# 7
# PLANSIM, um arcabouço para agentes planejadores

É reconhecido que planejadores projetados para a solução de um domínio específico, ou especializados, possuem desempenho melhor do que os planejadores de propósito geral [29]. A principal desvantagem dos planejadores especializados está no grande esforço tipicamente envolvido em sua construção. Inicialmente, o projetista de um sistema do gênero deve definir a técnica a ser utilizada, entre as várias disponíveis. Uma vez definida a técnica, existem ainda diversos testes que devem ser realizados. Por exemplo, se o sistema utilizar técnicas de programação linear inteira mista podem-se testar diversas modelagens, como feito em [3]. Se o sistema utilizar busca heurística, podem-se testar funções heurísticas e estratégias de busca e controle distintas, como mostrado na Seção 6.3.

Este grande esforço é o principal motivador para a pesquisa em planejadores de propósito geral, avaliados na Seção 4.3. No entanto, como vimos na Seção 6.3.1, a qualidade das funções heurísticas geradas automaticamente por estes planejadores pode ser ultrapassada facilmente com funções heurísticas simples que utilizem conhecimento específico do domínio. Além disso, a limitada capacidade do modelo utilizado para representar o sistema a ser planejado pode dificultar artificialmente o processo, como descrito na Seção 6.4. Estes dois fatores são grandes obstáculos para a utilização deste tipo de tecnologia no momento para a solução de problemas operacionais.

Apenas recentemente a utilização de arcabouços de *software* começou a ser considerada pela comunidade de planejamento em IA [37]. O arcabouço Planning4J[38] é direcionado aos desenvolvedores de planejadores de propósito geral, e foi incorporado recentemente ao arcabouço ABLE [39], cujo propósito é a construção de sistemas multi-agentes autônomos, também independentes de domínio e que suportam diversos paradigmas de IA como redes neurais, sistemas de inferência baseados em regras, etc. Ambos os arcabouços são para a plataforma Java.

O PLANSIM é um arcabouço de software para o desenvolvimento de planejadores especializados. Ele define uma arquitetura para planejadores

que utilizem busca heurística direta como mecanismo de planejamento e simuladores de eventos discretos para modelo do sistema, fixando as interfaces de componentes como objetivos, funções heurísticas e simuladores. Além disso, provê diversas estratégias de busca, cujo desempenho pode ser comparado facilmente para o problema de planejamento que está sendo analisado. Ele também permite que heurísticas específicas do domínio sejam construídas e avaliadas. Desta forma, o esforço necessário para a construção e testes de planejadores especializados é significativamente reduzido.

A decisão pelo foco na busca heurística direta como mecanismo de planejamento é motivada pelo fato de que planejadores deste tipo foram os que apresentaram o melhor desempenho nas últimas edições da IPC, como vimos na Seção 4.4.

O PLANSIM [41] é desenvolvido em C++, sendo um projeto de código aberto[1]. Planejadores que utilizem o PLANSIM como base ficam restritos aos mecanismos de busca implementados pelo mesmo. No entanto, o arcabouço já suporta vários mecanismos comumente utilizados, e pode ser estendido para suportar outros com relativa facilidade.



## 7.1 Estrutura do arcabouço

Esta Seção realiza uma descrição geral do arcabouço, cuja estrutura é mostrada na Figura 7.1. Muitos dos termos utilizados nesta Seção relativos ao processo de planejamento são exemplificados na Seção 6.1.

As classes derivadas de *search_algorithm* e a classe *enforced_hill_climbing* são responsáveis pela execução das estratégias de busca do PLANSIM. Elas são detalhadas na Seção 7.1.2. As classes *search_nodes_manager*, *search_node* e *merit_search_node* são finais e responsáveis pelo gerenciamento do espaço de estados explorado. Elas são detalhadas na Seção 7.1.1. As classes *plan*, *action*, *action_set*, *goal* e *evaluation* representam os conceitos clássicos da área de planejamento, com *evaluation* representando as funções heurísticas. Algumas destas classes são *hot spots* e devem ser especializadas durante a instanciação do arcabouço. Elas são detalhadas na Seção 7.1.4. As classes *simulator* e *state* definem a interface necessária para os simuladores a eventos discretos e a representação de seus estados, sendo detalhadas na Seção 7.1.3. Finalmente, a classe *problem_instance* é responsável pela representação de uma instância de um problema de planejamento, contendo referências para o simulador utilizado,

