo através da aplicação de operadores, essa é chamada de *Forward*; e a segunda maneira, chamada de *Backward,* traça o plano, mas aplicando os operadores partindo do objetivo e tentando chegar às condições finais.

Embora não haja diferença formal entre um sistema que trabalhe em um problema na direção *Forward* e um que trabalhe na direção *Backward*, é conveniente fazermos esta distinção explícita. Quando um problema tem intuitivamente claras as condições iniciais (*states*) e objetivos (goals) e quando nós escolhemos partir das informações das condições inicias para atingir o objetivo dizemos que o processo é uma busca *Forward*. Nesta busca ações e regras são aplicadas à condição inicial produzindo novos estados (sub-estado). Se decidirmos partir das informações dadas como objetivo e tentar atingir as condições iniciais, esse processo é chamado de busca *Backward*. Na

busca *Backward* ações e regras são aplicadas ao objetivo produzindo sub-objetivos e a busca continua até ser encontrada a condição inicial.

No processo de busca para traçar o plano válido, é gerada a chamada "árvore de busca" ou espaço de busca contendo os vários estados gerados pela aplicação dos operadores (ações). Essa árvore só pára de gerar novos estados quando o objetivo é atingido, no caso da busca *Forward* podemos dizer que o processo para quando existe um caminho que ligue a condição inicial ao objetivo. Cabe ao algoritmo selecionar o melhor caminho nessa "árvore de busca" assim fornecendo o tão desejado plano. Uma observação importante a ser feita é que o algoritmo não deve criar o espaço de estado para inferir o plano e sim uma "árvore de busca". A diferença é que a geração do espaço de estados é explosiva, pois ele gera todos os estados possíveis nas aplicações dos operadores mesmo que estes estados sejam redundantes. Já na árvore de busca é papel fundamental do algoritmo eliminar estados redundantes (ex.: um objeto não pode estar em dois lugares ao mesmo tempo).

Em alguns problemas a serem resolvidos pelos algoritmos, o tamanho da "árvore de estados" gerada difere dependendo da direção de busca utilizada. Há casos em que é mais eficiente resolver o problema em uma direção se comparada com a outra. Suponha, por exemplo, que exista um número grande de objetivos a serem atingidos e uma condição inicial. Não seria muito eficiente se tentássemos resolver o problema na direção *Backward*, nós não saberíamos a priori qual objetivo está mais "perto" do estado inicial e nós teríamos que começar uma busca a partir de cada um dos objetivos. Isso aumentaria a "árvore de busca" o que faz com que o tempo gasto pelo algoritmo, para definir o plano, crescer, o que é ruim.

Uma ótima idéia é resolver problemas bidirecionalmente, ou seja, usar simultaneamente a busca *Forward e a Backward*. O processo de busca e geração do grafo se dá tanto da condição inicial para o objetivo quanto do objetivo para a condição inicial simultaneamente. O processo termina quando as fronteiras das duas buscas se

encontram. Estudos mostram que em muitos casos a busca Bidirecional é mais eficiente que a busca unidirecional. Observe a Figura 4 que simboliza a diferença entre a busca bidirecional e a unidirecional no caso de *breadth-first*. [9]

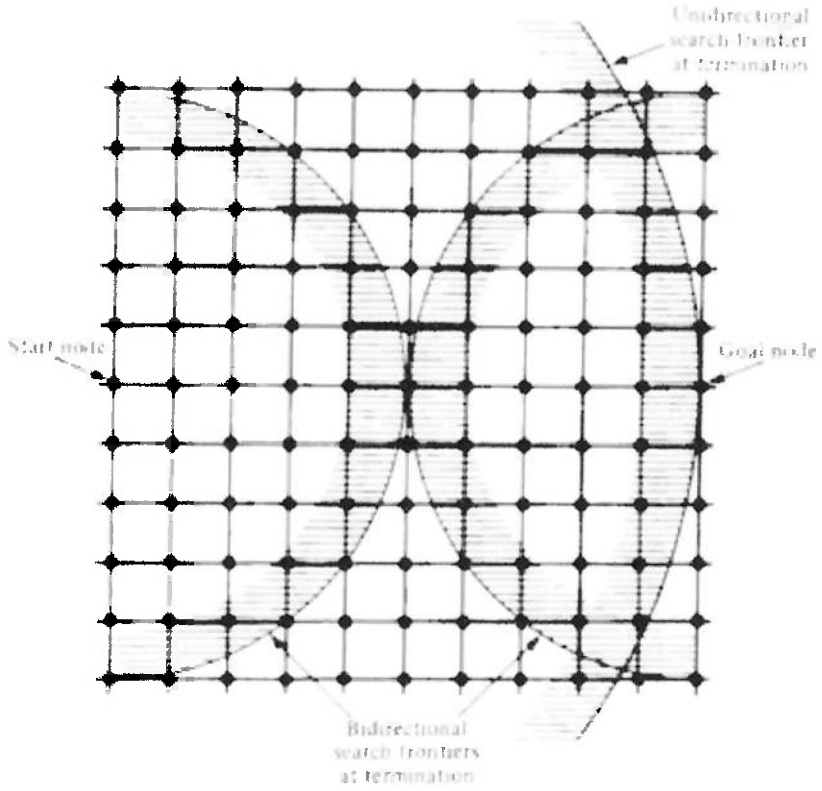


Figura 1.

Quando utilizamos o bidirecional ou o unidirecional ainda temos a opção de utilizar uma busca *depth-first* ou *breadth-first*. A busca *deph-first*, ou seja, busca em profundidade, analisa um ramo por vez da árvore e tenta ligar o estado inicial ao objetivo, se não for possível conectá-los passa-se a analisar outro ramo. Já a busca *breadth-first*, ou seja, busca em largura, tentar conectar os dois estados analisando todas as possibilidades relevantes ao redor do estado atual em analise. Devemos tomar cuidado com o uso do *deph-first*, pois esta pode causar problemas com o da Figura 5, o que torna o uso da busca bidirecional inútil, se tornando similar ao unidirecional.

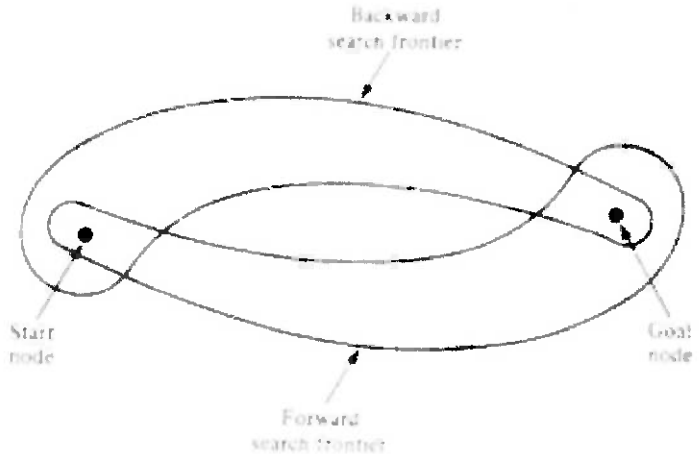


Figura 2.

Um aspecto muito interessante da busca bidirecional, por exemplo, com o uso do *breadth-first*, é que se continuarmos a busca mesmo após ter achado um caminho (plano) podemos achar outros caminhos válidos e até compará-los em diversos aspectos e ter mais de uma opção de plano.

3.2.4 GraphPlan

No que concerne ao uso de grafos em planning, um sistema que teve muito destaque na comunidade de Planning foi o GraphPlan, criado por Avrim e Merrick Furst em 1995 e que hoje possui muitos descendentes. O conhecido Graphplan é um algoritmo baseado nos princípios do STRIPS que nos permite construir e analisar a compacta estrutura de um Planejamento gráfico, chamando de *Planning Graph*. Esse algoritmo tem a característica de nos garantir o fornecimento de um plano válido, e se não houver um plano o algoritmo irá retornar "*no plan exist*".

É importante citar que o Planning Graph não é o espaço de estados do domínio, o que poderia ser muito grande, mas sim uma representação gráfica de um plano para se chegar ao objetivo a partir de uma dada condição inicial.

3.2.4.1 Definições

A análise do Planning Graph aplica os princípios do STRIPS. Neste domínio, os operadores têm precondições, efeitos adicionados, efeitos deletados, e esses são conjuntos de proposições. Tem-se também parâmetros que podem ser instanciados à objetos do mundo. Devemos lembrar que operadores não criam ou destroem objetos e que a representação do tempo é discreta.

Podemos entender como sendo um problema de planejamento como tendo:

- Um domínio com os princípios do STRIPS;
- Um conjunto de objetos
- Um conjunto de proposições (literais) que representam a Condição Inicial.
- E o Objetivo do Problema que também são proposições que são requeridas a serem verdades no final do plano.

No Planning Graph entendemos por nó como sendo as proposições e por ação como sendo o operador. Uma ação tomada no tempo *t* adiciona ao mundo todas as proposições que se encontra na sua lista de proposições dos efeitos adicionados e deleta todas as proposições que se encontraram nos efeitos deletados. Além desse dois efeitos temos os *frame actions* ou *on-op* que representam as coisas que não mudam após uma dada ação. Vale a pena citar que em alguns problemas passamos muito mais tempo tentando descrever coisas que não mudam do que coisas que realmente mudam, por isso, a representação do domínio coerente é muito importante.

Para entendermos o algoritmo é necessário visualizar algumas características do Graphplan. A propagação e redução dos nós se dão pelas relações *mutuas de exclusão*, que nos diz que duas ações são mutuas de exclusão se nenhum plano valido pode conter

as duas, ou seja, não se pode fazer ambas verdadeiras. A relação mutua de exclusão tem sua maior e principal contribuição na redução da busca no Planning Graph. A aplicação das relações de *exclusão mutua* se dá pelo uso de duas regras: Interferência e Necessidades Competitivas. Então entendemos como duas ações serem mutuas de exclusão (exclusivas) como tendo as seguintes propriedades:

Interferência: Se as ações deletam a precondição ou os efeitos adicionados um do outro. Um exemplo seria, no domínio de um foguete com 'Foguete 1 em Londres' e ' Foguete 1 com tanque cheio' como condições iniciais, as ações 'mova Foguete 1 de Londres até Paris' e 'ponha Alex no Foguete 1' no tempo 1 são exclusivas (mutuas de exclusão) porque a primeira deleta a proposição 'Foguete 1 está em Londres' que precondição da segunda.

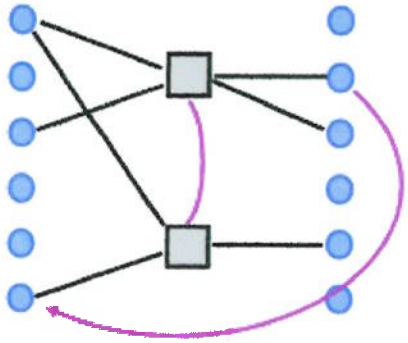


Figura 3.

Necessidades Competitivas: Se existe uma precondição de uma ação *a* e uma precondição de uma ação *b* que são marcadas por serem mutuamente exclusivas uma da outra. Um exemplo seria tomar como condição inicial como sendo 'Jason em Paris' e as ações côo sendo 'ponha Alex no Foguete 1 em Londres' e 'ponha Jason no Foguete 1 em Paris' no tempo 2, e estas são exclusivas por terem as necessidades competitivas como sendo as proposições 'Foguete 1 em Londres' e 'Foguete 1 em Paris'.

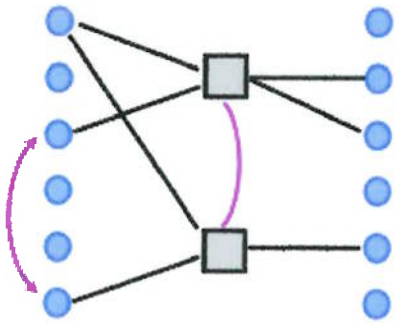


Figura 4.

3.2.4.2 Descrição do Algoritmo

Começando com um Planning Graph que possui apenas um simples nível de proposições contendo a Condição Inicial, o Graphplan ocorre em estágios. No estágio *i* o Graphplan utiliza o planejamento gráfico (Planning Graph) do estágio *i-1*, cria o nível de ações (ações aplicáveis, operadores) então utilizando o principio das relações mutuas de exclusão cria o plano válido do estágio *i*. A procura do Graphplan tanto pode achar um plano válido quanto pode determinar que o objetivo ou o sub-objetivo não pode ser alcançado no tempo *i*. Então, cada iteração desse algoritmo no loop de extensão/procura, o algoritmo ou inferi um plano ou prova que nenhum plano é possível. Podemos dizer que esse algoritmo é válido e completo, pois qualquer plano que o algoritmo encontre é um plano válido e se existe um plano válido então o Graphplan irá encontrar um.

A expansão do Planning Graph se dá da seguinte maneira. Todas as condições iniciais são postas no primeiro nível de proposições do grafo. Para se criar um nível genérico de ações fazemos para cada operador e para cada modo de instanciar as proposições das precondições do operador, inserimos a ação respeitando as relações mutuas de exclusão. Também devemos inserir todas as ações no-op e então, é checada a exclusividade da ação e crias-se uma lista relacionando as exclusivas correspondentes à cada ação.

Para se criar um nível genérico de proposições basta simplesmente olhar para os efeitos adicionados das ações do nível de ações anterior (incluindo os no-ops) o colocá-los no próximo nível de proposições conectando as ações as proposições de forma apropriada.

Devemos citar que as partes do Graphplan que tomam mais tempo a serem executadas são a determinação das relações exclusivas e a busca por um plano na árvore de busca utilizando a busca *backward*.

Dado um Planning Graph, o Graphplan procura por um plano válido usando a estratégia de busca unidirecional para trás (Backward-chaining strategy). Ao contrário de muitos outros algoritmos, o Graphplan utiliza uma aproximação/expansão nível-por-nível, para melhor aplicar as restrições da mutua exclusão. Um outro aspecto da busca do Graphplan é quando um conjunto de (sub)objetivos em um tempo *t* é determinado por ser inatingível, o algoritmo memoriza o que ele aprendeu, armazenando a informação. Assim, quando o Graphplan esta criando um conjunto de sub-objetivos em um tempo *t*, ele primeiro busca a informação se aquele conjunto já foi provado ser inatingível fazendo a procura do algoritmo ter uma velocidade maior.

A seguir temos um exemplo de aplicação do Graphplan. Como vemos temos a declaração da condição inicial. A seguir temos o objetivo, ações e efeitos. Com isso pode-se montar as alternativas de solução, como mostram o exemplo do jantar a seguir, onde o objetivo de um homem que gostaria de fazer uma surpresa para sua esposa é fazer o jantar, fazer um presente e jogar o lixo fora.

- **Initial Conditions**: (and (garbage) (cleanHands) (quiet))
- **Goal**: (and (dinner) (present) (not (garbage))
- **Actions**:
    - Cook :precondition (cleanHands)
      :effect (dinner)

- Wrap :precondition (quiet)
  :effect (present)
- Carry :precondition
  :effect (and (not (garbage)) (not (cleanHands))
- Dolly :precondition
  :effect (and (not (garbage)) (not (quiet)))

O primeiro passo é analisar quais operadores podem ser efetuados, ou seja, quais operadores possuem suas precondições nas condições iniciais. Neste primeiro nível criado vemos à esquerda as condições inicias, a direita o novo nível criado e ao centro os operadores que foram possíveis serem utilizados neste ponto para formar o próximo nível.

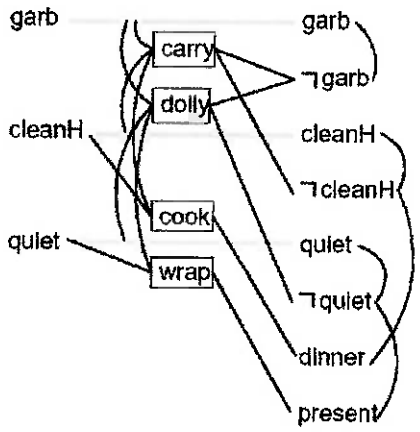


Figura 5.

Após criar os níveis até encontrar os objetivos no último nível de proposições criado, o Graphplan busca a solução do problema, no caso que atenda a (dinner) and (present) and (not (garbage)). Como na etapa anterior não foi possível encontrar a solução desejada, pois as relações de exclusão mutua estavam presentes em alguns operadores e proposições, o Graphplan continua a busca gerando um novo nível como mostra a figura a seguir.

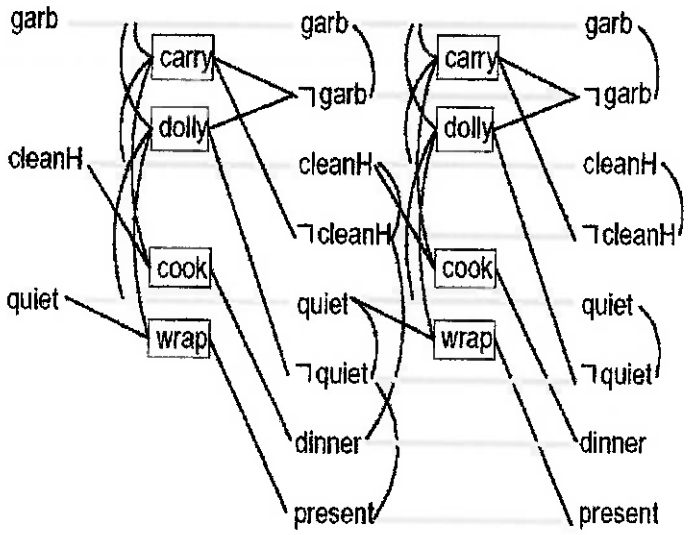


Figura 6.

Vemos que no segundo nível criado os objetivos também aparecem e o algoritmo, em uma busca *backward*, tenta achar o plano e neste ponto a solução foi encontrada respeitando a relação de mutua exclusão assim como mostra o esquema abaixo.

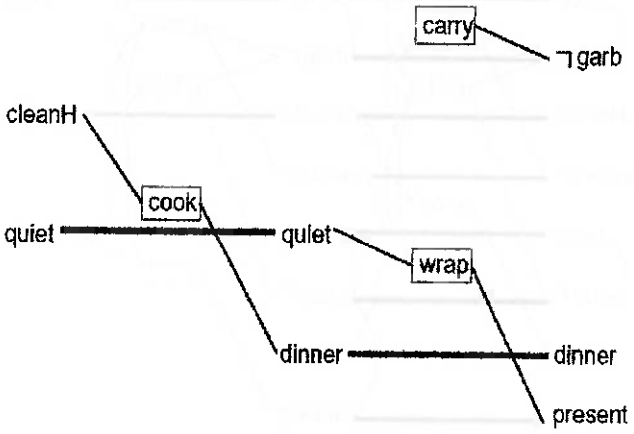


Figura 7.

3.2.4.3 Eficiência

O Graphplan foi comparado com alguns *planners*, tais como Prodigy e UCPOP, na resolução de problemas de planejamento. Seu desempenho foi provado ser muito bom. O tempo de processamento para se chegar a um plano válido foi significativamente menor em relação aos *planners* comparados. Podemos citar alguns fatores principais que levam o Graphplan a obter uma maior eficiência, e estes são:

- Mutua Exclusão: Em vários exemplos, as relações de mutua exclusão são capazes de representar a maior parte das restrições em um problema de planejamento. Propagando essas restrições eficientemente podemos eliminar uma grande parte do espaço de procura.

- Memorização: fixando ações em pontos específicos no tempo, o Graphplan é capaz de armazenar o conjunto de objetivos que prova ser inatingível em certos números de estágios em relação a condição inicial.

- Construindo um *Planning Graph* antes mesmo da busca, o Graphplan acaba evitando o custo envolvido na instanciação durante a fase de busca.

- È interessante notar que a criação do grafo é realmente rápida.

Muitos *planners*, que estão surgindo recentemente, têm tomado o Graphplan como base de seus algoritmos. Os resultados desses algoritmos têm provado ser de alta eficiência na competição de planejamento conhecida como AIPS (*Artificial Intelligence Planning and Scheduling*).

3.2.5 Rede de Petri (RdP)

O conceito de Redes de Petri foi introduzido por Carl Adam Petri em sua tese de doutorado (1962), como ferramenta para descrever relações entre condições e eventos no estudo de protocolos de comunicação entre componentes assíncronos. Embora ocorresse uma ampla divulgação acadêmica ao longo de três décadas, o potencial só foi reconhecido na metade da década de oitenta, onde esta teoria foi usada para implementações práticas nas áreas de informática e manutenção devido à disponibilidade de novos recursos de "hardware" e "software".

A rede de Petri é uma ferramenta matemática e gráfica que oferece um ambiente uniforme para modelagem, análise e projeto de sistemas a eventos discretos. Sua aplicação tem se estendido a uma grande quantidade e variedade de sistemas. Os principais sistemas onde é aplicada esta técnica são: sistemas de comunicação, sistemas de software, simulação e seqüenciação de sistemas flexíveis de manufatura. Em geral, utilizam-se as redes de Petri para modelagem gráfica da estrutura do sistema e de seu comportamento dinâmico.

3.2.5.1 Propriedades de Redes de Petri

Seja uma rede de Petri representada por um conjunto de lugares P, um conjunto de transições T, uma relação de fluxo P x T que representa como os elementos P e T estão associados e M_0 a marcação inicial da rede. Vejamos algumas propriedades:

- ***Alcançabilidade (Reachability)***

A marcação inicial de uma rede (P, T, F, M0) é dada por M_0. Dizemos que uma marcação M_0 é alcançável a partir de M_0 se existe uma seqüência de disparos que transformam M_0 em M_n.

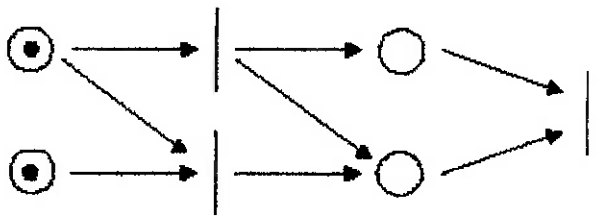


Figura 8.

A rede acima pode alcançar

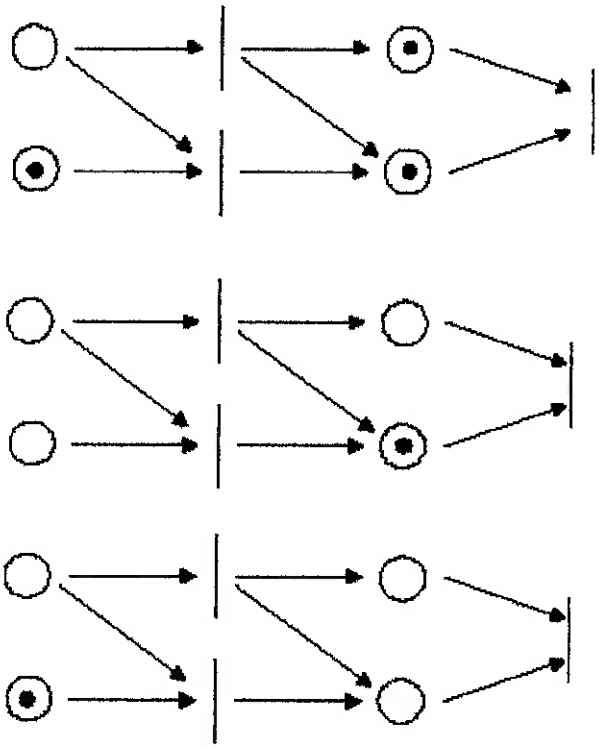


Figura 9.

Existem algoritmos que respondem à pergunta "M_n é alcançável a partir de M_0?" Contudo, estes algoritmos são no mínimo NP-hard. Em alguns casos práticos, incluindo o de planning, a análise das condições necessárias para atingibilidade é mais conveniente.

- ***Limite de Marcas (Boundness)***

Uma rede (P, T, F, M_0) é *k*-limitada se o número de marcas em cada lugar jamais excede *k* considerando todos os estados alcançáveis a partir de M_0.

- ***Vivacidade***

Uma rede (P, T, F, M0) é viva se, em qualquer marcação M_n alcançável a partir de M_0, qualquer transição poderá ser disparada em M_n ou marcações alcançáveis a partir de M_n.

- ***Reversibilidade***

Seja R(M0) o conjunto de estados de uma rede de Petri alcançáveis a partir da marcação M_0. Esta rede será reversível se M_0 for alcançável por qualquer M pertencente a R(M0). Isto é, será sempre possível retornar ao estado inicial.

- ***Cobertura***

Uma marcação M na rede (P, T, F, M_0) é dita coberta se existe M' em R(M0) tal que $M'(p) \geq M(p)$ para cada *p* pertencente a P. Isto é, na marcação M' existem pelo menos tantas marcas em cada lugar quanto em M. Como exemplo, a rede da Figura 3 a) cobre a da Figura 3.12 b).

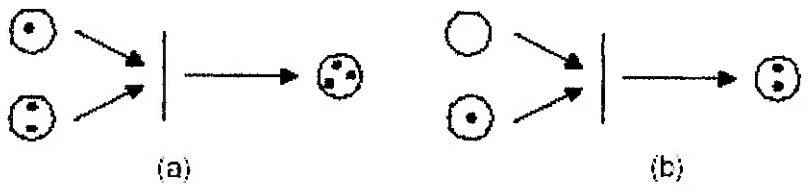


Figura 10.

As abordagens relacionadas com redes de Petri encontradas na literatura podem ser organizadas em três classes fundamentais:

- Redes de Petri básicas ou ordinárias: constituem-se em um modelo elementar adequado para visualizar comportamentos que envolvem paralelismo, sincronização e comportamento de recursos. Este é o modelo básico de rede de Petri que permite extensões segundo a aplicação para a qual se utiliza.

- Reduções de redes de Petri ordinárias: são descrições simplificadas que procuram sintetizar a apresentação gráfica dos modelos, mas ainda podem ser representadas adequadamente por redes de Petri ordinárias. Entre estas estão: redes de Petri generalizadas, redes de Petri de capacidade finita.

- Extensões das redes de Petri ordinárias: correspondem a modelos em que se incorporam regras adicionais de funcionalidade para enriquecer o poder de modelagem. São consideradas três subclasses: modelos equivalentes a máquina de Touring, modelos para sistemas contínuos e híbridos, modelos de sistemas que evoluem em função da ocorrência de eventos externos ou do tempo.

- Redes de alto nível que são geradas à partir de "dobramentos" ou reduções, ou através da inserção do conceito de objetos. Este último ainda é fruto de muita discussão e há um trabalho do D-Lab também nesta direção [18].

Uma outra forma de classificar as redes de Petri é em função de sua aplicação prática. Assim têm-se as redes de Petri interpretadas. Neste tipo de redes são associadas variáveis com significado prático às transições, representando condições e ações existentes no sistema. Tais variáveis podem indicar o estado de atuadores, sensores, etc., permitindo assim, modelar a interação com o ambiente externo.

Existem variações da teoria das redes de Petri, de onde o SFC (Seqüencial Flow Chart) tem sua base teórica. A representação de SED (Sistemas a Eventos Discretos) por Rede de Petri consiste na decomposição do SED em componentes passivos e ativos, a

transição da natureza estática dos componentes para o comportamento dinâmico de um SED, e o inter-relacionamento das representações por redes individuais indicam que a caracterização das diferentes redes é fundamental para que estas sejam devidamente exploradas no controle de SED.

Podemos ter vários tipos de rede de Petri. São alguns dos principais deles:

- Redes condição-evento
- Redes lugar-transição
- Redes de Marcas Individuais (Redes Coloridas)
- Redes Predicado-Transição
- Redes orientadas a objetos

A rede de Petri é uma técnica fundamental e extremamente efetiva para a modelagem de sistemas, a sua base teórica permite o desenvolvimento de poderosas técnicas e ferramentas de análise e síntese de estratégia de controle. Estes tipos de rede de Petri demonstram grande poder de descrição mesmo quando comparadas com outras técnicas de modelagem e análise como: teoria de filas, álgebra min-max, etc.

A seguir relacionaremos algumas características importantes das redes de Petri e conceitos existentes em *Planning* onde mostramos a boa contribuição que essa teoria pode nos trazer se unida à teoria de *Planning* em AI como proposto em nosso objetivo:

- Representa a dinâmica e a estrutura de um sistema segundo o nível de detalhamento desejado (sendo uma rede hierárquica).

- Identifica estados e ações de modo claro e explícito, facilitando com isto a monitoração dos sistemas em tempo real.

- Tem a capacidade para representar de forma natural as características dos Sistemas a Eventos Discretos SED's (sincronização, assincronismo, concorrência, causalidade, conflito, compartilhamento de recursos, etc.) muito importante na formulação de um plano.
- A representação de um plano pode ser traduzida para os conceitos básicos da RdP.

- Os conceitos envolvidos na representação de um plano são perfeitamente expressos em RdP. Por exemplo, uma característica fundamental existente em um plano é a seqüência de ações, e esta característica é muito bem representada pela rede de Petri. Podemos citar outras características de um plano como o paralelismo, ou a sincronização de ações. Essa característica também são de fácil representação pela RdP.

- Associa elementos de diferentes significados numa mesma representação, ou segundo o propósito do domínio (avaliação de desempenho, implementação do controle, etc.).

- Oferece um formalismo gráfico que permite a documentação e monitoração do sistema, facilitando assim o diálogo entre o sistema e projetista que analisa o comportamento do sistema (projetista, operador, gerente, etc.).

- Possui uma semântica formal e precisa que permita que o mesmo modelo possa ser utilizado tanto para análise de propriedades comportamentais e avaliação do desempenho, assim como para a construção de simuladores a eventos discretos e controladores (para implementar ou gerar códigos para controle de sistemas). Além de servir para verificar comportamentos indesejáveis como bloqueio, limitação, etc.

- Incorpora conceitos de modelagem por refinamento ("top down") e do tipo composição modular ("bottom up") através de técnicas como: modularização, reutilização, refinamento.
- Tem uma ótima capacidade de lidar com recursos. Vemos que os *planners* que estão sendo desenvolvidos atualmente tem grande dificuldade de lidar com recursos, pois a lógica clássica por si tem essa dificuldade e muito desses *planners* tem como base a lógica clássica. A habilidade de lidar com recursos é muito procurada pela comunidade de Inteligência Artificial que está muito embasada na lógica.

- Facilidade de representar capacitação. Essa característica está sendo bem explorada pela comunidade de Inteligência Artificial, mas também encontram grandes dificuldades na sua representação, o que torna o algoritmo lento em muitos casos se utilizado as técnicas hoje aplicadas pelos *planners.*

Como uma ferramenta matemática, um modelo em rede de Petri pode ser descrito por um sistema de equações lineares ou outros por modelos matemáticos que refletem o comportamento de sistema ou de um plano, os quais possuem a análise formal do mesmo. Esta característica permite realizar a verificação formal das propriedades comportamentais do sistema e do plano.

3.2.6 Rede de Petri e Inteligência Artificial

A princípio devemos analisar as similaridades e as diferenças entre as duas principais abordagens freqüentemente usadas em automação. Elas são:

- Uma modelagem por rede de Petri para análises, performance de evolução, controle em tempo real e planejamento (planning).
- Técnicas de base lógica para decisões e diagnósticos.

Foi notado que a rede de Petri podia ser traduzida em sistema de produção de regras, e foi provado que programas lógicos podiam ser traduzidos em redes de Transição de Predicados, mas em ambas as traduções são encontradas dificuldades.

A teoria de sistema de produção de regras é baseada na lógica clássica. Tem sido recentemente apontado que a lógica clássica não é consistente com recursos (máquinas, ferramentas, etc.). Como essa noção é uma das bases da teoria de rede de Petri, afirma-se que modelagem de rede de Petri difere de uma representação lógica. A semântica da informação expressa é basicamente diferente e parece razoável combinar essas duas abordagens para obter modelos mais complexos.

A Rede de Petri pode ser vista como uma ferramenta que se caracteriza por ser intermediária entre a pura abordagem declarativa (lógica e regras) e a pura abordagem procedural (seqüência de eventos). Esta é uma grande vantagem em sistemas seqüenciais como, por exemplo, o de manufatura, porque os recursos devem ser lidados de uma maneira procedural (seqüência de estados) e regras de decisão também devem ser usadas.

Embora rede de Petri e sistemas de produção de regras tenham um forte relacionamento, não é possível considerar que a lógica clássica é uma base de unificação comum das duas abordagens. Redes de Petri não podem ser reduzidas a um caso específico de sistema de produção de regras. O principal problema não está na especificação sintática. Quando traduzida uma dada regra ou lógica para uma rede de Petri, nós temos que definir, para cada proposição envolvida e para cada regra, se a proposição se mantêm verdadeira ou não após as aplicações de regras. Isto parece ser de pequena importância para quem trabalha no campo da Inteligência Artificial. Todavia, esse parece ser o único jeito de claramente modelar recursos disjuntivos.

O conceito de recursos prova ser inconsistente com os axiomas básicos da lógica clássica. O que também nos leva a perceber o potencial da combinação de redes de Petri e Inteligência Artificial. Na automação de fábricas o conceito de recursos (ferramentas,

maquinas, etc.), tempo (tempos implícitos como seqüências de operações, tempos explícitos como durações de operações, etc.), regras de decisão (por exemplo, para coordenar eventos), informação simbólica e incertezas (interações com o ser humano, imprecisões provenientes da duração de operações, informações perdidas ou mal conhecidas depois de um acidente, etc.) são necessários para ajudar e cooperar com a atual complexidade de sistemas integrados. Então, combinar as duas abordagens é muito eficiente.[13]

3.2.7 Resultados da Rede de Petri no AIPS

Uma das dos algoritmos baseado no GraphPlan com Redes de Petri que foi exposta no AIPS 2000 é o chamado TokenPlan desenvolvido por Patrick Fabiani e Yannick Meiller na ONERA (centro de Toulouse – France). TokenPlan combina o Planejamento clássico de Inteligência artificial (*Classical AI planning*) com a Teoria de jogo (*Game theory*). A idéia foi utilizar a Teoria de jogo para reduzir o tamanho do espaço de busca. O mesmo teve um bom desempenho se comparado com os demais algoritmos, em alguns domínios e com entradas relativamente não muito grandes, fornecia rapidamente o plano vencendo muitas vezes o vencedor daquele ano, mas para entradas contendo muitas informações estourava o tempo limite de resolução do plano. O algoritmo acabou mostrando a importância da Rede de Petri em problemas de planning.

Mais tecnicamente, o *planner* TokenPlan é uma ferramenta de planejamento baseado no uso da Rede de Petri com Tokens coloridos (rede de Petri colorida – RdP de alto nível). O domínio PDDL (*Planning Domain Description Language*) de entrada é convertido em uma Rede de Petri (lugar corresponde ao predicado e transição corresponde ao operador). Esse algoritmo é caracterizado por efetuar o espaço de busca unidirecionalmente.[10] [11].

Outro planner desenvolvido utilizando rede de Petri é o chamado Petriplan, desenvolvido por Fabiano Silva, Marcos A. Castilho e Luis A. Künzle em 2001, que também utiliza os conceitos do GraphPlan. Esse é um trabalho que também mostrou a

importância do relacionamento do campo do Planejamento com redes de Petri. Neste caso o *planner* utiliza uma rede de Petri lugar/transição acíclica.

Esse algoritmo não chegou a participar de AIPS, pois muitos aperfeiçoamentos precisam ser feitos e não obtiveram resultados inicias satisfatórios e ainda possui alguns problemas com a natureza teórica a serem resolvidos.

4 ANÁLISE DO GRAPHPLAN

Muitos *planners* que estão surgindo recentemente têm tomado o Graphplan como base de seus algoritmos. Os resultados desses algoritmos têm provado ser de alta eficiência na competição de planejamento AIPS. Mas, estudos feitos experimentalmente no laboratório D-LAB do departamento de Engenharia Mecatrônica da Escola Politécnica da USP mostraram que o algoritmo não é eficiente ao lidar com o problema capacitado. Podemos citar um exemplo: no domínio dos elevadores, imaginemos uma situação onde temos 10 andares e 4 passageiros distribuídos nos andares, todos desejando ir ao térreo, e a capacidade do elevador é de três pessoas. O Graphplan demorou aproximadamente 52 minutos para inferir o plano que satisfaz a todos os passageiros. Dependendo do modo como esses passageiros se posicionavam no edifício, o algoritmo leva mais ou menos tempo para fornecer a solução, assim a aplicação de um *planner* como o GraphPlan em um controle real é inviável e impraticável.

Surge então a necessidade de modificá-lo para que alguns conceitos sejam provados e inseridos no algoritmo. Nesta primeira etapa de trabalho no algoritmo, gostaríamos de enriquecer o Graphplan com as seguintes características:

- Um algoritmo mais rápido ao inferir um plano, ou seja, diminuir o tempo de aproximadamente 52 minutos para um tempo mais viável à sua aplicação. Tomaremos como exemplo o domínio do elevador para efeito de comparação de resultados

- Um algoritmo capaz de lidar com capacidade, ou podemos dizer com certa limitação de recursos. No exemplo do elevador podemos citar a capacidade limitada do número de passageiros na cabine.

- Um algoritmo capaz de lidar com lógica de primeira ordem e informações não lógicas (contadores, pilhas, etc). R. Valette e M. Courvoisier em *Petri nets and Artificial Intelligence* defendem o uso de bases não lógicas para tornar mais poderosa a representação de domínios utilizando proposições e rede de Petri.

- Os arquivos de entrada do Graphplan serem reduzidos quanto ao volume de informação mas contendo o mesmo significado e objetivo.

Com a mudança do Graphplan gostaríamos de provar alguns conceitos, estes são:

- O uso de informações não lógicas ao inferir um plano otimiza o processo de busca do Graphplan reduzindo seu tempo. Assim mostrando a importância do uso conjunto de lógica de primeira ordem com base (informação) não lógica.

- O Graphplan mesmo sendo modificado ainda terá sua aplicação em sistemas de tempo real inviável. Assim surge a necessidade de outras bases teóricas na formulação do algoritmo, ou seja, a criação de um novo algoritmo utilizando conceitos como o da rede de Petri que lida muito bem com recursos e capacidade (a lógica de primeira ordem apresenta dificuldade em lidar e expressar recursos).

- O Graphplan é um ótimo algoritmo para ser tomado como referência na criação de um novo algoritmo de Planning.

Para efetuar a modificação no Graphplan foi preciso entender como é feita a entrada de dados e como este armazena as informações. Vale citar que a linguagem de programação do algoritmo Graphplan é C e através de seu código fonte podemos analisar a estrutura de armazenamento de dados de entrada que a partir dela o entendimrnto do resto da programação se torna um pouco mais simples.

4.1 ESTRUTURA DE ARMAZENAMENTO DE DADOS DE ENTRADA

As entradas de dados do Graphplsan são dois arquivos textos, o primeiro contêm as informações sobre os operadores (*ops*) que serão utilizados no decorrer do plano (ações que podem ser tomadas sobre o domínio). O segundo (*facts*) representa a declaração dos parâmetros existentes no domínio, as condições iniciais e os objetivos a serem atingidos a partir das condições iniciais.

Os autores Avrim Blum e Merrick Furst utilizam listas ligadas de estrutura *structs* para armazenar cada informação inserida pelo usuário. Para entendermos essas estruturas vale utilizar esquemas visuais que representam tal estrutura. Analisaremos primeiramente o armazenamento dos operadores.

4.1.1 Operadores (ops)

A *Struct* declarada para os operadores é como vista a seguir:

```
typedef struct OPER {
  char *name;            /* a variável armazena o nome do operador
*/
  param_list params;     /* param_list é uma outra Struct */
  precond_list preconds; /* precond_list é uma outra Struct */
  effect_list effects;  /* effect_list é uma outra Struct */
  instantiation_list insts;  /* armazena as váriaveis */
  struct OPER *next;     /* lista ligada */
} op_s, *op_list, *op_ptr;
```

O tipo param_list que armazena os parametros utilizados no operador, o tipo precond_list que armazena as précondições para o operador ser utilizados e o tipo

effect_list que armazena os efeitos da aplicação do operador são declarado como uma *struct* da seguinte forma:

```
typedef struct FACTLIST {
  token_list item;  /* lista ligada de caracter */
  struct FACTLIST next; /* lista ligada da estrutura */
} fact_list_elt, *fact_list, *precond_list, *effect_list, *param_list;
```

O esquema que pode representar essa estrutura pode ser da seguinte forma:

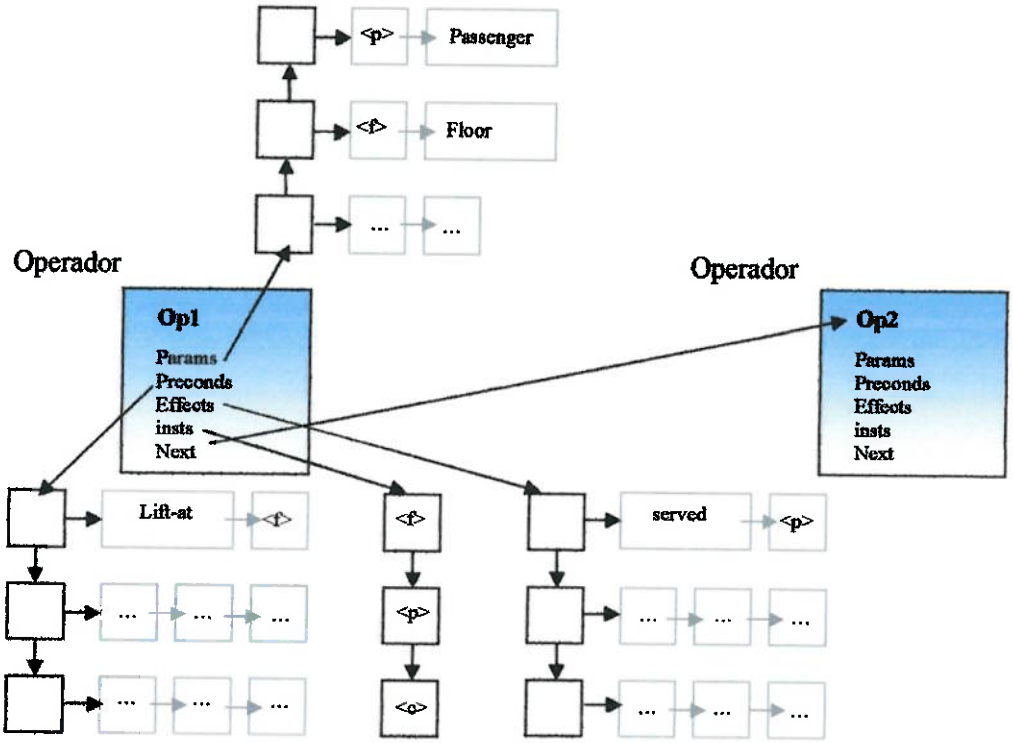


Figura 11.

4.1.2 Fatos: declaração dos parâmetros, fatos iniciais e objetivos (facts)

Os parâmetros os fatos iniciais e os objetivos são todos armazenados separadamente em estruturas como esta a seguir:

A *Struct* declarada para os operadores é como vista a seguir:

```
typedef struct FACTLIST {
  token_list item;  /* lista ligada de caracter */
  struct FACTLIST next; /* lista ligada da estrutura */
} fact_list_elt, *fact_list, *precond_list, *effect_list, *param_list;
```

O esquema que pode representar essa estrutura pode ser da seguinte forma:

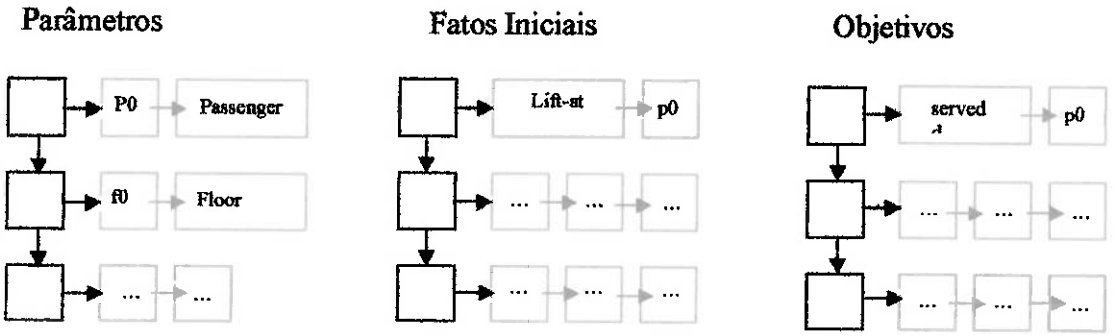


Figura 12.

A partir da analise efetuada sobre o algoritmo podemos iniciar as modificações com bases nos objetivos citados acima.

5 GRAPHPLAN MODIFICADO

5.1 MODIFICAÇÕES NA INTERPRETAÇÃO DOS DADOS DE ENTRADA

5.1.1 A entrada de Fatos (facts)

Para tornar o Graphplan eficiente ao lidar com problemas de capacidade e informações não lógicas eram necessárias modificações para que uma base não lógica trabalha-se juntamente com a lógica de primeira ordem.

A primeira modificação feita foi a separação das informações de entrada dos fatos como sendo não lógicas e lógicas. Essa separação se dá pelo usuário que ao querer inserir informações não lógicas, ao invés de separar suas proposições e informações por parênteses como recomendado para a informação lógica, utiliza chave " { " para iniciar uma proposição. Assim quando o algoritmo lê os dados de entrada, se este observar um parentes " (" saberá que a informação a seguir é lógica de primeira ordem, já se o algoritmo observar a existência de uma chave "{" este saberá que a seguir as informações são não lógicas e devem ser tratadas de forma diferente. O fim da captura de dados após o parentes da esquerda ou a chave da esquerda se dá quando o algoritmo encontra um parêntese da direita ") ", e passa a procurar outro.

A identificação das informações se da apenas pela utilização de uma variável inteiro (*int*) que quando seu valor é atribuído zero (0) indica a existência de informações lógicas (primeira ordem) e quando seu valor é declarado um (1) indica informações não lógicas.

Analisando o domínio dos elevadores podemos perceber que este tem comportamento similares a contadores, e assim como diversos outros domínios que

necessitam do uso de contadores. Com base nessa necessidade criamos uma estrutura *struct* que será associada as informações não lógicas. Essa *struct* (chamada de STACK - pilha) só é criada quando o algoritmo lê o arquivo de fatos (facts) pois é nele onde são declaradas estas estruturas. Vejamos um exemplo da declaração de um pilha realizada no ínicio do arquivo de fatos na região de declarações de parâmetros.

```
(Elevator STACK)
(Building STACK)
(p0 Passenger)
.
.
```

Quando identificado a chave (" { ") declaração dos parâmetros o algoritmo cria a pilha com o nome dado a ela pelo usuário. No exemplo acima duas pilhas são criadas com os nomes "Elevetor" e "Building". Vejamos a *struct* criada para essa finalidade.

```
/* Stack for No-logic stuffs */

      typedef struct STACK {
  char *name;
  int capacity;
  int counter;
  struct STACK next; /* lista ligada de pilhas */
} pilha_nl, *pilha;
```

No mesmo arquivo de fatos (facts) o usuário caracteriza a pilha fornecendo sua capacidade e o contador que indica qual valor esta associado a pilha no instante inicial.

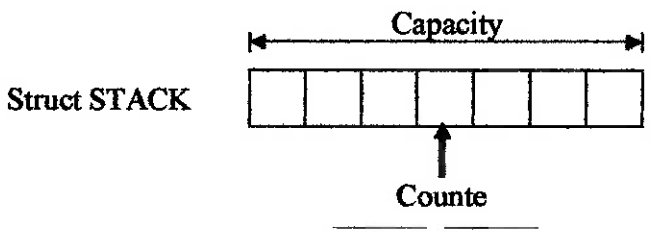


Figura 13.

Para caracterizar uma pilha declarada devemos iniciar com a chave "{" e utilizar palavras pré-determinadas como "*capacity*" e "*with*" (counter). Essa informações devem estar contidas na declaração dos fatos iniciais que são identificadas pela palavra "preconds". Vejamos o exemplo.

```
(preconds
(capacity Elevator 3)
(with Elevator 2)
(capacity Building 5)
(with Building 0)
(origin p0 Building 0)
(destin p0 Building 5))
```

Com esse dados a pilha esta declarada e possui de todas as informações necessária para iniciar a inferência do plano. A utilização da pilha gerada se dá na criação automática de tipos pelo algoritmo, ou seja, o algoritmo criará mais parâmetros na região de declarações dos parâmetros sem que o usuário perceba. Os parâmetros criados automaticamente para o exemplo citado acima seriam:

```
(Building 0)
(Building 1)

(Building 2)
(Building 3)
(Building 4)
(Building 5)
(Elevator 0)
(Elevator 1)
(Elevator 2)
(Elevator 3)
```

Essa informação acrescentada nos auxiliará na inferência do plano quando esta for efetuada.