[1] Disponível em http://plansim.sourceforge.net

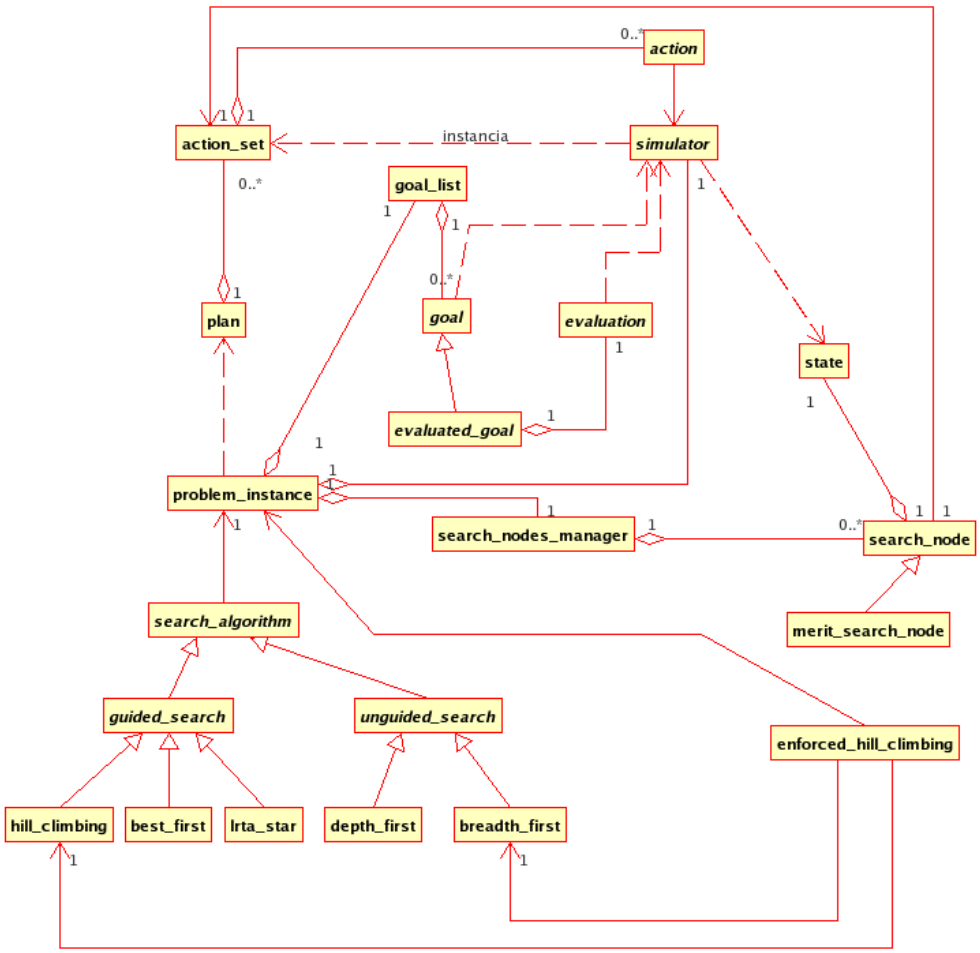


Figura 7.1: Estrutura geral do PLANSIM

os objetivos e o gerenciador de estados explorados. Ela é detalhada na Seção 7.1.5.

Para tornar a apresentação mais clara, este capítulo referencia apenas alguns métodos das classes do arcabouço, e a assinatura destes também não é mostrada em todos os casos. A documentação completa do arcabouço pode ser obtida no sítio do PLANSIM na internet.

### 7.1.1 Gerenciamento do espaço de estados

Conforme vimos na Seção 6.1, as estratégias de busca implementadas pelo PLANSIM precisam, no decorrer do processo, armazenar alguns estados já visitados do simulador e detectar eventuais duplicações na geração destes, impedindo que estados idênticos sejam gerados mais de uma vez no processo. O conjunto de classes descrito na Figura 7.2 implementa estes serviços.