Com a utilização desta pilha é possível perceber a redução do volume de informações contidas na entrada. Sem modificações a entrada dos fatos para o domínio dos elevadores com grande número de andares se torna muito grande, assim o algoritmo leva algum tempo a ler os dados. Para exemplificar esse problema daremos um exemplo de um prédio de 5 andares e um passageiro a espera. A entrada de dados que define os cinco andares é dada da seguinte forma.

```
(p0 Passenger)
(f0 Floor)
(f0 Floor)
(f0 Floor)

(above f0 f1)
(above f0 f2)
(above f0 f3)
(above f0 f4)
(above f0 f5)
(above f1 f2)
(above f1 f3)
(above f1 f4)
(above f1 f5)
(above f2 f3)
(above f2 f4)
(above f2 f5)
(above f3 f4)
(above f3 f5)
(above f4 f5)

(origin p0 Building 0)
(destin p0 Building 5)
```

```
(lift-at f0))

(effects
(served p0))
```

Podemos perceber que se aumentarmos o número de andares para, por exemplo, 40, o arquivo fica realmente grande comparado a uma representação como a proposta por nós após a modificação. Compare a entrada com 40 andares para o algoritmo modificado.

```
(Building STACK)
(p0 Passenger)

(preconds
(capacity Building 40)
(with Building 0)
(origin p0 Building 0)
(destin p0 Building 5))

(effects
(served p0))
```

Uma observação a ser feita nesse ponto é que a proposição (at-lift f0) é equivalente a informação não lógica {with Building 0).

Realmente a diferença é brutal. O aumento de número de andares não altera a quantidade de informação da entrada proposta no Graphplan modificados e esse fato se torna uma grande vantagem na otimização do algoritmo e sustenta o argumento da utilização de lógica de primeira ordem com base não lógica.

Com a utilização desta estrutura a representação da capacidade, por exemplo, do elevador também se torna mais fácil e seu planejamento também.

5.1.2 A entrada de Operadores (ops)

As modificações realizadas na leitura do arquivo de entrada dos operadores possuem características realmente marcantes.

Uma das mudanças esta na nas declarações de cada operador, lembrando que um operador possui três tipos de declarações sendo elas os parâmetros, as precondições e os efeitos. O Graphplan Modificado deve receber quatro (4) tipos de declarações: Parâmetros ("params"), Precondições Lógicas ("preconds"), Precondições Não Lógicas ("nologic preconds") e os Efeitos ("effects"). A forma com que as informações inseridas nos parâmetros, nas precondições lógicas e nos efeitos não sofreram nenhuma modificação, mas o ponto interessante esta nas precondições não lógicas.

As informações inseridas em *nologic preconds* podem ser comparações do tipo desigualdades entre números (">" ou "<") ou igualdade ("=") ou até mesmo desigualdade (" != "). Podemos comparar variáveis que estão sendo instanciadas e fazer com que essa condição (expressão) faça parte do processo de inferência do operador. Podemos utilizar expressões numéricas simples para validar a inferência do operador, ou seja, sua utilização em determinado estágio. Para entendermos o grande potencial que essa modificação nos traz vejamos o exemplo, continuando a analisar o domínio do elevador.

Vejamos o exemplo da utilização de desigualdade, para isso é dado exemplo do operador *up* do arquivo *ops* para o domínio do elevador onde entendemos o predicado Building representando os andares de um prédio (aqui não estamos utilizando o elevador com capacidade).

```
(operator
 up
  (params
    (<b1> Building) (<b2> Building))
  (preconds
    (with Building <b1>))
```

```
(nologic preconds
  (<b1> < <b2>))
(effects
  (del with Building <b1>) (with Building <b2>)))
```

A expressão `(<b1> < <b2>)` é fundamental no processo de escolha do operador *up* para o uso. É com essa expressão que o algoritmo sabe que só deve escolher andares acima do inferido nas precondições lógicas (no caso o valor que <b1> recebeu ao verificar as precondições no nível de fato anterior) assim inferindo <c2>.

Agora vejamos um exemplo do uso de expressões nas precondições não lógicas, para isso escolhemos como exemplo o problema do foguete (*rocket*) onde existen duas cidades e passageiros querendo viajar de uma cidade a outra, mas o foguete só tem as **capacidade** de levar duas pessoas assim o operador LOAD que faz com que uma pessoa entr no foguete deve incrementar a quantidade de pessoas no foquete, assim devemos utilizar uma expressão para incrementar 1 unidade. Veja o como se declara o operador LOAD:

```
(operator
 LOAD
 (params
   (<object> CARGO) (<rocket> ROCKET) (<place> PLACE) (<c1> Carga)
(<c2> Carga))
 (preconds
   (with Carga <c1>) (at <rocket> <place>)  (at <object> <place>))
 (nolgic preconds
   (<c1> + 1 = <c2>))
 (effects
   (with Carga <c2>) (del with Carga <c1>) (in <object> <rocket>)
(del at <object> <place>)))
```

Para o operador UNLOAD que representa a saída de um passageiro do foguete basta mudarmos as precondições não lógicas para `(<c1> - 1 = <c2>)` que decrementa.

Com essa característica podemos criar várias expressões e podemos gerar precondições criativas para um operador, como por exemplo `(<c1> + <c2> = <c3>)` ou talvez `(<c1> + <c2> > 2)`.

Assim para um operador ser utilizado, este dever ser testado nas precondições lógicas e nas não lógicas.

5.2 MODIFICAÇÕES NO PLANEJAMENTO

As modificações no planejamento (*planner*) estão em constante desenvolvimento e visam atingir os objetivos propostas. O foco da mudança no planejamento é utilizar, de forma adequada, as informações de entrada (proveniente das modificações) para que se possa obter uma otimização no tempo e nas características da busca pelo plano.

O planejamento modificado é executado de forma normal quando não há informações não lógicas nos operadores e leva em consideração a expressão imposta nas precondições não lógicas quando esta existir. Assim quando há uma expressão, o algoritmo procura o valor da variável que estabelece a verdade para aquela expressão e a infere, mas ele não para por ai, a busca continua para todos os valores da variável que tornam aquela expressão verdadeira. Então para todos os valores encontrados cada um acabará gerando um operador carregando o seu valor.

A estrutura não lógica interfere no procedimento normal de planejamento habilitando e desabilitando o uso de operadores e também inferindo as variáveis contidas em cada um.

As modificações foram realizadas de tal forma que o planejador (processo de *planning*) sinta o mínimo possível a existência da estrutura não lógica, não mudando assim a base do algoritmo, pois era necessário manter sua estrutura justamente para a visualização de algumas dificuldades de sua base teórica. Neste trabalho não foi modificado o processo de busca pelo plano após criada a árvore de estados, chamado de

backward, para mantermos suas características básicas e mostrarmos seus problemas, já nossa preferência é o uso de uma busca bidirecional.

6 ANÁLISE DOS RESULTADOS OBTIDOS

Conforme citado anteriormente, o propósito deste trabalho é modificar e otimizar o algoritmo do planner Graphplan, AIPS 98, que utiliza sua base teórica em um importante conceito de grafos, mas sem dinamismo para capacidades e equações algébricas. Vamos analisar as otimizações realizadas anteriormente citadas exemplificando um problema de planejamento de elevadores e comparando com ambos os planejadores (Graphplan original e Graphplan Modificado).

6.1 DESCRIÇÃO DO PROBLEMA DE TESTE

Temos um elevador em um prédio de 11 andares, sendo um deles o térreo (Andar 0), e desejamos planejar a melhor e mais eficiente rota deste elevador de modo a atender todos os três passageiros deste edifício.

Sendo os passageiros p0, p1 e p2, temos que:

- P0 se encontra no Andar 8 e deseja ir ao Andar 0;
- P1 se encontra no Andar 3 e deseja ir ao Andar 1;
- P2 se encontra no Andar 4 e deseja ir ao Andar 0.
- Ainda sabemos que o elevador se encontra inicialmente no Andar 5.

A seguir, mostraremos as declarações dos fatos iniciais para ambos os planners. Podemos reparar que o Graphplan Modificado possui uma estrutura de entrada muito mais enxuta e limpa de se estudar e para este caso é importante notar que não seu "volume" não varia (para um número fixo de passageiros), mesmo variando o numero de entrada .

```
/************************ Graphplan Original - Facts ************************/

Lift Domain: This problem has 3 passenger, 11 floors and 1 lift, to solve the
problem, graphplan needs to achieve the satisfability of the passenger.

(p0 Passenger)
(p1 Passenger)
(p2 Passenger)
(f0 Floor)
(f1 Floor)
(f2 Floor)
(f3 Floor)
(f4 Floor)
(f5 Floor)
(f6 Floor)
(f7 Floor)
(f8 Floor)
(f9 Floor)
(f10 Floor)

(preconds
(above f0 f1)
(above f0 f2)
(above f0 f3)
(above f0 f4)
(above f0 f5)
(above f0 f6)
(above f0 f7)
(above f0 f8)
(above f0 f9)
(above f0 f10)

(above f1 f2)
(above f1 f3)
(above f1 f4)
(above f1 f5)
(above f1 f6)
(above f1 f7)
(above f1 f8)
(above f1 f9)
```

```
(above f1 f10)

(above f2 f3)
(above f2 f4)
(above f2 f5)
(above f2 f6)
(above f2 f7)
(above f2 f8)
(above f2 f9)
(above f2 f10)

(above f3 f4)
(above f3 f5)
(above f3 f6)
(above f3 f7)
(above f3 f8)
(above f3 f9)
(above f3 f10)


(above f4 f5)
(above f4 f6)
(above f4 f7)
(above f4 f8)
(above f4 f9)
(above f4 f10)

(above f5 f6)
(above f5 f7)
(above f5 f8)
(above f5 f9)
(above f5 f10)

(above f6 f7)
(above f6 f8)
(above f6 f9)
(above f6 f10)

(above f7 f8)
(above f7 f9)
```

```
(above f7 f10)

(above f8 f9)
(above f8 f10)

(above f9 f10)

(origin p0 f8)
(destin p0 f0)

(origin p1 f3)
(destin p1 f1)

(origin p2 f4)
(destin p2 f0)

(lift-at f5))

(effects
(served p0)
(served p1)
(served p2))


/************************* Graphplan Modificado - Facts *********************/

Lift Domain MOD: This problem has 3 passenger, 11 floors and 1 lift, to solve
the problem, graphplan needs to achieve the satisfability of the passenger.

(p0 Passenger)
(p1 Passenger)
(p2 Passenger)
(Building STACK)
(0 Building)
(1 Building)
(2 Building)
(3 Building)
(4 Building)
(5 Building)
(6 Building)
```

```
(7 Building)
(8 Building)
(9 Building)
(10 Building)

(preconds

(capacity Building 10)
(with Building 5)

(origin p0 Building 8)
(destin p0 Building 0)

(origin p1 Building 3)
(destin p1 Building 1)

(origin p2 Building 4)
(destin p2 Building 0))

(effects
(served p0)
(served p1)
(served p2))
```

Podemos reparar a notável diferença entre os dois arquivos de fatos e a simplicidade do atual algoritmo modificado. Vejamos agora, as diferenças entre os operadores do Graphplan original e modificado.

```
/************************ Graphplan Original - Ops ************************/

(operator
 get-in
  (params
    (<f> Floor) (<p> Passenger))
  (preconds
    (lift-at <f>) (origin <p> <f>))
  (effects
    (boarded <p>)))
```

```
(operator
 get-out
  (params
    (<f>  Floor) (<p> Passenger))
  (preconds
    (lift-at <f>) (boarded <p>) (destin <p> <f>))
  (effects
    (del boarded <p>) (served <p>)))


(operator
 up
  (params
    (<f1> Floor) (<f2> Floor))
  (preconds
    (lift-at <f1>) (above <f1> <f2>))
  (effects
    (lift-at <f2>) (del lift-at <f1>)))


(operator
 down
  (params
    (<f1> Floor) (<f2> Floor))
  (preconds
    (lift-at <f1>) (above <f2> <f1>))
  (effects
    (lift-at <f2>) (del lift-at <f1>)))

/************************* Graphplan Modificado - Ops *********************/

(operator
 get-in
  (params
    (<p> Passenger) (<b1> Building))
  (preconds
    (with Building <b1>) (origin <p> Building <b1>))
  (nologic preconds
    ( ))
```

```
  (effects
    (boarded <p>) ))


(operator
 get-out
  (params
    (<p> Passenger) (<b1> Building))
  (preconds
    (with Building <b1>) (boarded <p>) (destin <p> Building <b1>))
  (nologic preconds
    ( ))
  (effects
    (del boarded <p>) (served <p>)))


(operator
 up
  (params
    (<b1> Building) (<b2> Building))
  (preconds
    (with Building <b1>))
  (nologic preconds
    (<b1> < <b2>))
  (effects
    (del with Building <b1>) (with Building <b2>)))


(operator
 down
  (params
    (<b1> Building) (<b2> Building))
  (preconds
    (with Building <b1>))
  (nologic preconds
    (<b1> > <b2>))
  (effects
    (del with Building <b1>) (with Building <b2>)))
```

Outro e mais importante destaque que podemos analisar na diferença entre estes dois *planners* é a sua **eficiência de planejamento**. Veja as tabelas abaixo que apresentam parâmetros fundamentais de comparação entre os dois planejadores e seus respectivos planos gerados.

	Graphplan Original				
No. de Passageiros	**Tempo**	**Action Tried**	**Total set-creation step**	**Nodes created**	**Time - step**
1	0,32 seg	4	3	943	5
2	0,94 seg	56	53	1496	9
3	8,31 seg	9676	8817	1545	11
4	***17,28 min***	***1167810***	***1032145***	***1594***	***13***
5	***> 2 horas***				
6	***> 2 horas***				

	Graphplan Modificado				
No. de Passageiros	**Tempo**	**Action Tried**	**Total set-creation step**	**Nodes created**	**Time - step**
1	0,28 seg	4	3	457	5
2	0,74 seg	47	54	749	9
3	1,57 seg	1453	1430	843	11
4	***4,50 seg***	***10265***	***10163***	***892***	***13***
5	***11,82 seg***	***34670***	***34487***	***941***	***15***
6	***41,85 seg***	***141910***	***138288***	***990***	***17***

Segundo a tabela, podemos analisar que para maior número de passageiros neste problema, o Graphplan original tem um crescimento exponencial do tempo gasto para geração do plano enquanto que o Graphplan Modificado possui um crescimento moderado, estável e muito baixo. Outro fator de destaque na comparação entre os *planners* é o número de nós gerados quando o algoritmo acha pela primeira vez os objetivos em um nível *i* de fatos (**nodes created**), podemos perceber que o número de nós criado é baixo comparado com o gerado no Graphplan original. Esse fato mostra que a arvore de busca gerada foi menor, conseqüentemente, temos uma busca mais rápida.

Outro parâmetro de destaque na tabela é o número de ações tentadas para produzir o plano, apesar dos números se tornarem grandes quando se aumento o número

de passageiros, o Graphplan modificado se mostrou estável quanto a este crescimento, já no Graphplan original esse número cresce rapidamente chegando a um milhão em poucos passageiros, o que dificulta a busca pelo plano.

Veja agora qual são os dados de saída (plano) dos algoritmos:

```
/******************** Graphplan Original ********************/

Welcome to graphplan. 'graphplan -h' gives help on command-line args.

give file name for operators: g_op
get-in
get-out
up
down
give file name for initial facts: gfact_3_10
facts loaded.
number of time steps, or <CR> for automatic:
Info type: (2 = max, 0 or <CR> = min):
Other: 'I' = look for irrelevants
       'L' = Lower bound time needed by counting steps
       'B' = Build up to goals
       'E' = Don't do mutual exclusions (for testing)
       'S' = examine subsets:
time: 1, 72 facts and 55 exclusive pairs.
time: 2, 75 facts and 88 exclusive pairs.
time: 3, 75 facts and 58 exclusive pairs.
time: 4, 78 facts and 97 exclusive pairs.
Goals reachable at 4 steps but mutually exclusive.
time: 5, 78 facts and 60 exclusive pairs.
Goals reachable at 5 steps but mutually exclusive.
time: 6, 78 facts and 55 exclusive pairs.
Goals first reachable in 6 steps.
1545 nodes created.
goals at time 7:
  served_p0 served_p1 served_p2
```

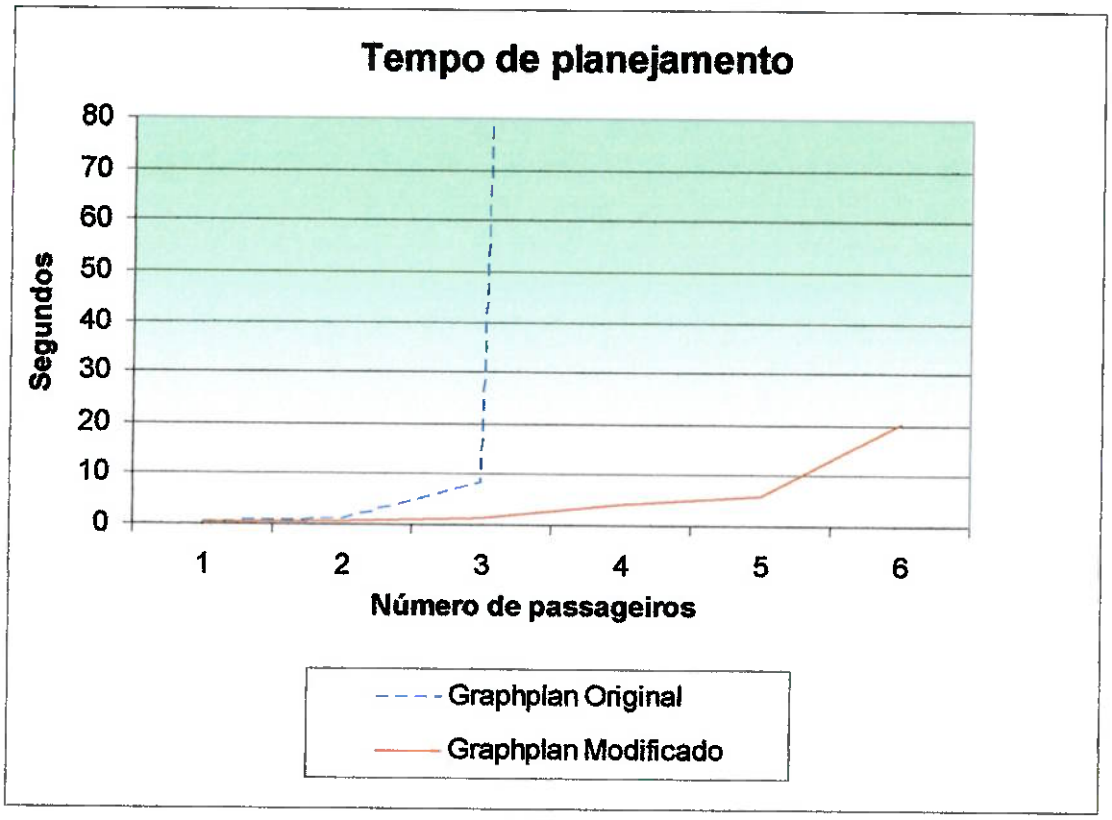
```
Can't solve in 6 steps
time: 7, 78 facts and 55 exclusive pairs.
272 new nodes added.
goals at time 8:
  served_p0 served_p1 served_p2

Can't solve in 7 steps
time: 8, 78 facts and 55 exclusive pairs.
272 new nodes added.
goals at time 9:
  served_p0 served_p1 served_p2

Can't solve in 8 steps
time: 9, 78 facts and 55 exclusive pairs.
272 new nodes added.
goals at time 10:
  served_p0 served_p1 served_p2

Can't solve in 9 steps
time: 10, 78 facts and 55 exclusive pairs.
272 new nodes added.
goals at time 11:
  served_p0 served_p1 served_p2

1 up_f5_f8
2 get-in_f8_p0
3 down_f8_f4
4 get-in_f4_p2
5 down_f4_f0
6 get-out_f0_p0
6 get-out_f0_p2
7 up_f0_f3
8 get-in_f3_p1
9 down_f3_f1
10 get-out_f1_p1
7512 entries in hash table, 1296 hash hits, avg set size 8.
8817 total set-creation steps (entries + hits + plan length - 1).
9676 actions tried
  8.10 secs
```

```
/************************************************************/
/****************** Graphplan Modificado ********************/

Welcome to graphplan. 'graphplan -h' gives help on command-line args.

give file name for operators: op1
get-in
get-out
up
down
give file name for initial facts: fact_3_10
- 1 no logic data created as < Building >
  1 no logic data (pilha) were found;

 Parametro Nao Logico < Building >:
     - Capacidade maxima : 10
     - Posicao atual: 5

facts loaded.
number of time steps, or <CR> for automatic:
Info type: (2 = max, 0 or <CR> = min):
Other: 'I' = look for irrelevants
       'L' = Lower bound time needed by counting steps
       'B' = Build up to goals
       'E' = Don't do mutual exclusions (for testing)
       'S' = examine subsets:
time: 1, 18 facts and 55 exclusive pairs.
time: 2, 21 facts and 88 exclusive pairs.
time: 3, 21 facts and 58 exclusive pairs.
time: 4, 24 facts and 97 exclusive pairs.
Goals reachable at 4 steps but mutually exclusive.
time: 5, 24 facts and 60 exclusive pairs.
Goals reachable at 5 steps but mutually exclusive.
time: 6, 24 facts and 55 exclusive pairs.
Goals first reachable in 6 steps.
```

```
843 nodes created.
goals at time 7:
  served_p0 served_p1 served_p2

Can't solve in 6 steps
time: 7, 24 facts and 55 exclusive pairs.
164 new nodes added.
goals at time 8:
  served_p0 served_p1 served_p2

Can't solve in 7 steps
time: 8, 24 facts and 55 exclusive pairs.
164 new nodes added.
goals at time 9:
  served_p0 served_p1 served_p2

Can't solve in 8 steps
time: 9, 24 facts and 55 exclusive pairs.
164 new nodes added.
goals at time 10:
  served_p0 served_p1 served_p2

Can't solve in 9 steps
time: 10, 24 facts and 55 exclusive pairs.
164 new nodes added.
goals at time 11:
  served_p0 served_p1 served_p2

1 up_5_8
2 get-in_p0_8
3 down_8_4
4 get-in_p2_4
5 down_4_0
6 get-out_p2_0
6 get-out_p0_0
7 up_0_3
```

```
8 get-in_p1_3
9 down_3_1
10 get-out_p1_1
261 entries in hash table, 1180 hash hits, avg set size 5.
1450 total set-creation steps (entries + hits + plan length - 1).
1453 actions tried
  1.57 secs

/*************************************************************/
```

Para uma melhor visualização da comparação dos tempos de planejamento vejamos o gráfico a seguir.



7 CONCLUSÕES

Ao final de toda a análise aqui apresentada e relembrando o objetivo deste trabalho, podemos afirmar que o objetivo específico proposto foi alcançado.

O algoritmo do *planner* Graphplan foi modificado e com ele foi apresentado um "novo" *planner* capaz de lidar com pilhas e bases não lógicas que são exigências de bases teóricas como a da rede de Petri para aplicações no mundo real. Além disso, foi mostrado um algoritmo mais eficiente e estável para grandes problemas de planejamento destacando as limitações do Graphplan como *planner*. Foi possível perceber que o Graphplan possui muitas falhas e é inviável aplicá-lo em situações de controle reais. Assim a necessidade de um novo algoritmo baseado em teorias como a rede de Petri é existente e real e vale termos o Graphplan como um de nossos pilares da base onde iremos nos apoiar para criar esse novo algoritmo.

Como próximas etapas e buscando agora o objetivo geral, proposto nas dissertações de mestrado dos estudantes Rodrigo Chanyon Chen e Tiago Stegun Vaquero, desenvolveremos um algoritmo planejador alinhando as teorias de Redes de Petri e Inteligência Artificial para problemas em tempos reais visando à participação no AIPS 2006.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[1] Ash, David W., Dabija, Vlag G. *Planning for Real Time Event Response Management*. Prentice Hall PTR. 2000.

[2] Blum, A. L., Furst, M. L.. *Fast Planning Through Planning Graph Analysis*. AI, (90:281-300, 1997).