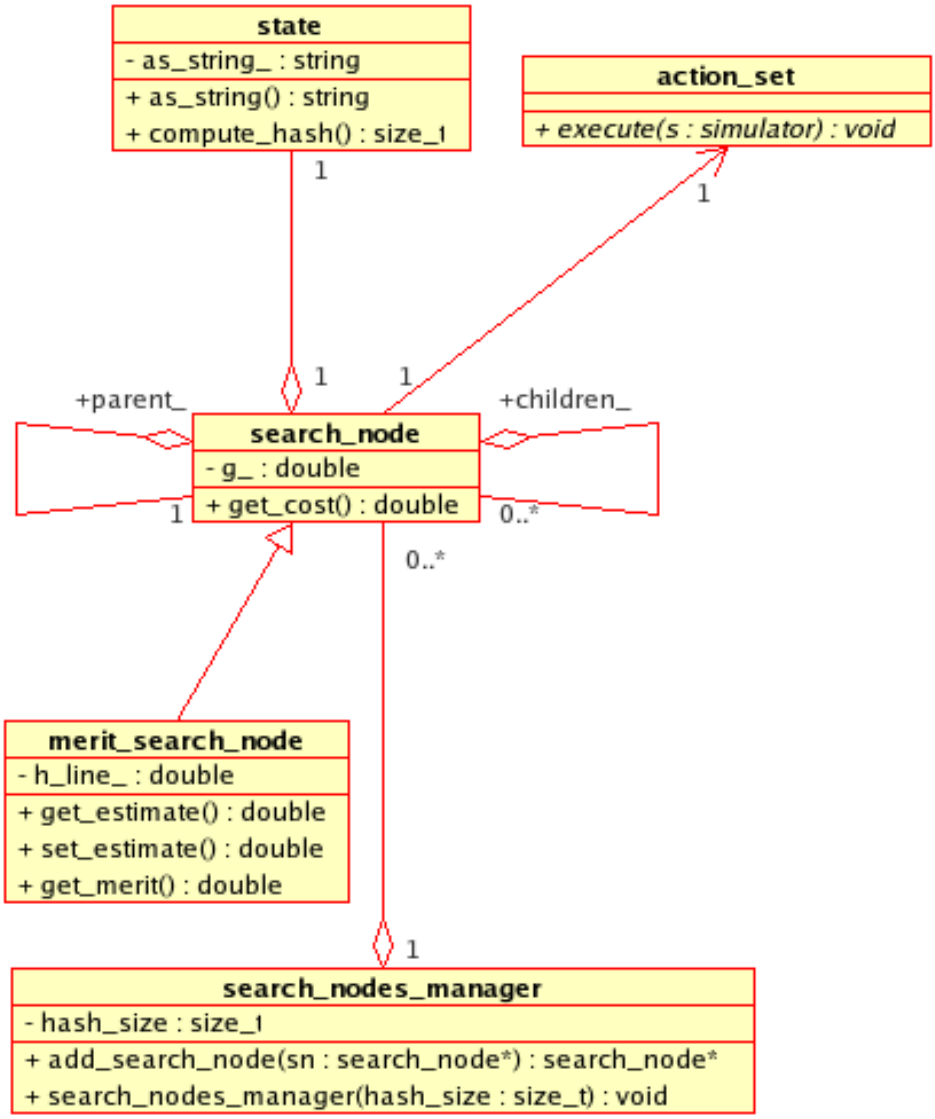


Figura 7.2: Gerenciamento do espaço de estados visitados

A classe *search_node* contém cada estado gerado, representado pela classe *state*, e as ações que foram aplicadas a partir do estado anterior para a sua geração, representada pela referência ao *action_set*, classe que é detalhada mais adiante. Os *search_nodes* são armazenados em uma estrutura de árvore, representada por referências aos nós sucessores e ao predecessor do mesmo. Estas referências são utilizadas, por exemplo, para a construção de um plano a partir de um estado que satisfaça os objetivos definidos para o problema de planejamento. Ele é montado através da navegação do estado final até o nó raiz que representa o estado inicial, verificando em cada passo o *action_set*.

A classe *merit_search_node* é uma especialização de *search_node* utilizada quando o estado representado possui também uma avaliação. Esta avaliação, representada pelo atributo *h_line_* é uma estimativa da distância deste estado a um estado objetivo, gerada pela função heurística.

A classe *search_nodes_manager* mantém uma lista com o código hash de todos os *search_nodes* já gerados, de forma a permitir a detecção eficiente de estados duplicados. Esta verificação é realizada no momento da chamada ao método *search_nodes_manager::add_search_node*.



### 7.1.2 Estratégias de busca

Um procedimento geral para muitas estratégias de busca direta pode ser representado da seguinte forma.

1. $Soluciona(Nos)$
2. Se $Nos$ está vazio, retorne $Falha$
3. Senão
4. $No = SelecionaNo(Nos)$
5. $Resto = Nos - No$
6. Se $No$ é objetivo, retorne $Sucesso$
7. Senão
8. $Filhos = ExpandeNo(No)$
9. $NovosNos = AdicionaNos(Filhos, Resto)$
10. Retorne $Soluciona(NovosNos)$

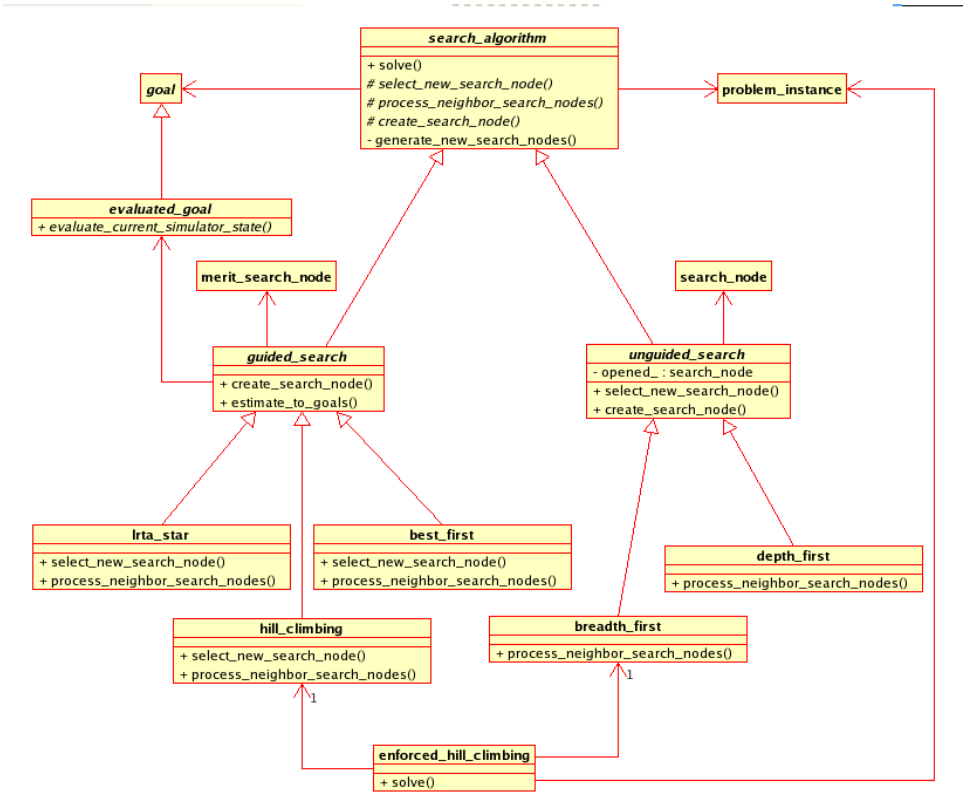


Figura 7.3: Implementação das estratégias de busca

O comportamento específico de cada estratégia pode ser gerado a partir da definição como ela trata os próximos estados possíveis, *AdicionaNos* e qual o critério utilizado para a seleção do próximo estado a ser explorado, *SelecionaNo*. Por exemplo, uma estratégia de busca em amplitude armazena todos os nós gerados ao final de uma lista de nós abertos, e escolhe como o próximo nó a ser visitado o primeiro elemento da lista. Já a busca em profundidade armazena os nós gerados no início da lista de nós abertos, selecionando também o primeiro elemento da lista. Uma busca do tipo subida de encosta não armazena os nós gerados, escolhendo simplesmente o melhor vizinho do estado corrente como o próximo estado a ser visitado.

A Figura 7.3 mostra como as estratégias de busca são implementadas pelo PLANSIM. A classe *search_algorithm* implementa o procedimento geral de busca, disparado pela execução do método *solve*, e define os métodos virtuais *select_new_search_node* e *process_neighbors_search_nodes*, responsáveis pela definição de *SelecionaNo* e *AdicionaNos*, respectivamente. Além destes, o método *create_search_node* também é definido como virtual já que os *search_nodes* podem ou não serem especializados em *merit_search_nodes*.