[3] Castilho, Alexandre; Silva, Fabiano; Künzle, Luis A. *Petriplan: a new algorithm for plan generatio (Prelimibary report)*. 2001.

[4] McDermott, Drew. *PDDL - The Planning Domain Definition Language*. 1998.

[5] Gerald Jay Sussman. *The Virtuous Nature of Bugs*. Readings on Plannig, 1990.

[6] Koehler, J. Ottiger D. *An AI-based Approach to Destination Control in Elevator*. 2002.

[7] Koehler, J. *From Theory to Practice: AI Planning forHigh Performance Elevator Control (Extended Abstract)*.2001.

[8] Koehler, J., Schuster, K. Elevator control as Planning Problem. Em S. Chien, S Kambhampati, e C. Knoblock, editores, Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling, pages 331-338. AAAI Press, Menlo Park. 2000.

[10] Mark E. Drummond. *Refining and Extending the Procedural Net*. IJCAI'85, 1985.

[11] Nils J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc. 1980.

[12] P. Fabiani, Y. Meiller. *Planning with Petri nets*. RJCAI'2000, 2000.

[13] P. Fabiani, Y. Meiller. *Planning with tokens*. ECAI-worshop (PuK200), 2000.

[14] Piero Scaruffi. *Inteligência Artificial – Guia para o programador*. Coleção Sistemas. Ed. Presença. 1987.

[15] R. Valette, M. Courvoisier. *Petri nets and Artificial Intelligence*. International Workshop on Emerging Technologies for factory Automation p.218-238, 1992.

[16] J.R.Silva, L.M.Shimada. *Um Sistema de Planejamento Semi-reativo para Sistemas de Produção Baseado em Redes de Petri*. 2°. Simpósio Brasileiro de Automação Inteligente, Curitiba, Setembro de 1995.

[17] J.R.Silva, L.M.Shimada. A Estrururação do Problema de Planejamento em uma Abordagem Baseada em IA e no Formalismo de Redes de Petri. 3°. Simpósio Brasileiro de Automação Inteligente, Vitória, Setembro de 1997.

[18] J.R.Silva, P.M.G. Del Foyo. Towards a Unified View of Petri Nets and Object Oriented Modeling. To appear in 17th. Int. Congress of Mechanical Engineering, São Paulo, December 2003.

[19] Schutzer. *Artificial Intelligence – An Applications – oriented Approach.* Nostrand Reinhold Company, NY. 1987.

[20] Vladimir Lifschitz. *On the Semantics of Strips.* Reading in Planning, 1990.

[21] Weld, Daniel S. *Recent Advances in AI Planning.* Appear in AI Magazine, 1999.

9 ANEXOS

Em anexo seguem os seguintes códigos em C que formam o Graphplan Modificado:

- graphplanMod.h
- graphplanMod.c
- utilitiesMod.c
- hashMod.c
- plannerMod.c
- dummyMod.c

graphplanMod.h

```
/********************** Graphplan ************************************
  (C) Copyright 1995 Avrim Blum and Merrick Furst
      Modified by Rodrigo Chen and Tiago Vaquero

  All rights reserved. Use of this software is permitted for non-commercial
  research purposes, and it may be copied only for that use.  All copies must
  include this copyright message.  This software is made available AS IS, and
  neither the authors nor CMU make any warranty about the software or its
  performance.
*************************************************************************/

#include<stdlib.h>
#include<stdio.h>
#include<ctype.h>
#include<strings.h>

typedef struct VERTEX vertex_s, *vertex_t;
typedef struct TOKENLIST *string_list;

typedef struct EDGE
{
  vertex_t endpt;
  struct EDGE *next;
} edgelist_s, *edgelist_t;

/* HERE ARE SOME ARBITRARY VALUES */

#define MAXMAXNODES 1024  /* max number of nodes per layer of graph */
#define NUMINTS 32 /* MAXMAXNODES / 32 */
#define HSIZE 50   /* For simplicity, fix hash table to have size HSIZE */
#define MAX_TOKENS 15 /* max number of tokens in a tokenlist */
#define MAXGOALS 50   /* max size of a goal set */
#define max_auto 200  /* arbitrary max #time steps for making graph */


/* a vertex in the graph */
struct VERTEX
{
  char *name;
  int hashval;          /* make easier to find in table */
  edgelist_t in_edges;
  edgelist_t out_edges;
  string_list del_list; /* names of things you delete (that aren't preconds) */
  edgelist_t del_edges; /* pointers (one way) to things you delete */
  edgelist_t exclusive;
  int exclusive_vect[NUMINTS];
  edgelist_t excl_in_this_step; /* can't create any pair of these right now */
  int excl_in_this_step_vect[NUMINTS];
  int used;  /* if used, 1+index in array (add 1 to make sure not zero) */
  int is_true;        /* is it made true */
  int cant_do;        /* marked as exclusive of something being done */
  int needed;
  int uid_block;    /* For planning: use uid_mask and uid_block  */
  unsigned int uid_mask;  /* -- unique for time                        */
  vertex_t prev_time;  /* for facts: the fact at prev time step */
  vertex_t next_time;  /* and next time step */
  long rand1;    /* random values associated with fact */
  int junk;  /* use for whatever one wants */
  int is_noop;          /* is it a NOOP? */
  struct VERTEX *next;
};

typedef struct PAIR{ int first; int second; } pair;

typedef vertex_t goal_arr[MAXGOALS];
typedef vertex_t hashtable_t[HSIZE];
```

```
typedef vertex_s element_s, *element_t;
typedef char * token;

typedef struct TOKENLIST {
  token item;
  struct TOKENLIST *next;
} token_list_elt, *token_list;

typedef struct FACTLIST {
  token_list item;
  int det; /* determina se a informacao eh logica(0) ou nao logica(1), visualizado
o
              parentese - logica ou chave - nao logica da esquerda */
  struct FACTLIST *next;
} fact_list_elt, *fact_list, *precond_list, *effect_list, *param_list,
*nologic_list;

typedef struct INSTLIST {
  char *const_part;
  char *var_part;
  char *type_name;
  struct INSTLIST *next;
} instantiation_s, *instantiation_list;

typedef struct OPER {
  char *name;
  param_list params;
  precond_list preconds;
  effect_list effects;
  nologic_list nologic;
/*   int num_preconds, num_effects, num_params; NOT NEEDED ANY MORE */
  instantiation_list insts;  /* use to store the variables */
  struct OPER *next;
} op_s, *op_list, *op_ptr;


/* Estrutura criada para amazenar as informacoes nao logicas,
   todas variaveis nao logicas serao armazenadas em pilhas */
/* Stack for No-logic stuffs */

typedef struct STACK {
  char *name; /* Nome do dado nao logico , pilha */
  int capacity;  /* Capacidade, limite da pilha */
  int counter;  /* Valor que indica posicao na pilha */
  struct STACK *next;
} pilha_nl, *pilha;



/****defines*****/
#define SAME 0
#define DIFFERENT (-1)
#define PARAM "params"
#define PRECOND "preconds"
#define EFFECT "effects"
#define CAPACITY "capacity"
#define WITH "with"
#define SUM "sum"
#define MINUS "minus"
#define OPR "operator"
#define DELETE "del"
#define LISPEXT ".lisp"
#define is_const(x) (*(x) != '<')
#define is_var(x) (*(x) == '<')
#define OK 1
#define LEFT_PAREN 2
```

```
#define RIGHT_PAREN 3
#define LEFT_KEY 4
#define RIGHT_KEY 5
#define max(x,y) ((x) > (y) ? (x) : (y))
#define min(x,y) ((x) > (y) ? (y) : (x))
#define equal_tokens(x,y) (!strcmp((x),(y)))
#define YES 1
#define NO  0
#define NEW_INSTS 2
#define CONNECTOR '_'
#define TRUE 1
#define NOOP "noop"
#define IS_NOOP(x) ((x)->is_noop)

/***********function prototypes: graphplan.c***************/
int create_graph(op_list olist, fact_list flist, int totaltime);
void create_graph_layer(op_list olist);
op_list load_ops(FILE *fp);
int load_fact_list(FILE *fp, fact_list *fptr);
void make_graph_piece(op_ptr op, int time);
void do_operator(hashtable_t htable, fact_list current_facts, op_ptr op,
precond_list p, nologic_list nologic_prec, int time);
void remove_unneeded_vertices(int time);
int can_stop(int time);
void make_copy(int time);
op_list load_prodigy_ops(FILE *fp);
fact_list useful_facts(op_list ops, fact_list f);
void print_cant_do(int time);
fact_list load_types(FILE *fp);
void do_final_viewing(void);


/***********function prototypes: hash.c********************/
element_s * lookup_from_table(hashtable_t htable, char *key);
element_t insert_token_list(hashtable_t htable, token_list t);
element_t insert_into_table(hashtable_t htable, char *key);
void delete_unneeded(hashtable_t htable);
void delete_from_table(hashtable_t htable, char *key, vertex_t to_kill);
edgelist_t insert_edge(edgelist_t e, vertex_t v);
element_t get_next(hashtable_t h, int flag);
/***********function prototypes: utilities.c***************/
char *make_noop_string(char *str);
fact_list make_list_from_htable(hashtable_t h);
instantiation_list insert_inst(char *v, char *c, char *t, instantiation_list i);
int instantiate_into_string(token_list t,
                    instantiation_list inst, char str[], int failflag);
int read_item(FILE *fp, char *str);
void make_op_string(op_ptr op, char *str);
void do_error(char *s);
int equal_facts(token_list f1, token_list f2);
pair find_mutex_facts(hashtable_t harr[], int time);
void find_currently_mutex_facts(void);
int are_facts_exclusive(vertex_t p, vertex_t q);
void find_all_mutex_ops(hashtable_t harr[], int time);
void find_mutex_ops(vertex_t v, int time);
int avoidable(vertex_t op);
int are_mutex(vertex_t v1, vertex_t v2);
void print_graph(hashtable_t *fact_arr, hashtable_t *op_arr, int len);
void read_initial_comments(FILE *fp);
void my_free(void * p);
void * my_alloc(int size);
void print_alloc(void);
void completely_free_fact_list(fact_list l);
void free_token_list(token_list l);
int compare_and_instantiate(token_list, token_list,instantiation_list);
token_list token_list_from_string(char str[]);
```

```
void set_uid(vertex_t v, int id);
/***********function prototypes: planner.c****************/
int do_plan(int maxtime);
void print_plan(int maxtime);
void print_info(int len);
int try_facts(vertex_t f1, vertex_t f2, int time);
/***********others (viewing)*************/
void setup_viewer(void);
void wait_until_left(void);
void reset_viewer(int max_time);
void draw_fact(vertex_t v, int time, int flag);
void draw_op(vertex_t v, int time, int flag, int thick);

/**************vbles*************/
extern int num_deleted;
extern int num_created;
extern int DEBUG_FLAG;
extern int MAXNODES;            /* MAXNODES is the maximum number of nodes
                           allowed per level. */


/***********DECLARACAO DA PILHA P ******************/
 pilha p; /* pilha cabeca */
 int numero_pilhas; /* numero de pilhas existentes no arquivo facts */

 int separa; /* guarda se eh um parenteses ou uma chave */
```