A classe *unguided_search* captura o comportamento comum das buscas exaustivas não guiadas. Como nestas todos os nós gerados são sempre

armazenados em uma lista, representada pelo atributo *opened_*, a forma de seleção do nó implementada pelo método *select_new_search_node* é a mesma, retornando o primeiro elemento da lista. A forma de criação dos *search_nodes* também é comum. A diferenciação em busca em amplitude e profundidade é realizada pelas classes *breadth_first* e *depth_first*, que implementam o método *process_neighbors_search_nodes* de forma diferente, uma colocando os nós vizinhos ao final da lista de nós abertos e outra colocando no início desta mesma lista.

A classe *guided_search* captura o comportamento comum das buscas guiadas, que está no processo de criação dos *merit_search_nodes*, utilizadas por todas elas. A avaliação de cada nó é fornecida pela classe *evaluated_goal*.

As classes *hill_climbing*, *best_first* e *lrta_star* implementam o comportamento associado a $SelecionaNo$ e $AdicionaNo$ para os algoritmos de subida de encosta, *best first search* e A* com aprendizado em tempo real. Por exemplo, no caso de *best_first*, em *process_neighbors_search_nodes* armazena todos os nós gerados por ordem de mérito em uma estrutura do tipo *heap*, e *select_new_search_node* retorna o primeiro nó desta lista.

Algumas estratégias de busca são implementadas em função de outras. Por exemplo, a subida forçada de encosta [17], apresentada na Seção 6.2.2, utiliza as estratégias de subida de encosta, representada pela classe *hill_climbing_search* e de busca em amplitude, representada pela classe *breadth_first_search*, dependendo das características do espaço de estados. Em sua implementação no PLANSIM estas estratégias foram reutilizadas. Este tipo de reuso das implementações das estratégias básicas de busca é importante no arcabouço pois a maioria das estratégias utilizadas pelos planejadores analisados na Seção 6.1.3 implementa algum tipo de combinação destas estratégias.

### 7.1.3 Simulador

No PLANSIM, o modelo é representado por um simulador a eventos discretos, que deve implementar a interface definida na Figura 7.4. Ao contrário dos resolvedores de propósito geral, o PLANSIM não possui acesso a aspectos internos do modelo simulado. Nos resolvedores de propósito geral, este acesso é importante pois é a partir deles que as funções heurísticas são derivadas (Seção 6.1.2). A forma como o PLANSIM implementa as funções heurísticas específicas para o domínio é mostrada na Seção 7.1.4.