graphplanMod.c

```
/*********************** Graphplan ************************************
  (C) Copyright 1995 Avrim Blum and Merrick Furst
      Modified by Rodrigo Chen and Tiago Vaquero

  All rights reserved. Use of this software is permitted for non-commercial
  research purposes, and it may be copied only for that use.  All copies must
  include this copyright message.  This software is made available AS IS, and
  neither the authors nor CMU make any warranty about the software or its
  performance.
************************************************************************/

#include "graphplanMod.h"
#include <sys/time.h>

/* The code consists of the following main parts.

1. Reading in the operator and fact files.  Note: we take 3 formats.
In the non-prodigy formats, everything up to first left paren is a
comment. Note: there is no special understanding of "not"s: these are
just viewed as another word.  In the non-prodigy formats, "(del blah
blah)" means to delete "(blah blah)".

ATENCÃO: As modificacoes do novo algoritmo nao permite prodigy formats.
Note: the code is lex.yy.c and y.tab.{c,h} is only for reading
prodigy-style files.

2. Creating the graph.

The layered graph will have facts-at-time-1 as the first layer,
then ops-at-time-1, then facts-at-time-2, etc.  Each layer is stored
in a hash table of vertex_s structures.  These structures have a
"name" field which is the key, and the hash functions return a pointer
to the structure.  (The reason for doing it this way is that sometimes
we want to access a vertex by using its name.) The graph is created in
a forward pass until all goals appear and none are listed as being
mutually exclusive.

  Creating the graph involves the following main steps.

  A. Performing instantiations.  The code for this part is a bit
messy, but conceputally, it's just performing the task: given a list
of facts and an operator, find all the ways in which that operator can
be instantiated so that its preconditions are in that list.  What is
confusing is that we have two different ways of storing the names of
things.  One way is as a token list (easier for instantiations) and
the other way is as a string with words connected by underscores (how
they are stored in the graph).

  B. Calculating mutual exclusions.  These propagate and will be used
  to speed up the later search. We will also use this to make the
  graph smaller (in particular, if an operator needs two facts that
  are mutually exclusive, then we don't put it in the graph.)

  Also, once the graph has "leveled off" (2 adjacent fact layers have
  the same number of facts and the same number of mutual exclusions)
  then all future layers will be the same.  So, we can just copy and
  don't need to instantiate any more.

3. Doing the search.  This is just a simple backward chaining, done
  level-by level to make most use of the exclusions.  Unsolvable goal
  sets at a given time step are stored in a hash table so that we can
  fail more quickly next time.  Most of the flags to the user have to
  do with this part of the planning.  E.g., We can do a lowerbounding
  check based on the principle that if there are 10 goals no two of
  which can be made true in the same time step then we will need at
```

```
   least 10 steps.  Or, we can check subsets of a current goal set to
   see if any of them has been stored in the unsolvable sets hash
   table.

   Right now, there are no operator-ordering or goal-ordering heuristics,
   except that NOOPs are always tried first.  I.e., If there are several
   ways of making fact F true at time T, and one such way is to have F be
   true at time T-1, then that way is tried first.  This seems to make
   plans look nicer.

*/

/* GLOBAL VARIABLES */

char junk[100];
fact_list the_types;           /* list of all the different type decls */
fact_list initial_facts, the_goals; /* initial facts and goals */
hashtable_t *fact_table, *op_table;  /* the arrays of hash tables */
hashtable_t useful;  /* USED for "do_irrel": CHECK USEFUL FACTS. */
hashtable_t types_table;  /* holds the types */

extern int hash_hits, hash_inserts; /* entries in plan hash table */

/* these are defaults */
int MAXNODES = 256;  /* default MAX number of nodes at any given time step. */
int DEBUG_FLAG = 0, do_memo = 0, do_subsets=0,
    do_noexcl = 0, do_buildup = 0, do_greedy = 0;
int oldstyle = 0;  /* type of ops file being read in */
int do_irrel = 0, going_backwards = 0;/* use for finding irrelevant features */
int do_lower = 0;  /* try to immediately  prove a set is notpossibly doable */
int xviewing = 0;  /* graphical animation */
int do_completeness = 1; /* do completeness check */

/* other flags, counters, etc. */
int same_as_prev_flag = 0;  /* graph layer is same as previous time step */
int first_full_time, num_hashes[2] = {-1,0}; /* for checking completeness */
/* num_hashes[0] = previous count, and num_hashes[1] = current count */
extern int number_of_recursive_calls, number_of_actions_tried;
int bvec_len = 1;  /* max facts in a time step / 32 */
int num_goals;

int instrs(void)
{
  printf("command line args.  Use any of: \n \
          -h                for this list \n \
          -o <op file>      to specify operator file\n \
          -f <fact file>    to specify fact file\n \
          -t <integer>      to specify a fixed number of time steps\n \
          -i <info level>   to specify info level 1 or 2 (default is 0)\n \
          -O <option list>  to specify options you want\n \
          -M <integer>      to specify alternative max nodes in a time step\n \
                             (default is 256)\n \
          -d                give default values to everything not specified\n\
\n    for example: graphplan -o fixit_ops -f fixit_facts1 -O IL -d \n\
          \n\n");
  return 0;
}


int main(int argc, char *argv[])
{
  op_list ops;
  int i,max_time=0, auto_stop = 0, old_num_created, givedef=0;
  FILE *fp;
  char opfile[50], factfile[50], option[10];
  struct timeval start, end;
```

```
  pilha p_mostra;

  /* Variaveis criadas com intuito de diminuir o tamanho da entrada, nao
necessitando
     declaracao de todos os elementos da pilha no arquivo facts */
  /*int contar;
  int alto;
  fact_list auxiliar;
  token_list tokenaux;
  token_list tokenaux2;
  char *nomeado;*/

  if (argc == 1) {
    printf("Welcome to graphplan. 'graphplan -h' gives help on command-line args.
\n\n");
  }
  option[0] = opfile[0] = factfile[0] = '\0';
  DEBUG_FLAG = -1;
  for(i = 1; i < argc; ++i) {
    if (argv[i][0] != '-') continue;
    if (argv[i][1] == 'h') {instrs(); exit(0);}
    if (argv[i][1] == 'x') xviewing = 1;
    else if (argv[i][1] == 'd') givedef = 1;
    else if (i == argc - 1) do_error("command line args not in proper format");
    else if (argv[i][1] == 'o') strcpy(opfile,argv[i+1]);
    else if (argv[i][1] == 'f') strcpy(factfile,argv[i+1]);
    else if (argv[i][1] == 't') sscanf(argv[i+1],"%d",&max_time);
    else if (argv[i][1] == 'i') sscanf(argv[i+1],"%d",&DEBUG_FLAG);
    else if (argv[i][1] == 'O') strcpy(option,argv[i+1]);
    else if (argv[i][1] == 'M') sscanf(argv[i+1],"%d",&MAXNODES);
  }

  if (MAXNODES > MAXMAXNODES)
    do_error("Sorry, can't handle that large MAXNODES");


  /*LOAD IN OPS AND FACTS FILES. If ends in ".lisp", assume prodigy style*/
  if (!opfile[0]) {
    printf("give file name for operators: ");
    gets(opfile);
  }
  if ((fp = fopen(opfile,"r")) == NULL) do_error("cant load operator file");
  if (strcmp(opfile+max((strlen(opfile)-strlen(LISPEXT)),0),LISPEXT) == SAME);

   /******* Modificacao: Elimina o uso de .LISP ********
            ops = load_prodigy_ops(fp);
    ************************************************** */

  else
    ops = load_ops(fp);
  if (!ops) do_error("illegal ops file");

  if (!factfile[0]) {
    printf("give file name for initial facts: ");
    gets(factfile);
  }
  if ((fp = fopen(factfile,"r")) == NULL) do_error("cant load facts file");
  numero_pilhas = 0;
  the_types = load_types(fp);  /* load in types */
  printf("  %d no logic data (pilha) were found;\n\n", numero_pilhas);

  /* Devemos criar aqui um outro load_token2 que identifica nao logico e insere na
pilha sua caracteristicas
     ou cria uma funcao de busca e vai inserindo quando vai achando as
caracteristicas. Nao podemos usar o load_fact_list e o load token
     como insercao de caracterisca na pilha pois vamos usar outras vezes essa
```

```
 funcao e isso pode atrapalhar */

   load_fact_list(fp, &initial_facts);  /* load in actual facts */
   read_item(fp,junk); /* left paren */
   read_item(fp,junk); /* "effects" */


   num_goals = load_fact_list(fp, &the_goals);  /* load in the goals */
   if (num_goals > MAXGOALS) do_error("MAXGOALS too small");

   /************** Mostrando o Banco de dados de pilhas Nao Logicas
 ********************/

    /* Diminuir entrada, conforme 10 linhas abaixo */
    /* while (the_types->next != NULL) the_types = the_types->next; */

    /* Imprimindo as pilhas reconhecidas - nome, capacidade e posicao */
     p_mostra = (pilha) malloc(sizeof(pilha_nl));
     p_mostra = p;

     while (p_mostra != NULL) {
        printf(" Parametro Nao Logico < %s >:\n", p_mostra->name);
        printf("      - Capacidade maxima : %d \n", p_mostra->capacity);
        printf("      - Posicao atual: %d \n\n", p_mostra->counter);

  /************ Ainda falta ser trabalhado para diminuir entrada ************/
  /* Evitar a necessidade de declarar todos os elementos da pilha estudada */

       /* alto = p_mostra->capacity;
        printf("%d\n", alto);
        for(contar = 0; contar <= alto; contar = contar + 1) {
          printf("chegou\n");
          tokenaux = (token_list) malloc(sizeof(token_list_elt));
          tokenaux->item = (char *) calloc(50,sizeof(char));
          printf("chegou\n"); */
          /*nomeado = (char *)contar;
          printf("nomeado: %s\n", nomeado);*/
          /*strcpy(tokenaux->item, (char *)contar); */
          /*strcpy(tokenaux->item, p_mostra->name);*/
          /*printf(" < %s >  ", tokenaux->item);
          tokenaux2 = (token_list) malloc(sizeof(token_list_elt));
          tokenaux2->item = (char *) calloc(50,sizeof(char));
          strcpy(tokenaux2->item, p_mostra->name);
          printf(" < %s >\n", tokenaux2->item);
          tokenaux->next = tokenaux2;
          tokenaux2->next = NULL;
          insert_token_list(types_table, tokenaux);
          auxiliar = (fact_list) malloc(sizeof(fact_list_elt));
          auxiliar->item = tokenaux;
          auxiliar->next = NULL;
          the_types->next = auxiliar;
          the_types = the_types->next;
        }*/

  /*************************************************************************/

        p_mostra = p_mostra->next;
     }

  /*************************************************************************/

  printf("facts loaded.\n");

/*  if (!givedef && !xviewing) {
 *    printf("X animation? (<CR> for no): ");
 *    gets(junk);
```

```
*    if (*junk == 'y') xviewing = 1;
* }
*/
 if (xviewing) setup_viewer();

 if (givedef && max_time == 0) auto_stop = 1;   /* NUMBER OF TIME STEPS */
 else if (max_time == 0) {
   printf("number of time steps, or <CR> for automatic: ");
   gets(junk);
   if (sscanf(junk,"%d",&max_time) != 1) {
     auto_stop = 1;
     max_time = 0;
   }
 }

 if (givedef && DEBUG_FLAG == -1) DEBUG_FLAG = 0; /* INFORMATION TO USER */
 else if (DEBUG_FLAG == -1) {
   printf("Info type: (2 = max, 0 or <CR> = min): ");
   gets(junk);
   if (*junk == '2') DEBUG_FLAG = 2;
   else if (*junk == '1') DEBUG_FLAG = 1;
   else DEBUG_FLAG = 0;
 }

 /* OTHER OPTIONS */
 if (option[0] == '\0' && !givedef) {
   printf("Other: 'I' = look for irrelevants\n");
   printf("       'L' = Lower bound time needed by counting steps\n");
   printf("       'B' = Build up to goals\n");
/*   printf("         'H' = Try a heuristic ordering of subgoals\n");  */
   printf("       'E' = Don't do mutual exclusions (for testing)\n");
   printf("       'S' = examine subsets:      ");
   gets(option);
 }
 do_memo = 1;  /* ALWAYS DOING MEMOIZING */
 for(i=0; option[i] != '\0'; ++i) {
   if (option[i] == 'S') {do_subsets = 1; do_completeness = 0;}
   if (option[i] == 'I') do_irrel = 1;
   if (option[i] == 'B') do_buildup = 1;
   if (option[i] == 'G') do_greedy = 1; /* Just for experimentation */
   if (option[i] == 'L') {
     do_lower = 1; do_completeness = 0;
   }
   if (option[i] == 'E') do_noexcl = 1;
 }
 if (do_completeness == 0) printf("turning off completeness check.\n");
 /* BEGIN TIMING */
 gettimeofday(&start,0);

 /* DO PRE-PREPROCESSING IF DESIRED */
 if (do_irrel) initial_facts = useful_facts(ops, initial_facts);

 /* MAKE GRAPH AND START PLANNING */
 max_time = create_graph(ops, initial_facts, max_time); /* treat 0 as auto*/
 if (max_time == -1) {
   if (same_as_prev_flag)
     printf("Problem not solvable: can't even reach non-mutex goals.\n");
   else
     printf("Can't get there from here in allotted time.\n");
 } else if (!auto_stop) {
   if (DEBUG_FLAG > 0) print_info(max_time);
   if (do_plan(max_time)) print_plan(max_time);
   else print_cant_do(max_time);
 } else {
   for(; max_time <= max_auto; max_time++) {
     if (DEBUG_FLAG > 0) print_info(max_time);
```

```
      /* try doing plan */
      if (do_plan(max_time)) {
      print_plan(max_time);
      break;
      } else {
      if (max_time == max_auto) print_cant_do(max_time);
      else printf("Can't solve in %d steps\n",max_time);
      if (same_as_prev_flag) { /* check for true unsolvability */
        if (num_hashes[0] == num_hashes[1] && do_completeness) {
          printf("Problem not solvable.\n");
          break;
        } else {
          num_hashes[0] = num_hashes[1];
        }
      }
      }
      /* create next level (unless at very end) */
      if (max_time != max_auto) {
      old_num_created = num_created;
      create_graph_layer(ops);
      printf("%d new nodes added.\n",num_created - old_num_created);
      }
    }
  }
  gettimeofday(&end,0);
  if (end.tv_sec - start.tv_sec < 60)
    fprintf(stdout,"  %.2f secs\n",end.tv_sec + end.tv_usec/1000000.0 -
          (start.tv_sec + start.tv_usec/1000000.0));
  else
    fprintf(stdout,"  %d min, %d seconds\n",(end.tv_sec - start.tv_sec) / 60,
          (end.tv_sec - start.tv_sec)%60);

  if (xviewing) {
    do_final_viewing();
  }
  return 0;
}


/* For printing, removes vertices that could never be reached by planner */
void remove_unneeded_vertices(int time)
{
  fact_list temp;
  vertex_t vert, goal;
  edgelist_t edge;
  char str[100];

  /* begin by marking goals as needed (even required) at the end*/
  for(temp=the_goals; temp; temp = temp->next) {
    instantiate_into_string(temp->item, NULL, str,1);
    if ((vert = lookup_from_table(fact_table[time],str)) == NULL)
      do_error("goal not found in R.U.V.");
    vert->needed = 2;
  }
  /* now work backwards from end.  An operator is needed if it has a
   * needed add-effect and no add-effects exclusive of a required fact.
   * NOOPs are needed if their effect is needed
   * A fact is needed if it has an edge into
   * a needed operator or NOOP (if it's needed in a later time step). */
  for(--time; time >= 0; --time) {
    /* do ops */
    get_next(op_table[time],0);                    /* INIT */
    while((vert = get_next(op_table[time],1)) != NULL) {
      vert->needed = 0;
      if (IS_NOOP(vert)) {
      if (vert->out_edges->endpt->needed) vert->needed = 1;
```

```
      } else {
      for(edge = vert->out_edges; edge; edge = edge->next)
        if(edge->endpt->needed) vert->needed = 1;
      }
    }

    /*do facts */
    get_next(fact_table[time],0);                    /* INIT */
    while((vert = get_next(fact_table[time],1)) != NULL) {
      vert->needed = 0;
      for(edge = vert->out_edges; edge; edge = edge->next) { /* incl' NOOP */
      if(edge->endpt->needed) {
        vert->needed = 1; break;
      }
      }
      goal = lookup_from_table(fact_table[time+1],vert->name);  /*kill this??*/
      if (goal->needed == 2 && goal->in_edges->next == NULL) { /*pass by NOOP*/
      vert->needed = 2;
      if (DEBUG_FLAG>2) printf("%s required, time %d\n",vert->name,time);
      }
    }
  }
}


/***********creating the graph*************************************
 *****************************************************************/


/* this routine makes a layer of noops.  Assumes next fact already in table.
 */
void make_noop_layer(int time)
{
  vertex_t noop,v,w;
  get_next(fact_table[time],0);  /**INIT**/
  while((v = get_next(fact_table[time],1)) != NULL) {
    w = v->next_time;
    noop = insert_into_table(op_table[time],make_noop_string(v->name));
    noop->is_noop = 1;  /* SAY THAT IT'S A NOOP */
    noop->in_edges = insert_edge(noop->in_edges, v);
    v->out_edges = insert_edge(v->out_edges,noop);
    noop->out_edges = insert_edge(noop->out_edges,w);
    w->in_edges = insert_edge(w->in_edges,noop);
  }
}

/***********create the graph*********************************
* Given a list of ops, a list of initial facts and a length
* of time, we create the graph.
************************************/


/* This routine creates a single layer: op_table[time] and fact_table[time+1].
   Assumes is always called on consecutive time steps.  To make sure
   I don't forget this, "time" is a static variable.
 */
void create_graph_layer(op_list olist)
{
  static int time = 0;
  int i;
  static pair fact_summary, old_fact_summary;
  fact_list flist;
  op_ptr op;
  vertex_t v,w;
  if (time == 0) fact_summary.first = fact_summary.second = 0;
  if (same_as_prev_flag) make_copy(time);
```

```
  else {
    /* first copy facts over to next time step */
    get_next(fact_table[time],0);  /**INIT**/
     /* printf("\n");*/
    while((v = get_next(fact_table[time],1)) != NULL) {
     /* printf("Fato :%s no time [ %d ] \n", v->name, time); */
      w = insert_into_table(fact_table[time+1],v->name);
      w->prev_time = v;
      v->next_time = w;
    }
    /* get fact list from the hash table (INEFFICIENT!) */
    flist = make_list_from_htable(fact_table[time]);
    /* do ops */
    for(op = olist; op != NULL; op = op->next)
      do_operator(fact_table[time], flist, op, op->preconds, op->nologic, time);
    completely_free_fact_list(flist);

    /* make the noop layer.  NOTE: doing it HERE so that the noops will be
     the first ones in the list.  Makes it easier for the planner to
     try them FIRST. */
    make_noop_layer(time);

    /* set up uids and rands*/
    get_next(op_table[time], 0);  /* INIT */
    for(i=0;  (v = get_next(op_table[time],1)) != NULL; ++i) {
      /* printf("Ops: %s no time [ %d ] \n", v->name, time); */
      if (i >= MAXNODES) do_error("Too many ops. Need to increase MAXNODES");
      set_uid(v,i);
    }
    get_next(fact_table[time+1],0);  /**INIT**/
    for(i=0; (v = get_next(fact_table[time+1],1)) != NULL; ++i) {
      if (i >= MAXNODES) do_error("Too many ops. Need to increase MAXNODES");
      v->rand1 = random(); /* these are used in memoizing */
      set_uid(v,i);        /* everybody needs a unique ID */
    }

    /* find mutually exclusive ops */
    find_all_mutex_ops(op_table, time);

    /* find mutually exclusive facts */
    old_fact_summary = fact_summary;  /* remember state of last time step */
    fact_summary = find_mutex_facts(fact_table, time+1);
    if (do_lower) find_currently_mutex_facts();  /* NEW */
    if (fact_summary.first == old_fact_summary.first &&
      fact_summary.second == old_fact_summary.second) {
      same_as_prev_flag=1;
      first_full_time = time; /* for checking completeness */
    }
  }
  /* setting bit vector length */
  if (bvec_len * 32 < fact_summary.first) bvec_len = 1 + fact_summary.first/32;
  printf("time: %d, %d facts and %d exclusive pairs.\n",
       time+1, fact_summary.first, fact_summary.second);

  ++time;  /* INCREMENT TIME... */
}

/* If maxtime=0, then treat this as auto.  Keep going until "can_stop"
 * Returns max time used (if not called with 0, just same argument)
 * Returns -1 if can't get there from here (i.e., goals not in graph or
 * not mutex)
 */
int create_graph(op_list olist, fact_list flist, int maxtime)
{
  int time = 0, i, autostop = !(maxtime);
  vertex_t v;
```

```
  num_created = 0;  /* GLOBAL: how many nodes created */
  if (autostop) maxtime = max_auto;

  /* allocate space */
  fact_table = (hashtable_t *) calloc(maxtime+1,sizeof(hashtable_t));
  op_table = (hashtable_t *) calloc(maxtime,sizeof(hashtable_t));

  /* load in initial facts. */
  for(i=0; flist != NULL; flist = flist->next) {
    v = insert_token_list(fact_table[0],flist->item);
    v->rand1 = random();
    set_uid(v,i++);
  }
  /* make the graph */
  for(time=0; time < maxtime; ++time) {
    /* for auto case, see if can stop */
    if (autostop && can_stop(time)) {
      printf("Goals first reachable in %d steps.\n",time);
      break;
    }
    if (autostop && same_as_prev_flag) break; /* Will never reach goals */
    create_graph_layer(olist);   /* CREATE LAYER OF GRAPH */

  }

  printf("%d nodes created.\n", num_created);
  if (can_stop(time)) return time;
  else return -1;
}

edgelist_t insert_edge_at_end(edgelist_t e, vertex_t v);

/*****copy op_table[time-1] to op_table[time] and fact_table[time] to
  fact_table[time+1]. (not exactly copies of course since need to point to
  objects at subsequent time step.)   This is done in the case that
  fact_table[time] was EXACTLY like fact_table[time-1], meaning that there
  were both the same number of ops and the same number of mutex's.

  Of course, rands are different.

  Also, the ordering of the exclusive lists gets reversed, but this shouldn't
  affect anything.
  *****/
void make_copy(int time)
{
  vertex_t u,v,w;
  edgelist_t e;

  /* first do the ops and connect to their preconditions */
  get_next(op_table[time-1],0);
  while((v = get_next(op_table[time-1],1)) != NULL) {
    w = insert_into_table(op_table[time],v->name);
    w->prev_time = v;
    v->next_time = w;
    w->uid_mask = v->uid_mask;
    w->uid_block = v->uid_block;
    w->is_noop = v->is_noop;

    /* in-edges of op */
    for(e = v->in_edges; e; e = e->next) {
      u = e->endpt->next_time;  /* the fact for OUR in-edge */
      w->in_edges = insert_edge_at_end(w->in_edges, u);
      u->out_edges = insert_edge_at_end(u->out_edges, w);
    }
  }
```

```
  /* now do facts and connect them up.  Keep order the same */
  get_next(fact_table[time],0);
  while((v = get_next(fact_table[time],1)) != NULL) {
    w = insert_into_table(fact_table[time+1],v->name);
    w->prev_time = v;
    v->next_time = w;
    w->rand1 = random();
    w->uid_mask = v->uid_mask;
    w->uid_block = v->uid_block;

    /* in-edges */
    for(e = v->in_edges; e; e = e->next) {
      u = e->endpt->next_time;  /* the op for OUR in-edge */
      w->in_edges = insert_edge_at_end(w->in_edges, u);
      u->out_edges = insert_edge_at_end(u->out_edges, w);
    }
  }

  /* now do del_edges, exclusives (don't really need del_list) */
  get_next(fact_table[time+1],0);
  while((v = get_next(fact_table[time+1],1)) != NULL) {
    for(e = v->prev_time->exclusive; e; e = e->next) {
      v->exclusive = insert_edge(v->exclusive,e->endpt->next_time);
    }
    for(e = v->prev_time->excl_in_this_step; e; e = e->next) {
      v->excl_in_this_step =
      insert_edge(v->excl_in_this_step,e->endpt->next_time);
    }
    for(e = v->prev_time->del_edges; e; e = e->next) {
      v->del_edges = insert_edge(v->del_edges,e->endpt->next_time);
    }
  }
  get_next(op_table[time],0);
  while((v = get_next(op_table[time],1)) != NULL) {
    for(e = v->prev_time->exclusive; e; e = e->next) {
      v->exclusive = insert_edge(v->exclusive,e->endpt->next_time);
    }
    for(e = v->prev_time->del_edges; e; e = e->next) {
      v->del_edges = insert_edge(v->del_edges,e->endpt->next_time);
    }
  }
}


/* for auto case.  See if can stop expanding graph.  Can stop if (A)
 * all goals are reachable, and (B) they're not mutex of each other.
 */
int can_stop(int time)
{
  static int flag = 0;
  fact_list temp;
  char str[100];
  static vertex_t *v;   /* doing it this way since MAXGOALS not a constant */
  int num=0, i, j;
  if (flag == 0) { /* set up v */
    v = (vertex_t *) malloc(MAXGOALS*sizeof(vertex_t));
    flag = 1;
  }

  for(temp = the_goals; temp; temp = temp->next) {
    instantiate_into_string(temp->item, NULL, str,1);
    if ((v[num++] = lookup_from_table(fact_table[time],str)) == NULL)
      return 0;
  }
  /* reached all goals, but check to make sure none are exclusive. */
```

```
  for(i=0; i < num; ++i)
    for(j=i+1; j < num; ++j)
      if (are_mutex(v[i],v[j])) {
      fprintf(stderr,"Goals reachable at %d steps but mutually exclusive.\n",
              time);
      return 0;
      }
  return 1;
}


/***********LOAD IN OPERATOR AND FACT FILES*****************************/
/*******fn prototypes needed for load_ops********/
op_list load_ops_recursive(FILE *fp);
int find_catalysts(op_ptr op, int flag);
void set_insts(op_ptr op);
/***********************************************/


/* reads in operators into a big list and returns it. A hash mark "#"
   means old-style.  That means that preconditions are assumed to be
   deleted unless they are in the add-effects.  A precondition
   that wasn't deleted was called a "catalyst" in the old version.
*/
op_list load_ops(FILE *fp)
{
  int c;
  /* start with # if old style */
  if ((c = getc(fp)) == '#') oldstyle = 1;
  else oldstyle = 0;
  ungetc(c, fp);
  return load_ops_recursive(fp);
}

/******* Coisas do .LISP ******/
/* extern   FILE * yyin;
int yyparse();
int lineno;
op_list parser_return;
op_list current_op; */


/******Eliminacao da funcao que utiliza .LISP ******
op_list load_prodigy_ops(FILE *fp)
{
  op_list op;
  yyin = fp;
  yyparse();
  // /* now go down ops
  for(op = parser_return; op; op = op->next) {
   //  /* set instantiation list
    set_insts(op);
  }
  return parser_return;
} */


op_list load_ops_recursive(FILE *fp)
{
  op_list op; char str[100];
  int result;
  /* first read up to left paren */
  do {
    result = read_item(fp,junk);
  } while ((result != LEFT_PAREN) && (result != EOF));
```

```
  if (result == EOF) return NULL;

  read_item(fp,junk); /* this should be OPR */

  if (read_item(fp,str) != OK) do_error("error reading operator name");
  printf("%s\n",str);
  op = (op_list) malloc(sizeof(op_s));
  op->name = (char *) calloc(strlen(str) + 1, sizeof(char));
  strcpy(op->name, str);


  /* read in parameters */
  if (read_item(fp,junk) != LEFT_PAREN)
    do_error("expecting left paren before parameters");
  read_item(fp,junk);  /* PARAMS */
  load_fact_list(fp, &(op->params));


  /* read in preconditions. */
  if (read_item(fp,junk) != LEFT_PAREN)
    do_error("expecting left paren before preconds");
  read_item(fp,junk);  /* PRECOND */
  load_fact_list(fp, &(op->preconds));

   /* Entrada de equacoes algebricas entre variaveis e numeros */
   /* read in nologic preconds. */
  if (read_item(fp,junk) != LEFT_PAREN)
    do_error("expecting left paren before nologic preconds");
  read_item(fp,junk);  /* NOLOGIC */
  load_fact_list(fp, &(op->nologic));


  /* read in effects */
  if (read_item(fp,junk) != LEFT_PAREN)
    do_error("expecting left paren before preconds");
  read_item(fp,junk);  /* EFFECT */
  load_fact_list(fp, &(op->effects));
  read_item(fp,junk);  /* the final right parenthesis */


  /* Because the input is "old-style", need to put in deletes for the
     preconditions deleted and remove effects for the catalysts */
  if (oldstyle) find_catalysts(op,0);
  else if (find_catalysts(op,1) == 1) { /** just to verify **/
    printf("I notice that %s is adding one of its preconditions.\n",
          op->name);
    printf("This means you are probably using the old-style format\n");
    printf("If so, just put a hash mark '#' as the first character\n");
    printf("in your operator file and I will read it in correctly.\n");
    printf("If not, then I don't know what it means to add your ");
    printf("precondition.\n\n");
    exit(0);
  }
  /* set instantiation list */
  set_insts(op);
  op->next = load_ops_recursive(fp);
  return op;
}

/* check down two lists. If precond is in the effect list, then remove it.
 * If precond is not, then put in a new delete-effect.
 * This is done in a really slow way, but it's only done once per operator...
 * This will go away once we change the input format
 */
effect_list remove_catal_effects(token_list p, effect_list e)
{
```

```
  if (e == NULL) return NULL;
  else if (equal_facts(e->item, p))
    return remove_catal_effects(p,e->next); /* should free storage...*/
  else {
    e->next = remove_catal_effects(p, e->next);
    return e;
  }
}
/* if flag=1, then just return 1 if a catalyst exists and 0 otherwise,
 * but don't do any work.
 */
int find_catalysts(op_ptr op, int flag)
{
  precond_list p;
  effect_list e, new_e;
  token_list t;

  /* first, put in the deletes */
  for(p = op->preconds; p; p = p->next) {
    for(e = op->effects; e; e = e->next) {
      if (equal_facts(p->item, e->item)) break;
    }
    if (flag == 1) {
      if (e != NULL) return 1; /* found catalyst */
      else continue;
    }
    if (e == NULL) { /* didn't do break, so precondition is deleted */
      new_e = (effect_list) calloc(1, sizeof(fact_list_elt));
      t = (token_list) calloc(1, sizeof(token_list_elt));
      t->item = (char *) calloc(strlen(DELETE)+1,sizeof(char));
      strcpy(t->item, DELETE);
      t->next = p->item;  /* ok for them to point to same storage */
      new_e->item = t;
      new_e->next = op->effects;
      op->effects = new_e;
    }
  }
  if (flag == 1) return 0; /* found no catalysts */
  /* Now, get rid of the copy's */
  for(p = op->preconds; p; p = p->next)
    op->effects = remove_catal_effects(p->item, op->effects);
  return 0;
}

int really_is_var(char *str)
{
  return (str[strlen(str)-1] == '>');
}

/****have instantiation list associated to op.  Set all the var-parts.*****/
/** Changing this to allow the parameters to have some variables mentioned
 ** more than once.  Just makes a list of all of the different ones.
 ** This goes in reverse order. (Gets re-reversed when op-string is created)
 **/
void set_insts(op_ptr op)
{
  param_list pptr;
  token_list tptr;
  token_list tipos;
  instantiation_list iptr;
  op->insts = NULL;
  for(pptr = op->params; pptr != NULL; pptr = pptr->next) {
    for(tptr = pptr->item; tptr; tptr = tptr->next) {
      if (!is_var(tptr->item)) continue;
      if (!really_is_var(tptr->item)) {
      sprintf(junk,"Badly-formed op: %s",op->name);
```

```
        do_error(junk);
        }
        /* it's a variable.  Check to make sure it's a new one */
        for(iptr = op->insts; iptr; iptr = iptr->next) {
        if (strcmp(iptr->var_part, tptr->item) == SAME) break;
        }
        if (iptr != NULL) continue;  /* did break above: not new */

        tipos = tptr->next;
        op->insts = insert_inst(tptr->item, NULL, tipos->item, op->insts);
      }
    }
}

token_list load_tokens(FILE *fp);

/* returns number of facts loaded */
int load_fact_list(FILE *fp, fact_list *fptr)
{
  int result;
  token_list tlist;
  /* if not left paren, then we're done. */
  result = read_item(fp, junk);
  if (result == RIGHT_PAREN) {
    *fptr = NULL;
    return 0;
  }
   *fptr = (fact_list) malloc(sizeof(fact_list_elt));

   / * Registrando se os dados a seguir, tokens desta fact_list sao logicos ou nao
logicos */
   if (result == LEFT_PAREN){
      (*fptr)->det = 0;
     /* printf("PAREN %d\n", (*fptr)->det);  apagar !!!! */
    }
    if (result == LEFT_KEY){
      (*fptr)->det = 1;
     /* printf("KEY %d\n", (*fptr)->det);   apagar !!!! */
   }
   separa = (*fptr)->det; /* guarda se eh parentese - logico ou chave - nao logico
*/

   if ((tlist = load_tokens(fp)) == NULL) do_error("bad input file");

  (*fptr)->item = tlist;
  return 1 + load_fact_list(fp, &( (*fptr)->next ));
}


/* special for loading in types: ugly.  Should change format.
   Reads up to and including next left paren.  Also, sets up
   types_table */
fact_list load_types(FILE *fp)
{
  char temp_token[100];
  int result1;
  int temp_det;
  token_list t; fact_list f;
  pilha ptail;

for ( (result1= read_item(fp, junk)) ; (result1!= LEFT_PAREN && result1!=LEFT_KEY)
; (result1=read_item (fp, junk)) );

/* gravar a informacao de informacao logica ou nao para no final definir se o
chamado foi correto */
```

```
    if(result1 == LEFT_KEY) { temp_det = 1;}

    if(result1 == LEFT_PAREN) {temp_det = 0;}


    read_item(fp, temp_token); /* first in list: see if it is "preconds" */
    if (strcmp(temp_token, PRECOND) == SAME) return NULL; /* at end */

    /***************** Criar pilhas - nome *************************/
    / * Caso trata-se de uma variavel nao logica */
    if (temp_det == 1) {

      /* Primeiro token apos a chave de esquerda - trata-se do nome da pilha */
      /* Armazenando a primeira pilha encontrada - cabeca da lista ligada */
      if (numero_pilhas == 0){
        p = (pilha) malloc(sizeof(pilha_nl));
        p->name = (char *) calloc(1+strlen(temp_token),sizeof(char));
        strcpy(p->name, temp_token);
        p->next = NULL;
        printf("- 1 no logic data created as < %s >\n", p->name);
      }
      / * insrindo proximas pilhas */
      else {

        ptail = (pilha) malloc(sizeof(pilha_nl));
        ptail->name = (char *) calloc(1+strlen(temp_token),sizeof(char));
        strcpy(ptail->name, temp_token);
        ptail->next = p->next;
        p->next = ptail;
        printf("- 1 no logic data created as < %s >\n", ptail->name);

      }
      numero_pilhas = numero_pilhas + 1;
    }

    /***************** Armazenar na lista *************************/
    t = (token_list) malloc(sizeof(token_list_elt));
    t->item = (char *) calloc(1+strlen(temp_token),sizeof(char));
    strcpy(t->item, temp_token);

    t->next = load_tokens(fp);
    insert_token_list(types_table, t);
    f = (fact_list) malloc(sizeof(fact_list_elt));
    f->item = t;
    f->det= temp_det;
    /*printf("%d\n", f->det); apagar!!!!!!! */
    f->next = load_types(fp);
    return f;
}

/* Prototipo load_tokens3 e load_tokens2 */
token_list load_tokens3(FILE *fp, int *cap);
token_list load_tokens2(FILE *fp, int choose);

/* if next word is a right paren or right key, returns NULL. Otherwise, read in
tokens up
   to right parenthesis and return the list of tokens.  */
token_list load_tokens(FILE *fp)
{
  char temp_token[100];
  int result;
  int choose;
  token_list_elt *current;
  result = read_item(fp, temp_token);
  if ( result == RIGHT_PAREN || result == RIGHT_KEY ) return NULL;
```

```
  /* Se nao logico (separa == 1) e a informacao a seguir eh a capacidade da pilha
*/
  /* {capacity Building 10) */
  if ((strcmp(temp_token, CAPACITY) == SAME) && (separa == 1)) {
  /* printf("achado capacity  < %s >\n", temp_token);  apagar !!!! */
       current = (token_list) malloc(sizeof(token_list_elt));
       current->item = (char *) calloc(1 + strlen(temp_token),sizeof(char));
       strcpy(current->item, temp_token);
       choose = 1;     /* indica que devemos preencher o campo capacity da pilha */
       current->next = load_tokens2(fp, choose);
       return current;
  }
  else{
    /* {with Building 5) */
    if ((strcmp(temp_token, WITH) == SAME) && (separa == 1)) {
     /* printf("achado with < %s >\n", temp_token);   apagar !!!! */
       current = (token_list) malloc(sizeof(token_list_elt));
       current->item = (char *) calloc(1 + strlen(temp_token),sizeof(char));
       strcpy(current->item, temp_token);
       choose = 2;     /* indica que devemos preencher o campo counter da pilha */
       current->next = load_tokens2(fp, choose);
       return current;
     }

     /* Se eh dado logico - separa == 0 */
     else{
       current = (token_list) malloc(sizeof(token_list_elt));
       current->item = (char *) calloc(1 + strlen(temp_token),sizeof(char));
       strcpy(current->item, temp_token);
       current->next = load_tokens(fp);
       return current;
     } /*do segundo else */
   }
}

/* identifica qual pilha sera caracterizada */
token_list load_tokens2(FILE *fp, int choose)
{
  char temp_token[100];
  int result;
  int cap;
  token_list_elt *current;
  pilha g;

  result = read_item(fp, temp_token);
  if ( result == RIGHT_PAREN || result == RIGHT_KEY ) return NULL;
  /* Le o nome da pilha e procura a mesma para em load_tokens3, inserir o valor */

 /* Guardar em capacity - capacity */
 if (choose == 1) {
    /* printf("Inserted in CAPACITY of < %s >", temp_token); */
    current = (token_list) malloc(sizeof(token_list_elt));
    current->item = (char *) calloc(1 + strlen(temp_token),sizeof(char));
    strcpy(current->item, temp_token);
    current->next = load_tokens3(fp, &cap);

    g = (pilha) malloc(sizeof(pilha_nl));
    g->name = (char *) calloc(1+strlen(temp_token),sizeof(char));
    g = p;
    while ((g != NULL) && (strcmp(g->name, temp_token) != SAME))  g = g->next;
    g->capacity = cap;
    /* printf(" the number %d\n", g->capacity); */
    return current;
 }

 /* Guardar em counter - with */
```

```
 if (choose == 2) {
   /* printf("Inserted in COUNTER of < %s >", temp_token); */
   current = (token_list) malloc(sizeof(token_list_elt));
   current->item = (char *) calloc(1 + strlen(temp_token),sizeof(char));
   strcpy(current->item, temp_token);
   current->next = load_tokens3(fp, &cap);

   g = (pilha) malloc(sizeof(pilha_nl));
   g->name = (char *) calloc(1+strlen(temp_token),sizeof(char));
   g = p;
   while ((g != NULL) && (strcmp(g->name, temp_token) != SAME))  g = g->next;
   g->counter = cap;
   /* printf(" the number %d\n", g->counter); */
   return current;
 }

}

/* preenche os valores da pilha */
token_list load_tokens3(FILE *fp, int *cap )
{
  char temp_token[100];
  int result;
  token_list_elt *current;
  result = read_item(fp, temp_token);
  *cap = atoi(temp_token);
  if ( result == RIGHT_PAREN || result == RIGHT_KEY ) return NULL;
  current = (token_list) malloc(sizeof(token_list_elt));
  current->item = (char *) calloc(1 + strlen(temp_token),sizeof(char));
  strcpy(current->item, temp_token);
  current->next = load_tokens(fp);
  return current;

}

/*************************END OF STUFF FOR LOADING IN OPS,FACTS**********/
/************************************************************************/


/********* ROUTINES FOR TAKING THE OPS AND A LIST OF FACTS AND ***********
 *********      INSTANTIATING TO CREATE NODES IN THE GRAPH      ***********/

/* These routines create nodes in the graph by instantiating operators  */
/* This part is kindof messy, but conceptually it's not bad             */

/* see if match different var names to same constant */
int illegal_match(instantiation_list insts)
{
  instantiation_list i,j;
  for(i = insts; i; i = i->next)
    for(j = i->next; j; j = j->next)
      if (strcmp(i->var_part,j->var_part) != SAME &&
        strcmp(i->const_part,j->const_part) == SAME)
      return 1;
  return 0;
}

/*****************here's the important one for making the graph*************/
/* this takes in an operator and an instantiation list, and a time.
   Inserts the operators and new facts and connects them appropriately.

   Inserts op into op_table[time]. attaches edges to fact_table[i] and
   fact_table[i+1].

   If the op can be avoided due to preconds being mutually exclusive, then
   op is still created, but no effects made. So, will be cleaned up
```

```
  on next pass.
 */
void make_graph_piece(op_ptr op, int time)
{
  vertex_t op_vert, fact_vert, fv2;
  effect_list elist; precond_list plist, p2;
  char str[100], str2[100];
  string_list temp;

  /* get op name */
  make_op_string(op, str);
  /* NEW: need to make sure that vars of different names match to different
     constants.  Otherwise can get into trouble. */
  if (illegal_match(op->insts)) {
    if (DEBUG_FLAG > 1) printf("discarding %s.\n",str);
    return;
  }

  /* first, insert the operator and get a pointer to it */
  op_vert = insert_into_table(op_table[time], str);


  /* if op can be avoided then return now */
  for(plist = op->preconds; plist; plist = plist->next) {
    instantiate_into_string(plist->item, op->insts, str,1);
    fact_vert = lookup_from_table(fact_table[time], str);
    for(p2 = plist->next; p2; p2 = p2->next) {
      instantiate_into_string(p2->item, op->insts, str2,1);
      fv2 = lookup_from_table(fact_table[time], str2);
      if (fact_vert == NULL || fv2 == NULL) {      /* wasn't there */
      do_error("didn't find precondition. Shouldn't happen");
      }
      if (are_mutex(fact_vert, fv2)) { /* DONT NEED TO PUT OP IN */
      if (DEBUG_FLAG) printf("Avoiding %s, time %d\n",op_vert->name, time);
      return;
      }
    }
  }

  /* set up the connections to the preconditions */
  for(plist = op->preconds; plist; plist = plist->next) {
    instantiate_into_string(plist->item, op->insts, str,1);
    fact_vert = lookup_from_table(fact_table[time], str);
    op_vert->in_edges = insert_edge(op_vert->in_edges, fact_vert);
    fact_vert->out_edges = insert_edge(fact_vert->out_edges, op_vert);
  }

  /* set up the connections to effects. Now allowing deletes */
  for(elist = op->effects; elist; elist = elist->next) {
    if (strcmp(elist->item->item, DELETE) != SAME) {
      instantiate_into_string(elist->item, op->insts, str,1);
      fact_vert = lookup_from_table(fact_table[time+1], str);
      if (fact_vert == NULL) {                        /* wasn't there */
      if (do_irrel) {  /* in this case, check to see if it's useful first */
        if (lookup_from_table(useful,str) == NULL) {
          if (DEBUG_FLAG > 2)
            printf("%s not useful. Not putting into graph at time %d.\n",
                 str, time+1);
          continue;
        }
      }
      fact_vert = insert_into_table(fact_table[time+1],str);
      }
      op_vert->out_edges = insert_edge(op_vert->out_edges, fact_vert);
      fact_vert->in_edges = insert_edge(fact_vert->in_edges, op_vert);
    } else {
```

```
      /* just put into del list.  Don't try creating new node */
      instantiate_into_string(elist->item->next, op->insts, str,1);
      temp = (string_list) malloc(sizeof(token_list_elt));
      temp->item = (char *) calloc(strlen(str) + 1, sizeof(char));
      strcpy(temp->item, str);
      temp->next = op_vert->del_list;
      op_vert->del_list = temp;
    }
  }
}


/*** do one of the operators ***/
/* The do_operator_rec routine can be used to do forwards or backwards,
   but is written using variable names as if going in forwards direction.

   This routine takes the list of facts true at current time, a ptr to an
   operator, a list of preconditions to go, and some instantiations.
   It calls make_graph_piece for each new node that needs to be created.

   --> Also takes in hash table.  For speedup...

   When d_o_rec matches, it calls itself recursively with
   flag=1 and replacing current facts by the types list, to make sure that
   those get matched.  Should now handle correctly the case where there
   is a parameter that is not one of the preconditions.
*/
void do_operator_rec(hashtable_t, fact_list, op_ptr, precond_list, nologic_list,
int, int);

void do_operator(hashtable_t htable, fact_list current_facts, op_ptr op,
             precond_list p, nologic_list n, int time)
{
  do_operator_rec(htable, current_facts,op,p, n, time,0);
}

/* calls make_graph_piece unless "going backwards", in which case calls
   list_useful_preconds. */
void list_useful_preconds(op_ptr op, int time);
void selector(op_ptr op, int time)
{
  if (!going_backwards) make_graph_piece(op,time);
  else list_useful_preconds(op,time);
}

 /********** Estudando a expressao algebrica entre variaveis e numeros apresentada
         nos operadores para dados nao logicos ********************************/

int verifica_nologic_prec(nologic_list nologic_prec, op_ptr op, int flags) {

/* return 0: Nao se trata de uma verdade */
/* return 1: Expressao eh verdadeira */
/* "flags", CUIDADO!!! NAO "flag", indica se eh a primeira vez que se chama a
funcao
    verifica_nologic_prec */
char *pr_termo;
char *sg_termo;
char *tr_termo;
char *qr_termo;
char *simbolo;
char *sinal;
token_list h;
instantiation_list coisa;
int t1;
int t2;
int t3;
```

```
int oka = 0; /* Analisa qual o formato da expressao */
int recebeu = 0;

/* Se fim das expressoes */
/*if (nologic_prec->next == NULL){*/
h = (token_list) malloc(sizeof(token_list_elt));
/*h->item = (char *) calloc(1+strlen(temp_token),sizeof(char)); */
h->item = (char *) calloc(100,sizeof(char));

if ( flags == 1) {
 if (nologic_prec == NULL) return 1;
}

h = nologic_prec->item;
if ( flags == 0) h = h->next;
else { if (nologic_prec == NULL) return 1;
}

/************* Le os termos da equacao ******************/
if (h == NULL) return 1;
pr_termo = h->item;
/* printf("primeiro: %s\n", pr_termo);*/
h = h->next;
simbolo = h->item;
sinal = h->item;
/* printf("simbolo: %s\n", simbolo);*/
h = h->next;
sg_termo = h->item;
/* printf("segundo: %s\n", sg_termo);*/
/*}*/


if ( (strcmp(simbolo, "+") == SAME) || (strcmp(simbolo, "-") == SAME)) {
 h = h->next;
 tr_termo = h->item;
 sinal = h->item;
 /* printf("terceiro: %s\n", tr_termo);*/
 h = h->next;
 qr_termo = h->item;
 /* printf("quarto: %s\n", qr_termo);*/
 oka = 1;
}

/* Se a equacao tem o formato: "variavel" "simbolo" "variavel", como <var1> >
<var2>, oka == 0;
   Se a equacao tem o formato: <var1> <operador> <var2> <simbolo> <var3>, oka ==
1; */

/************ Traz os valores das variaveis ***************/
if(oka == 0){
 if(is_var(sg_termo)){
  for(coisa = op->insts; coisa!= NULL; coisa = coisa->next) {
    if (strcmp(coisa->var_part, pr_termo) == SAME) {
      t1 = atoi(coisa->const_part);
      /* printf("t1: %d\n", t1); */
     }
    if (strcmp(coisa->var_part, sg_termo) == SAME) {
      t2 = atoi(coisa->const_part);
      /* printf("t2: %d\n", t2); */
    }
  }
 }
 else {
  t2 = atoi(sg_termo);
  for(coisa = op->insts; coisa!= NULL; coisa = coisa->next) {
    if (strcmp(coisa->var_part, pr_termo) == SAME) {
```

```
      t1 = atoi(coisa->const_part);
    }
  }
 }
}

if(oka == 1) {
  if(is_var(qr_termo)){
    for(coisa = op->insts; coisa!= NULL; coisa = coisa->next) {
      if (strcmp(coisa->var_part, pr_termo) == SAME) {
        t1 = atoi(coisa->const_part);
        /*printf("t1: %d\n", t1);*/
      }
      if (strcmp(coisa->var_part, qr_termo) == SAME) {
        t2 = atoi(coisa->const_part);
        /*printf("t2: %d\n", t2);*/
      }
      if (strcmp(coisa->var_part, sg_termo) == SAME) {
        t3 = atoi(coisa->const_part);
        /*printf("t3: %d\n", t3); */
        recebeu = 1;
      }
    }
    if (recebeu == 0) t3 = atoi(sg_termo);
  }
  else {
    t2 = atoi(qr_termo);
    for(coisa = op->insts; coisa!= NULL; coisa = coisa->next) {
      if (strcmp(coisa->var_part, pr_termo) == SAME)
        t1 = atoi(coisa->const_part);
      if (strcmp(coisa->var_part, sg_termo) == SAME) {
        t3 = atoi(coisa->const_part);
      recebeu = 1;
      }
    }
    if (recebeu == 0) t3 = atoi(sg_termo);
  }
  if (strcmp(simbolo, "+") == SAME) {
    t1 = t1 + t3;
  }
  if (strcmp(simbolo, "-") == SAME) {
    t1 = t1 - t3;
  }
}

/********** Verifica se a expressao eh verdadeira *************/
if (strcmp(sinal, "=") == SAME) {
 if (t1 == t2) return (1*verifica_nologic_prec(nologic_prec->next, op, 1));
else return 0;
}

if (strcmp(sinal, ">") == SAME) {
 if (t1 > t2) return (1*verifica_nologic_prec(nologic_prec->next, op, 1));
else return 0;
}

if (strcmp(sinal, "<") == SAME) {
 if (t1 < t2) return (1*verifica_nologic_prec(nologic_prec->next, op, 1));
else return 0;
}

if (strcmp(sinal, "!=") == SAME) {
 if (t1 != t2) return (1*verifica_nologic_prec(nologic_prec->next, op, 1));
else return 0;
}
```

```
}


void do_operator_rec(hashtable_t htable, fact_list current_facts, op_ptr op,
                     precond_list precs_to_go, nologic_list nologic_prec, int
time, int flag)
{
  fact_list factptr;
  token_list prec;
  int result;
  int check;
  int i, old_matched[MAX_TOKENS]; /* 0 if const_part is NULL. Else 1 */
  instantiation_list iptr;
  char str[100];

  if (precs_to_go == NULL) {
    /* if flag is 1, that means we're done. Otherwise, call recursively
        * using the_types. */
    if (flag == 1){
      check = verifica_nologic_prec(nologic_prec, op, 0); /*checa as condições não
lógicas para vada operador */
     /*  printf("resultado: %d\n", check);*/
     /*  printf("nome do operador: %s\n", op->name);*/
      if (check == 1) {
      selector(op,time);}
      else return;
    }
    else do_operator_rec(types_table, the_types, op, op->params, nologic_prec,
time, 1);
    return;
  }

  for(i=0, iptr = op->insts; iptr!= NULL; iptr = iptr->next, i++) {
    if (iptr->const_part == NULL) old_matched[i]=0;
    else old_matched[i] = 1;
  }
  prec =  precs_to_go->item;

  /* first see if it's fully instantiated.  If so, it's lots faster */
  if (instantiate_into_string(prec, op->insts, str,0)) { /* YAY! */
    if (lookup_from_table(htable, str))           /* found it */
      do_operator_rec(htable, current_facts, op, precs_to_go->next, nologic_prec,
time,flag);
    return;
  }
  /* now we know that it is NOT fully instantiated */

  for(factptr = current_facts; factptr != NULL; factptr = factptr->next) {
    result = compare_and_instantiate(prec, factptr->item, op->insts);
    if (result == NO) continue;  /* No match.  Most common */
    if (result == YES) do_error("bug in do_operator_rec");
    do_operator_rec(htable, current_facts, op,  precs_to_go->next, nologic_prec,
time, flag);
    /* now, un-instantiate the new ones */
    for(i=0,iptr=op->insts; iptr!=NULL; iptr = iptr->next, i++)
      if (old_matched[i] == 0) iptr->const_part = NULL;
  }
}

/* this routine takes an uninstantiated pattern, a fact, and a list of
   partially made instantiations.
   Sees if the precondition matches the fact given the
   instantiations so far.  Returns YES if exact match NO if no exact match
   and NEW_INSTS if new instantiations were needed.
 */
int compare_and_instantiate(token_list patt, token_list fact,
```

```
                                instantiation_list insts)
{
  token_list p,f;          /* p,f,iptr are pointers to go down the lists */
  instantiation_list iptr;
  int temp_arr[MAX_TOKENS]; /* temp_arr[i] = NO or NEW_INSTS */
  int i=0,result = YES;  /* change result to NEW_INSTS when add instantiatons*/

  /* Check the constant parts.  If any fail to match,  we can return.  */
  for(p = patt, f = fact; p && f; p = p->next, f = f->next)
    if (is_const(p->item) && !equal_tokens(p->item,f->item)) return NO;

  /* check they're the same length */
  if (p || f) return NO;

  /* initialize temp_arr */
  for(i=0,iptr=insts; iptr !=NULL; i++,iptr=iptr->next) temp_arr[i] = NO;

  for(p = patt, f = fact; p && f; p = p->next, f = f->next) {
    if (is_const(p->item)) continue;     /* we already know this matches */
    for(i=0,iptr = insts; iptr != NULL; i++, iptr = iptr->next)
      if (equal_tokens(p->item,iptr->var_part)) {  /* matched */
        if (iptr->const_part == NULL) {           /* need new inst */
          iptr->const_part = f->item;             /* so, do it */
          temp_arr[i] = NEW_INSTS;                /* and remember it */
          result = NEW_INSTS;                     /* and change result */
        } else if (!equal_tokens(f->item,iptr->const_part)) { /* not equal*/
          result=NO;
      }
      break;
      }
    if (iptr == NULL) {
      sprintf(junk,"Badly formed operator: '%s' not a parameter",p->item);
      do_error(junk);
    }
    if (result == NO) break;
  }

  /* if result is NO, then reset insts */
  if (result == NO)
    for(i=0,iptr=insts; iptr !=NULL; i++,iptr=iptr->next) {
      if (temp_arr[i] == NEW_INSTS) iptr->const_part = NULL;
    }
  return result;
}

/*************************END OF OP->PART_OF_GRAPH section***************/



/***************TRY TO GET RID OF IRRELEVANT FEATURES, IF WE'RE ASKED TO***/
/* ----------------------------------------
   Get rid of irrelevant features by going backwards from goal to see
   all the possible things that might be useful.
   Should give this a parameter that says how long to spend before giving up,
   but for now, let's ignore that issue.

   Idea: start with hashtable containing all goals, marked as un-visited
   (used == 0) and at iteration 0 (is_true == 0).  Go through all goals
   at current iteration and for each one, mark as used and then for each op,
   see if the goal is an add-effect of the op. If so, put into the
   hashtable all the possible preconditions for the op (use the-types) that
   aren't already there, and set their is_true to the current time +1.

   Keep going until everything is used.
```

```
   Don't free up storage: keep around the table for use in keeping graph
   pruned.

 */

static int num_facts_searched = 0; /* how many looked at. For info. */

/** prototypes **/
fact_list delete_useless(fact_list f);
int main_search_loop(int time, op_list ops);
int const_parts_match(token_list patt, token_list fact);
void init_useful(void);
int add_fact(fact_list f);

fact_list useful_facts(op_list ops, fact_list f)
{
  int time = 0;
  going_backwards = 1;
  printf("beginning pre-preprocessing...\n");
  init_useful();  /* set up the hash table */
  while(main_search_loop(time++, ops))
    {printf("time %d\n",time);} /* if (time == 2) exit(0);}  */
  printf("found %d possibly useful facts.\n",num_facts_searched);
  going_backwards = 0;
  f = delete_useless(f);
  printf("...done pre-preprocessing.\n");
  return f;
}

fact_list delete_useless(fact_list f)
{
  char str[100];
  if (f == NULL) return f;
  instantiate_into_string(f->item, NULL, str,1);
  if (lookup_from_table(useful, str)) {
    f->next = delete_useless(f->next);
  } else {
    printf("Noticed fact %s not useful.\n",str);
    f = delete_useless(f->next);  /* should really free storage here...*/
  }
  return f;
}

void list_useful_preconds(op_ptr op, int time)
{
  precond_list plist;
  char str[100];
  vertex_t v;
  if (illegal_match(op->insts)) return;  /* not a legal match of stuff */
  for(plist = op->preconds; plist; plist = plist->next) {
    instantiate_into_string(plist->item, op->insts, str,1);
    if (lookup_from_table(useful, str)) continue; /* don't insert */
    v = insert_into_table(useful, str);
/*    printf("%s\n",str); */
    v->is_true = time;
  }
}

/* returns 0 if don't need to continue. Returns 1 if did something. */
int main_search_loop(int time, op_list ops)
{
  vertex_t v;
  op_ptr op;
  effect_list e;
  int return_val = 0;
  fact_list_elt effect, goal;
```

```
  effect.next = goal.next = NULL;

  get_next(useful,0);
  while((v = get_next(useful,1)) != NULL) {
    if (v->is_true != time || v->used) continue;
    v->used = 1;  /* mark as used */
    ++num_facts_searched;
    goal.item = token_list_from_string(v->name);/* put into token list format*/

    for(op = ops; op; op = op->next) {
      for(e = op->effects; e; e = e->next) {
      if(!add_fact(e)) continue;  /* only want to look at adds */
            /* for speed, first check const parts */
      if(!const_parts_match(e->item,goal.item)) continue;
      effect.item = e->item;  /* set up one-element list */
      return_val = 1;
      do_operator(useful, &goal, op, &effect, op->nologic, time+1);
/*    printf("%d. Did %s for goal %s\n",num_facts_searched,op->name, v->name); */
      }
    }
  }
  return return_val;
}

/* given an effect(patt) and a goal(fact). See if constant parts match */
int const_parts_match(token_list patt, token_list fact)
{
  token_list p,f;
  for(p = patt, f = fact; p && f; p = p->next, f = f->next)
    if (is_const(p->item) && !equal_tokens(p->item,f->item)) return 0;
  if (p || f) return 0;  /* different lengths */
  return 1;
}

void init_useful(void)
{
  fact_list temp;
  for(temp = the_goals; temp; temp = temp->next)
    insert_token_list(useful, temp->item);
}


/* is the first fact an effect that's added? */
int add_fact(fact_list f)
{
  if (strcmp(f->item->item, DELETE) == SAME) return 0;
  else return 1;
}
```

utilitiesMod.c

```
/********************** Graphplan ************************************
  (C) Copyright 1995 Avrim Blum and Merrick Furst
      Modified by Rodrigo Chen and Tiago Vaquero

  All rights reserved. Use of this software is permitted for non-commercial
  research purposes, and it may be copied only for that use.  All copies must
  include this copyright message.  This software is made available AS IS, and
  neither the authors nor CMU make any warranty about the software or its
  performance.
*************************************************************************/


#include "graphplanMod.h"
#ifndef FALSE
#define FALSE 0
#endif
extern hashtable_t *fact_table, *op_table;  /* the arrays of hash tables */
extern int hash_hits, hash_inserts, hash_numctr, hash_ctr;
extern int do_noexcl;
char templine[100]; /* global var that can use to store lines read in */

fact_list fact_list_from_vertex_list(vertex_t vlist);


/***given a string, makes a new one with NOOP in front***
  Returns new storage.
*/
char *make_noop_string(char *str)
{
  char *newp;
  newp = (char *) calloc(strlen(str) + strlen(NOOP) + 2, sizeof(char));
  sprintf(newp,"%s_%s",NOOP,str);
  return newp;
}

/*set uid_mask and uid_block*/
void set_uid(vertex_t v, int id)
{
  v->uid_block = id/32;
  v->uid_mask = 1 << (id % 32);
}


edgelist_t insert_edge(edgelist_t e, vertex_t v)
{
  edgelist_t newedge = (edgelist_t) calloc(1,sizeof(edgelist_s));
  newedge->next = e;
  newedge->endpt = v;
  return newedge;
}

/* like above, but insert at end of list */
edgelist_t insert_edge_at_end(edgelist_t e, vertex_t v)
{
  edgelist_t newedge;
  if (e == NULL) {
    newedge = (edgelist_t) calloc(1,sizeof(edgelist_s));
    newedge->endpt = v;
    return newedge;
  } else {
    e->next = insert_edge_at_end(e->next, v);
    return e;
  }
}
```

```
/* makes a fact list from the hash table */
fact_list make_list_from_htable(hashtable_t h)
{
  int i;
  fact_list the_facts = NULL,current, p;
  for(i=0; i < HSIZE; ++i)
    if (h[i] != NULL) {
      current = fact_list_from_vertex_list(h[i]);
      for(p = current;  p->next; p = p->next);
      p->next = the_facts;
      the_facts = current;
    }
  return( the_facts );
}

void completely_free_fact_list(fact_list l)
{
  fact_list temp;
  while(l) { temp = l->next; free_token_list(l->item); free(l); l = temp;}
}

void free_token_list(token_list l)
{
  token_list temp;
  while(l) {   temp = l->next; free(l->item); free(l); l = temp;  }
}


/* makes string list into fact list. */
fact_list fact_list_from_vertex_list(vertex_t vlist)
{
  fact_list retval = NULL, f;
  char *str;
  for(; vlist; vlist = vlist->next) {
    str = vlist->name;
    f = (fact_list) malloc(sizeof(fact_list_elt));
    f->next = retval;
    retval = f;
    f->item = token_list_from_string(str);
  }
  return retval;
}

token_list token_list_from_string(char str[])
{
  char *p = str, *q, temp[50];
  token_list_elt dummy, *current;
  current = &dummy;
  while (*p) {
    current->next = (token_list) malloc(sizeof(token_list_elt));
    current = current->next;
    for(q = temp; (*p != '\0') && (*p != CONNECTOR); *q++ = *p++);
    if (*p == CONNECTOR) ++p;
    *q = '\0';
    current->item = (char *) calloc(1 + strlen(temp),sizeof(char));
    strcpy(current->item, temp);
  }
  current->next = NULL;
  return( dummy.next );
}


/* insert into instantiation list */
instantiation_list insert_inst(char *v, char *c, char *t, instantiation_list i)
{
```

```
    instantiation_list temp=(instantiation_list) malloc(sizeof(instantiation_s));
    temp->var_part = v;
    /* printf("%s\n", v); */
    temp->const_part = c;
    /* printf("%s\n", c); */
    temp->type_name = t;
    /* printf("%s\n", temp->type_name); */
    temp->next = i;
    return( temp );
}


/* Returns instantiate token list in a string (last argument) Uses
   CONNECTOR to connect the tokens together.
   if "failflag" is 0 then Return 1 if ok, 0 if failure.
   Otherwise, exit on failure.
   */
int instantiate_into_string(token_list l, instantiation_list inst,char str[],
                        int failflag)
{
  instantiation_list i;
  char *p;
  token_list t = l;

  for(p = str; t; t = t->next) {
    if (is_const(t->item))  /* just copy over */
      { strcpy(p,t->item); p+=strlen(t->item); *p++ = CONNECTOR; }
    else {
      for(i=inst; i; i = i->next)
        if (strcmp(i->var_part, t->item) == SAME) {
        if (i->const_part == NULL) {i = NULL; break;}
          strcpy(p,i->const_part);p+=strlen(i->const_part);*p++ = CONNECTOR;
          break;
        }
      if (!i) {
      if (failflag) do_error("instantiation failed");
      return 0;
      }
    }
  }
  *(--p) = '\0';
  return 1;
}

/* "whitespace" is blank or paren */
int is_stopper(int c)
{
  return (isspace(c) || (c == ')') || (c == '('));
}

/* read the next "item".  An item is either a string of non-parentesis,
   non-blank characters, or a parenthesis.  In the former case, put into
   str and return OK. In the latter case, return LEFT_PAREN or RIGHT_PAREN
   appropriately.  Return EOF on end-of-file.

   if character is CONNECTOR, change it to REPLACEMENT
 */
#define REPLACEMENT '.'
int read_item(FILE *fp, char *str)
{
  int letter;
  /* first read in any blankspace and check for EOF */
  while(isspace(letter = getc(fp)));
  if (letter == EOF) return EOF;

  /* check if parenthesis */
```

```
    if (letter == '(') return LEFT_PAREN;
    if (letter == ')') return RIGHT_PAREN;


    /* checa as chaves - informacao nao logica */
    if (letter == '{') return LEFT_KEY;
    if (letter == '}') return RIGHT_KEY;

    ungetc((char) letter, fp);
    while(!is_stopper(*str++ = getc(fp)))     /* read word in */
      if (*(str-1) == CONNECTOR) *(str-1) = REPLACEMENT;  /* change '_'to'.'*/
    ungetc(*--str, fp);                       /* went one too far */
    *str = '\0';                              /* end it nicely */
    return OK;
  }

  /* string to describe an instantiated operator: re-reverse order */
  void rec_make_op_string(op_ptr op,instantiation_list insts, char *str)
  {
    if (insts == NULL) return;
    rec_make_op_string(op, insts->next, str);
    if (insts->const_part == NULL)
      {fprintf(stderr,"op %s, ",op->name); do_error("bad instantiation");}
    sprintf(str + strlen(str), "_%s", insts->const_part);
  }
  /* string to describe an instantiated operator */
  void make_op_string(op_ptr op, char *str)
  {
    sprintf(str, "%s",op->name);
    rec_make_op_string(op, op->insts, str + strlen(str));
  }

  void do_error(char *s)
  {
    fprintf(stderr,"%s\n",s); exit( 1 );
  }

  int equal_facts(token_list f1, token_list f2)
  {
    for(; f1 && f2; f1 = f1->next, f2 = f2->next)
      if (!equal_tokens(f1->item,f2->item)) return 0;  /* different tokens*/
    if (f1 || f2) return 0;  /* different lengths */
    return 1;
  }


  /************ ROUTINES FOR DOING MUTUAL EXCLUSION RELATIONS ************
   ************ --SHOULD PROBABLY BE IN A DIFFERENT FILE      ************/

  /* Note: the first fact layer doesn't have uids set, but then again, they
     aren't exlusive either */
  int are_mutex(vertex_t v1, vertex_t v2)
  {
    return ((v1->exclusive_vect[v2->uid_block]) & (v2->uid_mask));
  }

  int are_mutex_in_this_step(vertex_t v1, vertex_t v2)
  {
    return ((v1->excl_in_this_step_vect[v2->uid_block]) & (v2->uid_mask));
  }


  /* Given time in op hash table, finds all mutually exclusive ops.
   */
  void find_all_mutex_ops(hashtable_t harr[], int time)
  {
```

```
  vertex_t v;
  get_next(harr[time],0); /* initialize */
  while((v = get_next(harr[time],1)) != NULL)
    find_mutex_ops(v, time);
  return;
}

/* make exclusive if they aren't already */
void do_exclusive(vertex_t v, vertex_t v2)
{
  if (are_mutex(v,v2)) return;
  v->exclusive_vect[v2->uid_block] |= v2->uid_mask;
  v2->exclusive_vect[v->uid_block] |= v->uid_mask;
  v->exclusive = insert_edge(v->exclusive, v2);
  v2->exclusive = insert_edge(v2->exclusive, v);
}


/* Given a vertex representing an operator, it finds other mutually-exclusive
 * ops.
 *
 * The method is: (1) if a precond p is mutually exclusive with another fact
 * q s.t. q is a precond for another op then the ops are also mutually
 * exclusive.
 *
 * LOCAL PART: If the operator deletes something that is either
 * someone else's precondition or someone else's effect, then the two ops
 * are exclusive.  THE REASON FOR THIS is (I) if you delete a precond then
 * impossible to do both, and (II) if you delete an effect, then the order of
 * these ops will matter, though this only is a problem in domains where an op
 * might add a fact that was already there (otherwise these two can't possibly
 * happen at the same time anyway).
 *
 * NEW: Make noop exclusive of any other action which adds its effect. The
 * Reason is that you need never include both in a plan.
 *
 * ALSO: put into del_edges.
 *
 * For speedup: for type(1) only call do_exclusive for vertices v2
 * for which v<v2 as ptrs
 */
void find_mutex_ops(vertex_t v, int time)
{
  string_list slist;
  edgelist_t temp1, temp2, temp3;
  vertex_t v2, nextfact, prevfact;

  /* Do type (1) */
  for(temp1 = v->in_edges; temp1; temp1 = temp1->next) {
    for(temp2 = temp1->endpt->exclusive; temp2; temp2 = temp2->next) {
      for(temp3 = temp2->endpt->out_edges; temp3; temp3 = temp3->next) {
      v2 = temp3->endpt;  /* this is the op to check */
      if ((int) v < (int) v2) do_exclusive(v, v2);    /* Do it */
      }
    }
  }

  /* LOCAL PART: */
  for(slist = v->del_list; slist; slist = slist->next) {
    nextfact = lookup_from_table(fact_table[time+1], slist->item);
    if (nextfact == NULL) {
      if (DEBUG_FLAG > 1) printf("%s deleting nonexistent fact\n",v->name);
      continue;
    }
    prevfact = nextfact->prev_time;
    /* do preconditions */
```

```
    if (prevfact) {
      for(temp2 = prevfact->out_edges; temp2; temp2 = temp2->next) {
      if (v != temp2->endpt) {
        do_exclusive(v, temp2->endpt);
      }
      }
    }
    /* now effects */
    for(temp2 = nextfact->in_edges; temp2; temp2 = temp2->next) {
      if (v != temp2->endpt) {
      do_exclusive(v, temp2->endpt);
      }
    }
    /* now put into del_edges (and put info into FACT too, for DELETES)*/
    v->del_edges = insert_edge(v->del_edges, nextfact);
    nextfact->del_edges = insert_edge(nextfact->del_edges,v);
  }
  /* free storage of delete list */
  while(v->del_list) {
    slist = v->del_list;
    v->del_list = v->del_list->next;
    free(slist);
  }
  /* SPECIAL for NOOPs */
  if (v->is_noop) {
    for(temp2 = v->out_edges->endpt->in_edges; temp2; temp2 = temp2->next) {
      if (v != temp2->endpt)
      do_exclusive(v, temp2->endpt);
    }
  }
}

/* Given two vertices representing facts, are they mutually exclusive?
 * just see if all ops generating p are exclusive of all ops generating q.
 */
int are_facts_exclusive(vertex_t p, vertex_t q)
{
  edgelist_t temp1, temp2;
  if (do_noexcl) return 0; /* option to turn off exclusivity */
  for(temp1 = p->in_edges; temp1; temp1 = temp1->next) {
    for(temp2 = q->in_edges; temp2; temp2 = temp2->next) {
      if (!are_mutex(temp1->endpt, temp2->endpt)) return 0;
    }
  }
  if (DEBUG_FLAG>2) printf("Facts %s and %s mutually exclusive\n",
                    p->name, q->name);
  return 1;
}



/* Given time in fact hash table, finds all mutually exclusive facts.
 * Assumes there are no more than MAXNODES facts at this time step.
 * Returns number of facts and number of exclusive pairs.
 *
 * Note: varr is used in find_currently_mutex_facts too.
 *
 */
static vertex_t *varr;
pair find_mutex_facts(hashtable_t harr[], int time)
{
  static int flag = 0;
  pair retval;
  edgelist_t e;
  vertex_t v;
  int i, j, fcount=0, ecount=0;
```

```
  if (flag == 0) {  /* doing this since MAXNODES is not a constant */
    varr = (vertex_t *) malloc(MAXNODES*sizeof(vertex_t));
    flag = 1;
  }
  get_next(harr[time],0); /* initialize */
  for(i=0; (varr[i] = get_next(harr[time],1)) != NULL; ++i);
  for(i=0; varr[i] != NULL; ++i) {
    ++fcount;
    /* if it's NEW, then check against everybody */
    if (!varr[i]->prev_time) {
      for(j=0; varr[j] != NULL; ++j) {
      if (!varr[j]->prev_time && j <= i) continue; /* already checked */
      if (are_facts_exclusive(varr[i], varr[j])) {
        ++ecount;
        do_exclusive(varr[i], varr[j]);
      }
      }
    } else { /* OLD, so only check previous exclusives */
      for(e = varr[i]->prev_time->exclusive; e; e = e->next) {
      v = e->endpt->next_time;
      if ((int) v < (int) varr[i]) continue;  /* get in other direction */
      if (are_facts_exclusive(varr[i], v)) {
        ++ecount;
        do_exclusive(varr[i], v);
      }
      }
    }
  }
  retval.first = fcount;
  retval.second = ecount;
  return retval;
}

/* This routine looks at a pairs of facts at a given time step such that
   neither fact is in the initial state, and returns 1 if they cannot both
   be created by ops in this time step.

   Note: be sure to skip over the noops here.
*/
int are_facts_excl_in_this_step(vertex_t v1, vertex_t v2)
{
  edgelist_t e1, e2;
  for(e1 = v1->in_edges; e1; e1 = e1->next) {
    if (IS_NOOP(e1->endpt)) continue;
    for(e2 = v2->in_edges; e2; e2 = e2->next) {
      if (IS_NOOP(e2->endpt)) continue;
      if (!are_mutex(e1->endpt, e2->endpt)) return 0;
    }
  }
  return 1;
}

/* make them excl in this step */
void do_excl_in_this_step(vertex_t v1, vertex_t v2)
{
  v1->excl_in_this_step_vect[v2->uid_block] |= v2->uid_mask;
  v2->excl_in_this_step_vect[v1->uid_block] |= v1->uid_mask;
  v1->excl_in_this_step = insert_edge(v1->excl_in_this_step, v2);
  v2->excl_in_this_step = insert_edge(v2->excl_in_this_step, v1);
}

/* This routine looks at all pairs of facts a given time step and for each
   pair such that neither fact is in the initial state marks the two facts as
   "excl_in_this_step" if they cannot both be created by ops in this time step.

   Using same storage as find_mutex_facts, so CALL THIS ONE AFTER
```

```
   CALLING find_mutex_facts.
 */
void find_currently_mutex_facts()
{
  edgelist_t e;
  vertex_t v2;
  int i, j;

  /* initialization of varr done already in find_mutex_facts */

  for(i=0; varr[i] != NULL; ++i) {
    if (lookup_from_table(fact_table[0],varr[i]->name)) continue;

    /* if it's NEW, then check against everybody */
    if (!varr[i]->prev_time) {
      for(j=0; varr[j] != NULL; ++j) {
      if (!varr[j]->prev_time && j <= i) continue; /* already checked */
      if (lookup_from_table(fact_table[0],varr[j]->name)) continue;
      if (are_facts_excl_in_this_step(varr[i], varr[j]))
        do_excl_in_this_step(varr[i], varr[j]);
      }
    } else {                /* OLD, so only check previous exclusives */
      for(e = varr[i]->prev_time->excl_in_this_step; e; e = e->next) {
      v2 = e->endpt->next_time;
      if ((int) v2 < (int) varr[i]) continue;  /* get in other direction */
      if (are_facts_excl_in_this_step(varr[i], v2))
        do_excl_in_this_step(varr[i], v2);
      }
    }
  }
}
/*************************END MUTUAL EXCLUSION PART**********************/



/************ ROUTINES FOR PRINTING OUT INFORMATION TO USER**************/
void print_info_piece(hashtable_t t, int flag)
{
  vertex_t vert;
  edgelist_t edge;
  get_next(t, 0); /* get ready */
  while((vert = get_next(t, 1)) != NULL) {
    if (vert->needed == 0 || IS_NOOP(vert)) continue;
    printf("%s\n",vert->name);
    if (flag>=2 && ((edge = vert->exclusive) != NULL)) {
      printf("   exclusive: "); /* now, includes noops */
      for(; edge; edge = edge->next)
      if (edge->endpt->needed) printf("%s ",edge->endpt->name);
      printf("\n");
    }
  }
}

/* for printing out info to user */
void print_info(int len)
{
  int i;
printf("Printing graph. For clarity, ignoring parts unreachable by planner\n");
  remove_unneeded_vertices(len);  /**just for printing **/

  for(i=0; i < len; ++i) {
    printf("\nTime: %d\n",i+1);
    print_info_piece(op_table[i], DEBUG_FLAG);
    if (DEBUG_FLAG >= 2) {
```

```
      printf("\nFacts: \n");
      print_info_piece(fact_table[i+1], 2); /* PRINT FACTS*/
    }
  }
}


/************ more random utilities ***************/
void read_initial_comments(FILE *fp)
{
  int c;
  while((c = getc(fp)) != '('); /* read up to first left paren */
}

int allocs_in_use;

void my_free(void * p) {
    allocs_in_use--;
    free(p);
}
void * my_alloc(int size) {
    void * p = malloc(size);
    allocs_in_use++;
    if (p == NULL) {
        fprintf(stderr, "Ran out of space.  Requested size=%d.\n", size);
        exit(1);
    }
    return p;
}

void print_alloc(void)
{ printf("allocs - frees = %d\n",allocs_in_use); }


void yyerror(char *x)
{
      printf("%s\n",x);
}

fact_list fact_list_append(fact_list f1, fact_list f2)
{
      if (f2 == NULL) return f1;

      if (f1 == NULL) return f2;

      f1->next = fact_list_append(f1->next, f2);

      return f1;
}


token_list token_list_append(token_list f1, token_list f2)
{
      if (f2 == NULL) return f1;

      if (f1 == NULL) return f2;

      f1->next = token_list_append(f1->next, f2);

      return f1;
}
void print_cant_do(int time)
{
  fprintf(stderr,"Can't solve in %d steps\n",time);
  printf("%d entries in hash table and %d hits.\n",hash_inserts, hash_hits);
  printf("   avg set size %d\n", hash_numctr/hash_ctr);
}
```

hashMod.c

```
/********************** Graphplan ************************************
  (C) Copyright 1995 Avrim Blum and Merrick Furst
      Modified by Rodrigo Chen and Tiago Vaquero

  All rights reserved. Use of this software is permitted for non-commercial
  research purposes, and it may be copied only for that use.  All copies must
  include this copyright message.  This software is made available AS IS, and
  neither the authors nor CMU make any warranty about the software or its
  performance.
*************************************************************************/

#include "graphplanMod.h"

/****This file contains a lot of stuff related to creation of the graph****/

/************* hash table functions ***************
 *
 * A hash table has type hashtable_t and is an array of lists of type
 * element_t.  A element_t is a pointer to a element_s structure.
 * A vertex_s structure contains a "name" field that is also the key.
 *
 * Hashing Method:  (key mod BIGPRIME) mod HSIZE.
 * Use result of first mod to do quick comparisons...
 *
 * lookup returns a pointer to the element_s structure or NULL.
 * insert creates a new element_s structure and inserts the name, returning
 *   the pointer.  Allocates new space for the name.
 * An element_s is the same as a vertex_s.  Same for element_t and vertex_t
 */
int num_created;
int hash(char *key, int size);

#define BIGPRIME 8000977
/* extern int NUMINTS; */

/**FOR BETTER PRINTING: remember to take out!!!!**/
/* #define HSIZE1 1                                    */
#define HSIZE1 HSIZE

int hash(char *key, int size)
{
  int h;
  for(h=0; *key != '\0'; key++)  h = (h*256 + (*key)) % size;
  return h;
}
element_s * lookup_from_table(hashtable_t htable, char *key)
{
  element_t l;
  int hval = hash(key, BIGPRIME);
  int index = hval % HSIZE1;
  for(l = htable[index]; l != NULL; l = l->next) {
    if (hval == l->hashval && strcmp(key,l->name) == SAME) return l;
  }
  return NULL;
}

/* inserts token list but ONLY IF IT'S NOT THERE ALREADY. Returns ptr. */
element_t insert_token_list(hashtable_t htable, token_list t)
{
  char str[100];
  element_t retval;
  instantiate_into_string(t, NULL, str,1);
  if ((retval = lookup_from_table(htable,str)) == NULL)
    retval = insert_into_table(htable, str);
  return retval;
}
```

```
element_t insert_into_table(hashtable_t htable, char *key)
{
  element_t l;
  int hval = hash(key, BIGPRIME);
  int index = hval % HSIZE1;
  l = (element_t) calloc(1, sizeof(element_s));
  l->name = (char *) calloc(1 + strlen(key), sizeof(char));
  strcpy(l->name, key);
  l->hashval = hval;
/*   l->exclusive_vect = (int *) calloc(NUMINTS, sizeof(int)); */
  l->next = htable[index];
  htable[index] = l;
  ++num_created;  /**global, use for info purposes **/
  return l;
}


/* if flag=0, just reset. Otherwise,
 * go from where you left off last (use static vars to hold state).
 * Return NULL if no more.
 * NOTE: this is DANGEROUS the way it's written since can't interleave
 * with different hash tables, etc.  Should really have flag be state.
 */
element_t get_next(hashtable_t h, int flag)
{
  static int i;
  static element_t current;
  element_t temp;
  if (flag==0) {i = 0; current = h[0]; return NULL;}
  while(1) {
    if (current != NULL) {
      temp = current;
      current = current->next;
      return temp;
    } else {
      if ((++i) >= HSIZE1) return NULL;
      current = h[i];
    }
  }
}
```

plannerMod.c

```
/*********************** Graphplan **************************************
  (C) Copyright 1995 Avrim Blum and Merrick Furst
      Modified by Rodrigo Chen and Tiago Vaquero

  All rights reserved. Use of this software is permitted for non-commercial
  research purposes, and it may be copied only for that use.  All copies must
  include this copyright message.  This software is made available AS IS, and
  neither the authors nor CMU make any warranty about the software or its
  performance.
************************************************************************/


/***** plannerMod.c: do the planning part.*********************************/


/************************************************************************

  About completeness check.  The basic idea of the completeness check
  is as follows.  Say the graph has leveled off at level N.  Let S be
  the the collection of unsolvable goal-sets we have stored at level N
  (in "bad_table").  Notice two things.  First, any plan to achieve
  our true goals must have at some point achieved some goal-set in S.
  Second, if a time step has passed in which S has not grown larger,
  then the collection of goal-sets at level N+1 is the same as S (this
  is because the graph has level off, so you can think of a time step
  on which we fail as shifting everything to the right).  That means
  that in order to achieve some goal set in S, we must previously have
  already achieved some other goal set in S, implying there is an
  infinite loop and the problem is not solvable.


************************************************************************/


#include "graphplanMod.h"

extern hashtable_t *fact_table, *op_table;  /* the arrays of hash tables */
extern int do_memo, do_subsets, do_lower, xviewing, do_buildup, do_completeness,
do_greedy, do_heuristic;
extern fact_list the_goals;

/*stuff for completeness */
extern int same_as_prev_flag, first_full_time, num_hashes[2];

int setup(int maxtime);
void insert_bad(goal_arr a, int num, int time);
void insert_good(goal_arr a, int num, int time);
int lookup_bad(goal_arr a, int num, int time);
int lookup_good(goal_arr a, int num, int time);
int subset_lookup(goal_arr a, int num, int time);
int PARTIALsubset_lookup(goal_arr a, int num, int time); /*using THIS one now*/
int can_hope_to_solve(goal_arr g, int length, int time);
int build_up_to_plan(int num_goals, int time);
int constrict(int num_goals, int time);
int basicplan(int cindex, int nindex, int time);
int runbasicplan(int numgoals, int time);
void mark_exclusive(vertex_t v);
void unmark_exclusive(vertex_t v);

int hash_inserts = 0, hash_hits = 0, hash_ctr = 0, goodhash_hits = 0,
  hash_numctr = 0, subset_hits = 0;
extern int bvec_len;
int number_of_actions_tried = 0;
int for_printing;  /* if 1, then leave "used" flags in, else don't */
int global_maxtime;  /* global variable containing maxtime */
/*****use for storing current goals*****/
```

```
goal_arr *goals_at;


int do_plan(int maxtime)
{
  int num_goals, i, result;
  num_goals = setup(maxtime);
  global_maxtime = maxtime;
  printf("goals at time %d:\n  ",maxtime+1);
  for(i=0; i < num_goals; ++i) printf("%s ",goals_at[maxtime][i]->name);
  printf("\n\n");
  if (do_lower && !can_hope_to_solve(goals_at[maxtime],num_goals,maxtime)) {
    if (do_memo) insert_bad(goals_at[maxtime], num_goals, maxtime);
    return 0;
  }
  for_printing = 1;
  if (do_buildup) result = build_up_to_plan(num_goals, maxtime);
  else if (do_greedy) result = constrict(num_goals, maxtime);
  else result = runbasicplan(num_goals, maxtime);
  if (result == 0 && do_memo)insert_bad(goals_at[maxtime], num_goals, maxtime);
  if (result == 0 && xviewing == 1) { /* clean up the viewer */
    for(i=0; i < num_goals; ++i) {
      draw_fact(goals_at[maxtime][i],maxtime,0);
    }
  }
  return result;
}


/* Print out info.  The best match to a "state-space step" in something like
   Prodigy is somewhere between the number of set-creation steps and the
   number of actions tried. */
void print_plan(int maxtime)
{
  int i;
  vertex_t v;
  for(i=0; i < maxtime; ++i) {
    get_next(op_table[i],0);
    while((v = get_next(op_table[i],1)) != NULL) {
      if (v->used && !IS_NOOP(v)) printf("%d %s\n",i+1,v->name);
    }
  }
  if (do_memo) { /* print out some useful info */
    printf("%d entries in hash table, ",hash_inserts);
    if (hash_inserts == 0) printf("\n");
    else printf("%d hash hits, avg set size %d.\n",
            hash_hits, hash_numctr/hash_ctr);
    printf("%d total set-creation steps (entries + hits + plan length - 1).\n",
         hash_inserts + hash_hits + maxtime - 1);
  }
  if (do_subsets) printf("%d subset hits.\n", subset_hits);
  if (do_greedy) printf("%d hits on doable sets\n", goodhash_hits);
  printf("%d actions tried\n", number_of_actions_tried);
}


/******Routines for checking proactively if we can stop already **********/

/* when you make a decision, immediately
   mark each op/noop as not doable, by doing MARK(v) (which adds 1 to
   v->cant_do).  Test by doing IS_MARKED(v).
   */
#define MARK(v) ((v)->cant_do += 1)
#define UNMARK(v) ((v)->cant_do -= 1)
#define IS_MARKED(v) ((v)->cant_do > 0)
```

```
void mark_exclusive(vertex_t v)
{
  edgelist_t e;
  for(e = v->exclusive; e; e = e->next) MARK(e->endpt);
}

void unmark_exclusive(vertex_t v)
{
  edgelist_t e;
  for(e = v->exclusive; e; e = e->next) UNMARK(e->endpt);
}

/**** see if remaining goals still possible **/
int goals_still_possible(int index, int time)
{
  int i;
  edgelist_t e;
  for(i=0; i < index; ++i) {
    for(e = goals_at[time][i]->in_edges;e && IS_MARKED(e->endpt);e = e->next);
    if (e == NULL) return 0;  /* all were marked */
  }
  return 1;
}

/* this is a minimality check: give up right now if there are any
   operators being used at the current level that can be removed while
   keeping it a legal plan.  Note: we are guaranteed that all minimal sets
   of operators will be tried.

   Note: we're given the goal set in the PREVIOUS time....

  Note: At one point had in planner that won't consider any op that
  adds a previous goal.  Reason: either already tried it (so cannot be good)
  or else should be tried with prev goal in mind so that you don't get
  unnecessarily-many goals in goals_at[time-1]. The problem is you could get
  a situation where there are only two ways to make goal #1 true.  One makes
  goal #2 true and the other makes goal #3 true, both of which are later in
  the list than goal #1.
*/
int action_set_is_minimal(int index, int time)
{
  int i;
  edgelist_t e, e2;
  vertex_t op;
  for(i=0; i < index; ++i) {
    for(e = goals_at[time][i]->out_edges; e; e = e->next) {
      if (!(op = e->endpt)->used) continue;  /* not an op we're using */
      for(e2 = op->out_edges; e2; e2 = e2->next) {
      if (!e2->endpt->used) continue;  /* not one of our goals */
      if (e2->endpt->is_true == 1) break; /* op is needed */
      }
      if (e2 == NULL) { /* Hey, the op wasn't needed after all! */
      return 0;
      }
    }
  }
  return 1;
}

/* helper for setting up */
int runbasicplan(int numgoals, int time)
{
  int i, result;
  for(i=0; i < numgoals; ++i) goals_at[time][i]->used = 1;
  result = basicplan(numgoals-1, 0, time);
  for(i=0; i < numgoals; ++i) goals_at[time][i]->used = 0;
```

```
  return result;
}

/***********End of checking routines*************************************/


/*********** THIS IS THE MAIN RECURSIVE BACKWARD CHAINING SEARCH **********
   This is a recursive planner.  Two interesting things it does are
   (1) use the mutual-exclusivity relations, and (2) go layer by layer.

   Idea: Given an array of goals at some time, for each way of generating
   these goals from facts in previous time step that doesn't violate
   exclusivity constraints, try running recursively.

   Returns 1 if doable and 0 if not.

   Arguments:
    cindex tells us the index of the Current goal we're working on.
    nindex is the next free index at the previous level(the one we're creating)
      We use this to put preconditions of our ops into.
    time is the current time.

   This examines the goals one by one (starting with the one of highest
   index in the goals_at[time] array) and for each, looks at all ways of
   creating it (that aren't exclusive of previous commitments).  For each
   such way, we first check to see if some future goal has been "cut off",
   and if not, we put the preconditions into goals_at[time-1][nindex].

   Note: NOOP is the first one in list of ways to make a goal true.
   This routine relies on the exclusivity relations between ops (and noops)
   being in the graph.  I.e., it does no other (redundant) checking to
   make sure plan is LEGAL.

   The planner is proactive in the sense that it checks to see if remaining
   goals in the list are still possible. (this is what is meant above by
   "cut off")

   We tried using the heuristic of having the next goal be the
   one with the fewest ways of making it true (rather than the next one
   in the list) but it didn't generally help much.
   */
int basicplan(int cindex, int nindex, int time)
{
  int i, result, was_used;
  vertex_t v, op;
  edgelist_t e, edge2;
  if (time <= 0) return 1;  /* we're done */

  if (cindex < 0) { /* go on to the previous time step */
    if (DEBUG_FLAG) {
      printf("goals at time %d:\n  ",time);
      for(i=0; i < nindex; ++i) printf("%s ",goals_at[time-1][i]->name);
      printf("\n\n");
    }
    if (do_memo) {
      if (lookup_bad(goals_at[time-1], nindex, time-1)) return 0;
      if (do_greedy && lookup_good(goals_at[time-1], nindex, time-1)) return 1;
      if (do_subsets) {  /* option to see if a subset is inside */
      if(PARTIALsubset_lookup(goals_at[time-1], nindex, time-1)) {
        insert_bad(goals_at[time-1],nindex, time-1);/*find faster next time*/
        return 0;
      }
      }
    }

    /** do minimality check **/
```

```
  if (!action_set_is_minimal(nindex, time-1)) {
    printf("found non-minimal action set, time %d\n",time);
    return 0;
  }

  /** if desired, do lower bound check **/
  if (do_lower && !can_hope_to_solve(goals_at[time-1],nindex, time-1)) {
    insert_bad(goals_at[time-1],nindex, time-1);
    return 0;
  }
  /* do greedy check */
  if (do_greedy) result = constrict(nindex, time-1);
  else result = basicplan(nindex-1, 0, time-1);
  if (result == 0 && do_memo) insert_bad(goals_at[time-1], nindex, time-1);
  return result;
}

/* Let v be the next goal, skipping over any that may just by luck have
   been made true by the operators we've already committed to */
while((v = goals_at[time][cindex])->is_true) {
  --cindex;
  if (cindex < 0) return basicplan(-1, nindex, time);
}

/* Try the ops and noops */
for(e = v->in_edges; e; e = e->next) {
  op = e->endpt;
  if (IS_MARKED(op)) continue; /* exclusive of something used*/

  /* ok, do it. */
  mark_exclusive(op);
  /* check to see if causes problems for future goals */
  if (!goals_still_possible(cindex,time)) {
    unmark_exclusive(op);
    continue;
  }
  op->used++;

  /* THIS SORT-OF CORRESPONDS TO STATE-SPACE STEPS */
  if (!IS_NOOP(op)) ++number_of_actions_tried;

  if (xviewing) draw_op(op, time-1, 1, 1); /* XVIEWING */

  for(edge2 = op->in_edges; edge2; edge2 = edge2->next) {
    was_used = edge2->endpt->used++;
    if (!was_used) goals_at[time-1][nindex++] = edge2->endpt;
    if (nindex > MAXGOALS) do_error("MAXGOALS too small");
  }
  for(edge2 = op->out_edges; edge2; edge2 = edge2->next)
    ++edge2->endpt->is_true;

  /* now recurse */
  if (DEBUG_FLAG > 1) {
    if (IS_NOOP(op)) printf("trying %s from prev time step.\n",v->name);
    else printf("Trying %d %s to achieve %s.\n",time, op->name, v->name);
  }
  result = basicplan(cindex-1, nindex, time);

  /* undo */
  for(edge2 = op->out_edges; edge2; edge2 = edge2->next)
    --edge2->endpt->is_true;
  for(edge2 = op->in_edges; edge2; edge2 = edge2->next) {
    was_used = --edge2->endpt->used;
    if (!was_used) goals_at[time-1][--nindex] = NULL;
  }
  unmark_exclusive(op);
```

```
    if (result == 1 && for_printing) return 1;   /* DONE */

    op->used--;
    if (xviewing) draw_op(op, time-1, 0, 1);  /* XVIEWING */

    if (result == 1) return 1;                  /* DONE */
  }
  /* otherwise, return 0 */
  return 0;
}



/**************Here are a couple more helpers/heuristics*****************
 **************that can be turned on by using options.*****************/


/* This routine takes a goal array, its length, and a time and
   returns 0 if there is just no hope of being able to solve the goals.

   Currently, it does the following reasoning: if it can find a subset
   of the goals whose excl_in_this_step form a clique of size s, then
   the time had better be at least s.
 */
int can_hope_to_solve(goal_arr g, int length, int time)
{
  int i, yesind, noind, maxval, maxind;
  vertex_t tempv;
  edgelist_t edge;
  goal_arr arr;
  static int *count, *marked_as_nbr, flag = 0;

  if (flag == 0) {  /* doing this in case MAXGOALS is not a constant */
    count = (int *) malloc(MAXGOALS*sizeof(int));
    marked_as_nbr = (int *) malloc(MAXGOALS*sizeof(int));
    flag = 1;
  }

  /* if time = 0, special case: just return yes */
  if (time==0) return 1;

  /* copy over goal array and initialize: give each vertex the index in arr*/
  for(i=0; i < length; ++i) {
    arr[i] = g[i];
    count[i] = 0;
    arr[i]->junk = i; /* using junk ptr to hold this info */
  }
  yesind = 0;  /* "potential" ones for clique between yes and no indices */
  noind = length;

  /* set up counts */
  for(i=0; i < length; ++i) {
    for(edge = arr[i]->excl_in_this_step; edge; edge = edge->next) {
      if (arr[edge->endpt->junk] == edge->endpt) ++count[i];
    }
  }

  /* get clique based on greedy strategy: pick node with highest degree and
     put in. then of all nbrs, put in node of next highest degree (into the
     set of vertices still under consideration), etc.
   */
  while(yesind < noind) {
    /* find highest degree (quit if not big enough) and put it into yes pile*/
    for(i=yesind, maxval = 0; i < noind; ++i) {
      if (count[i] > maxval) {maxval = count[i]; maxind = i;}
```

```
    }
    if (maxval < time) return 1;
    tempv = arr[yesind]; /* the lowest in "potential" set */
    arr[yesind] = arr[maxind];
    arr[maxind] = tempv;
    arr[yesind]->junk = yesind;
    arr[maxind]->junk = maxind;
    yesind++;

    /* begin thinking all are non-neighbors */
    for(i=yesind; i < noind; ++i) marked_as_nbr[i]=0;

    /* mark as no longer potential all the non-neighbors, and decrease counts*/
    /* first, mark all neighbors in "potential" set*/
    for(edge = arr[yesind-1]->excl_in_this_step; edge; edge = edge->next) {
      tempv = edge->endpt;
      if (tempv->junk < noind && arr[tempv->junk] == tempv)
      marked_as_nbr[tempv->junk] = 1;
    }
    /* now move all non-neighbors into "no" region, and for each one, decrease
       counts */
    for(i=yesind; i < noind; ++i) {
      if (!marked_as_nbr[i]) {
      tempv = arr[noind-1];
      arr[noind-1] = arr[i];
      arr[noind-1]->junk = noind-1;
      arr[i] = tempv;
      arr[i]->junk = i;
      count[i] = count[noind-1];
      count[noind-1] = 0;          /* might as well zero it out */
      --noind;                     /* decrease noindex pointer */
      for(edge = arr[noind]->excl_in_this_step; edge; edge = edge->next) {
        if (arr[edge->endpt->junk] == edge->endpt)
          --count[edge->endpt->junk];
      }
      }
    }

  }
  if (yesind > time) {
    if (DEBUG_FLAG) printf("Can't hope to solve by time %d.\n",time);
    return 0;
  }
  else return 1;
}


/* Build up incrementally to a plan for entire set.  Start with first goal,
   and if successful go to a plan for the first two, etc.  Store record
   so that if at time t we succeed on first i goals, but not first i+1, then
   at time t+1 we start with i+1. */
int build_up_to_plan(int num_goals, int time)
{
  static int i = 0;
  int j,success, oldfp = for_printing;
  for(;i < num_goals; ++i) {
    if (i != num_goals-1) {
      printf("trying first %d goals\n",i+1);
      for_printing = 0;
    } else for_printing = oldfp;
    if (xviewing) {      /* XVIEWING */
      reset_viewer(time);
      for(j=0; j <=i; ++j) draw_fact(goals_at[time][j], time, 1);
    }
    success = runbasicplan(i+1,time);
    if (!success) {
```

```
      printf("failed\n");
      if (do_memo) insert_bad(goals_at[time], num_goals, time);
      for_printing = oldfp;
      return 0;
    }
  }
  return 1;
}

/* Just for experimentation..... */
/* try greedily constricting a set until find minimal one causing failure. Do
   "hold one out" for each possibility
 */
int constrict(int num_goals, int time)
{
  goal_arr storage;
  vertex_t temp, temp2;
  int i, j, num_current, result, oldfp;

  for(i=0; i < num_goals; ++i) storage[i] = goals_at[time][i];
  if (runbasicplan(num_goals, time)) return 1; /* success */

  oldfp = for_printing;
  for_printing = 0;
  for(num_current = num_goals-1; num_current > 0; --num_current) {
    temp = goals_at[time][num_current];
    for(j = num_current-1; j >= 0; --j) {
      result = runbasicplan(num_current, time);
      if (result == 0) {
      if (do_memo) insert_bad(goals_at[time], num_current, time);
      break; /* failed on subset */
      }
      insert_good(goals_at[time], num_current, time);
      temp2 = goals_at[time][j];
      goals_at[time][j] = temp;
      temp = temp2;
    }
    if (j < 0) break;  /* all succeeded, so return right away */
  }

  for(i=0; i < num_goals; ++i) goals_at[time][i] = storage[i];
  for_printing = oldfp;
  return 0;
}
/***************End of options code*************************************/


/*******************Hashing/memoizing routines*******************/
/****************************************************************/
/* These are the functions for memoizing.  Idea: store in a hash table
   the sets of goals that have been seen so far.
   Since we're just doing lookup, we might as well use a big table.

   For hash table, look at rand1 values. Just add the
   values to hash since in fact you WANT to have lists that are same
   as sets but different as sequences to collide.  Will also store the actual
   sum, since it's likely will only have equal sums when really are same
   sets.

   NEW: store sets as bit-vectors of length MAXNODES. That way can compare
   with just a few equality or bitwise AND tests. (also need to check they
   are at the same time)

 */
```

```
#define P_HSIZE 8192  /* planner hash table size */
#define P_HASH 8191 /* 111111111111  */
#define BVEC_LEN ((MAXNODES - 1)/32 + 1)

/* store sets as bit-vectors: two sets equal if same vector at same time */
typedef unsigned int *bit_vector;

typedef struct SET_ELT {
  bit_vector item;
  int num;
  long sum;
  int time;
  struct SET_ELT *next;
  struct SET_ELT *thread;
} *hash_entry;

typedef hash_entry set_table[P_HSIZE];
set_table bad_table, good_table;
hash_entry *global_list;  /* global_list[i] is a hash_entry */
int max_time_seen = -1;

/* sum instead of xor since doesn't seem to matter */
long sum_arr(goal_arr a, int num)
{
  int i;
  unsigned long sum;
  for(i=0,sum=0; i < num; i++)  sum = sum + a[i]->rand1;
  return sum;
}

/* turn a goal array into a bit vector. Assume there is space in v. */
/* NOTE: if levels are equal, then uid_*'s should be equal too */
void make_bit_vector(goal_arr a, int len, bit_vector v)
{
  int i;
  vertex_t w;
  for(i=0; i < bvec_len; ++i) v[i] = 0;  /* zero out first */
  for(i=0; i < len; ++i) {
    w = a[i];
    v[w->uid_block] = v[w->uid_block] | w->uid_mask;
  }
}

/* make more space for global list */
void fix_global_list(int time)
{
  hash_entry *temp;
  int i;
  temp = global_list;
  global_list = (hash_entry *) calloc(time+1, sizeof(hash_entry));
  for(i=0; i <= max_time_seen; ++i) global_list[i] = temp[i];
  if (temp) free(temp);
  max_time_seen = time;
}


/***Doing the arbitrary subset case ***/
/***This is currently not being used, but to use it, just replace calls
    to PARTIALsubset_lookup by calls to this.  This routine actually
    does help a lot sometimes, but for it to be generally effective, we
    need a good data structure to acess the subsets more quickly ***/

int subset_lookup(goal_arr a, int num, int time)
{
  static bit_vector query;
  static int flag = 0;
```

```
  int j;
  hash_entry elt;
  if (flag == 0) {  /* (since MAXNODES not constant) */
    query = (bit_vector) malloc(BVEC_LEN*sizeof(int));
    flag = 1;
  }
  if (time > max_time_seen) fix_global_list(time); /* make more space */

  make_bit_vector(a, num, query);   /* "query" is what we actually check*/
  for(elt = global_list[time]; elt; elt = elt->thread) {
    for(j=0; j < bvec_len; ++j) {
      if ((query[j] | elt->item[j]) != query[j]) break;
    }
    if (j == bvec_len) { /* same */
      ++subset_hits; /* just for info */
      return 1;
    }
  }
  return 0;
}


/***** Since it seems hard to find if an arbitrary subset is in the
  hash table, let's just see if a set of size one smaller is in the
  hash table.
  ****/
int PARTIALsubset_lookup(goal_arr a, int num, int time)
{
  long sum, current;
  static bit_vector v;
  static int flag = 0;
  int i, j, index, vind;
  hash_entry elt;
  if (flag == 0) {
    v = (bit_vector) malloc(BVEC_LEN*sizeof(int));
    flag = 1;
  }
  sum = sum_arr(a, num);
  make_bit_vector(a, num, v);
  /* try removing one */
  for(i=0; i < num; ++i) {
    current = sum - a[i]->rand1;
    vind = a[i]->uid_block;
    v[vind] = v[vind] ^ a[i]->uid_mask;   /* flip that bit */
    index = (int) (current & P_HASH);
    /* if (index < 0) index += P_HSIZE; */
    for(elt = bad_table[index]; elt; elt = elt->next) {
      if (elt->sum != current) continue;  /* not same */
      if (elt->time != time) continue;    /* not same */
      /* if get here, probably the same, but need to verify. */
      for(j=0; j < bvec_len; ++j) {
      if (v[j] != elt->item[j]) break;
      }
      if (j == bvec_len) { /* what we expect */
      ++subset_hits;
      return 1;
      }
    }
    v[vind] = v[vind] ^ a[i]->uid_mask;   /* flip that bit back */
  }
  return 0;
}


/* doing lookup with verification.  Other option is to hash both
   random values and to use other hash for probabilistic verification
```

```
    storing as bit vectors so can check equality easily
  */
 int set_lookup(set_table table, goal_arr a, int num, int time)
 {
   long sum;
   static bit_vector query;
   static int flag = 0;
   int index, i;
   hash_entry elt;

   if (flag == 0) {  /* (since MAXNODES not constant) */
     query = (bit_vector) malloc(BVEC_LEN*sizeof(int));
     flag = 1;
   }
   ++hash_ctr;  /* just for info */
   hash_numctr += num; /* just for info */
   sum = sum_arr(a, num);
   index = (int) (sum & P_HASH);  /* BITWISE AND */
   /*  if (index < 0) index += P_HSIZE; */
   make_bit_vector(a, num, query);   /* "query" is what we actually check*/
   for(elt = table[index]; elt; elt = elt->next) {
     if (elt->sum != sum) continue;   /* not same */
     if (elt->time != time) continue; /* not same */
     /* if get here, probably the same, but need to verify */
     for(i=0; i < bvec_len; ++i) {
       if (query[i] != elt->item[i]) break;
     }
     if (i == bvec_len) return 1;
   }
   return 0;
 }

 int lookup_bad(goal_arr a, int num, int time)
 {
   int val = set_lookup(bad_table, a, num, time);
   if (val == 1) {
     if (DEBUG_FLAG) printf("hash table hit.\n");
     ++hash_hits; /* just for info */
   }
   return val;
 }

 int lookup_good(goal_arr a, int num, int time)
 {
   int val = set_lookup(good_table, a, num, time);
   if (val == 1) {
     if (DEBUG_FLAG) printf("good hash table hit.\n");
     ++goodhash_hits; /* just for info */
   }
   return val;
 }


 /* insert into table.  Return pointer */
 hash_entry set_insert(set_table table, goal_arr a, int num, int time)
 {
   long sum;
   int index;
   static int mem_flag = 0;
   hash_entry elt;
   sum = sum_arr(a, num);
   index = (int) (sum & P_HASH);
   /* if (index < 0) index += P_HSIZE; */
   elt = (struct SET_ELT *) malloc(sizeof(struct SET_ELT));
   if (elt == NULL) { /* out of memory */
```

```
    if (mem_flag == 0)
      ++mem_flag, printf("no memory at insert %d\n",hash_inserts);
    return NULL;
  }
  elt->num = num;
  elt->sum = sum;
  elt->time = time;
  elt->item = (bit_vector) malloc(BVEC_LEN*sizeof(int));
  make_bit_vector(a, num, elt->item);
  elt->next = table[index];
  table[index] = elt;
  return elt;
}

void insert_bad(goal_arr a, int num, int time)
{
  int i;
  static goal_arr just_for_debugging;
  hash_entry elt = set_insert(bad_table, a, num, time);
  if (do_subsets) {
    if (time > max_time_seen) fix_global_list(time);
    elt->thread = global_list[time];
    global_list[time] = elt;
  }
  ++hash_inserts;  /* global: just for info */
  if (do_completeness && same_as_prev_flag) { /* info for completeness test */
    if (time == first_full_time) ++num_hashes[1];
    if (time == first_full_time + 1) { /* JUST FOR DEBUGGING */
      for(i=0;i<num;++i) just_for_debugging[i] = a[i]->prev_time;
      if (!set_lookup(bad_table, just_for_debugging,num,time-1)){
      /* SHOULD NEVER HAPPEN */
      printf("Error! set in table at time %d not in at time %d.\n",
             first_full_time+1, first_full_time);
      for(i=0;i<num;++i) printf("%s ",a[i]->name);
      printf("\n");
      exit(1);
      }
    }
  }
}

void insert_good(goal_arr a, int num, int time)
{
  set_insert(good_table, a, num, time);
}

/******************** set up ****************************/
void handle_events(void);
void do_graph_created(void);

/* set up goal array "goals_at" and mark the initial facts as "is_true".
   Return number of goals.  Note: possibly might be some nodes in
   fact_table[maxtime] that aren't goals.
 */
int setup(int maxtime)
{
  vertex_t v;
  fact_list temp;
  char str[100];
  int i = 0, j;
  /* set up initial facts */
  get_next(fact_table[0],0);
  while((v = get_next(fact_table[0],1)) != NULL) v->is_true = 1;

  if (xviewing) reset_viewer(maxtime);  /* XVIEWING */
  if (xviewing == 2) {  /* draw whole graph */
```

```
    for(j=0; j < maxtime+1; ++j) {
      /* facts */
      get_next(fact_table[j],0);
      while((v = get_next(fact_table[j],1)) != NULL) draw_fact(v, j, 1);
      handle_events();
      /* ops */
      if (j == maxtime) break;
      get_next(op_table[j],0);
      while((v = get_next(op_table[j],1)) != NULL)
      if (v->out_edges) draw_op(v, j, 1, 0);
    }
  }

  /* set up goal array */
  if (goals_at != NULL) free(goals_at);
  goals_at = (goal_arr *) calloc(maxtime+1,sizeof(goal_arr));
  for(temp=the_goals; temp; temp = temp->next) {
    instantiate_into_string(temp->item, NULL, str,1);
    if ((v = lookup_from_table(fact_table[maxtime],str)) == NULL)
      do_error("goal not found in output");
    goals_at[maxtime][i++] = v;
    if (xviewing) draw_fact(v, maxtime, 1);  /* XVIEWING */
    if (i > MAXGOALS) do_error("MAXGOALS too small.");
  }
  return i;
}
```

dummyMod.c

```
/* contains dummy routines for compiling without certain features */
#include "graphplanMod.h"

void setup_viewer(void)
{ printf("Sorry, can't do X viewing in this version.\n"); }

void wait_until_left(void)
{ int i=0; }

void reset_viewer(int max_time)
{ int i = 0;}
void draw_fact(vertex_t v, int time, int flag)
{ int i=0; }
void draw_op(vertex_t v, int time, int flag, int thick)
{ int i=0; }

void do_final_viewing(void) {int i = 0; }

void do_graph_created() { int i=0; }

void handle_events() { int i=0; }
```