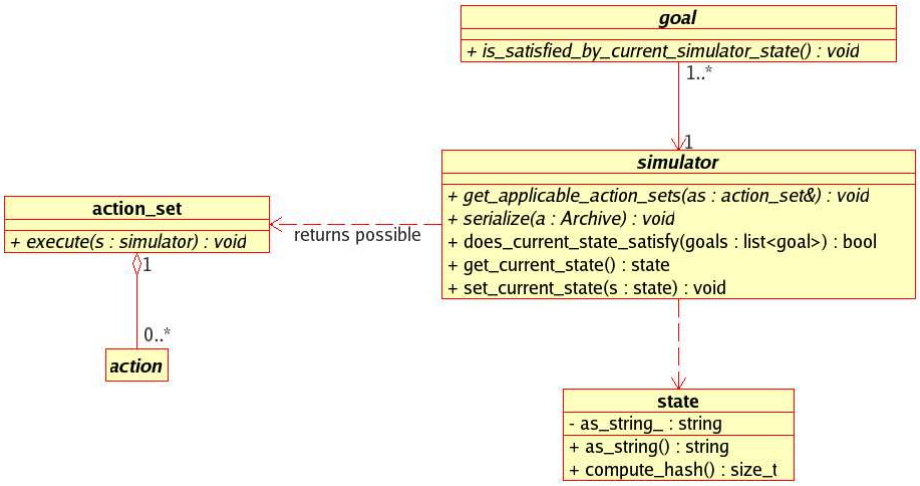


Figura 7.4: Interface para simulador a eventos discretos

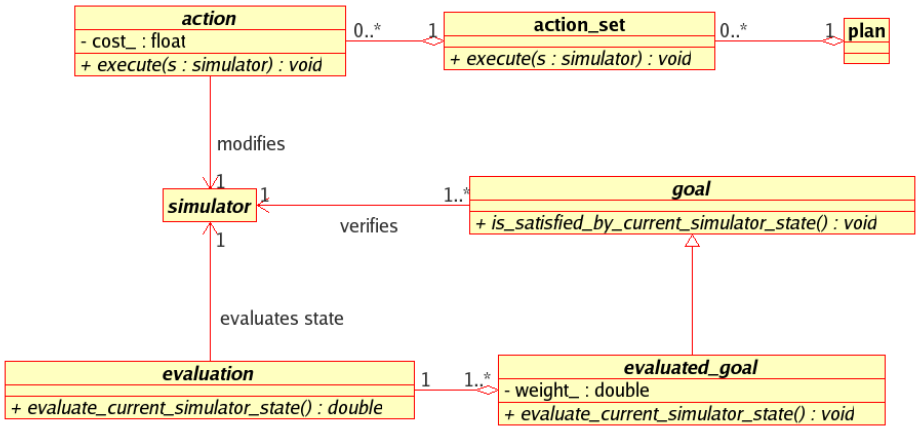


Figura 7.5: Objetivos, planos e funções heurísticas

A interface ao simulador é limitada às funcionalidades de obtenção do estado corrente do simulador, implementada pelo método *get_current_state*, colocação do simulador em um determinado estado, implementada pelo método *set_current_state* e a obtenção de todas as ações possíveis de serem aplicadas no estado corrente, implementada pelo método *get_applicable_action_sets*. A forma como o estado do simulador é modificado depende da ação que está sendo aplicada, e é reponsabilidade da classe *action_set*, detalhada na Seção 7.1.4.

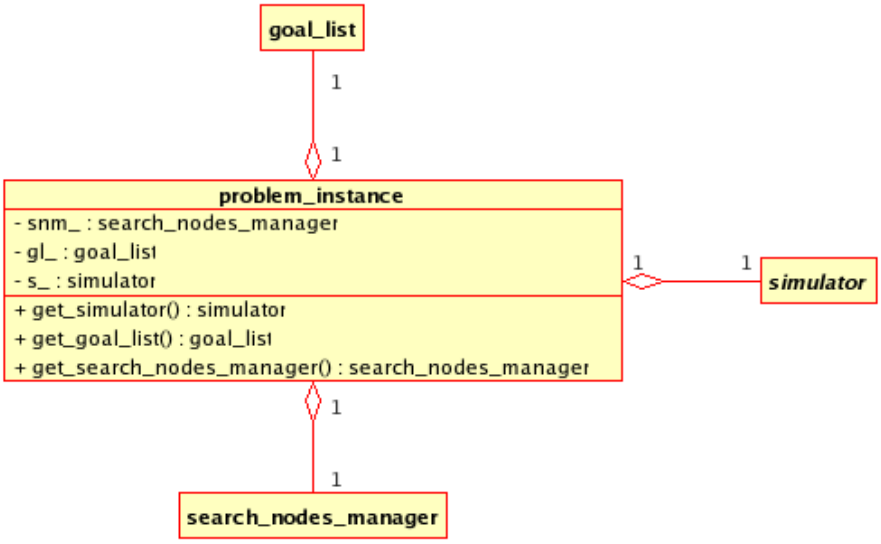


Figura 7.6: Instância de um problema de planejamento



### 7.1.4 Objetivos, planos e funções heurísticas

A Figura 7.5 implementam no PLANSIM os conceitos básicos da área de planejamento clássico. Os objetivos de um determinado problema de planejamento devem ser derivados da classe *goal*. O método *is_satisfied_by_current_simulator_state* verifica se o estado atual do simulador satisfaz o objetivo. Um objetivo é especializado em um *evaluated_goal* se existe uma função heurística, representada pela classe *evaluation*, que avalia o estado do simulador em relação a um objetivo.

O plano, representado pela classe *plan* é basicamente uma lista ordenada de *action_set*. Cada *action_set* por sua vez contém um conjunto de ações, representada pelas classes *action*. A chamada ao método *action_set::execute* tem como conseqüência a aplicação, uma a uma, das ações definidas no conjunto, através da chamada a *action::execute*.

### 7.1.5 Instância de um problema de planejamento

A classe *problem_instance*, mostrada na Figura 7.6, agrega toda a informação necessária pelo mecanismo de busca, *search_algorithm*, para conduzir o processo de solução do problema de planejamento. Nela temos um simulador, um gerenciador de nós visitados e uma lista de objetivos. Esta informação é passada a *search_algorithm::solve* para a solução do problema.

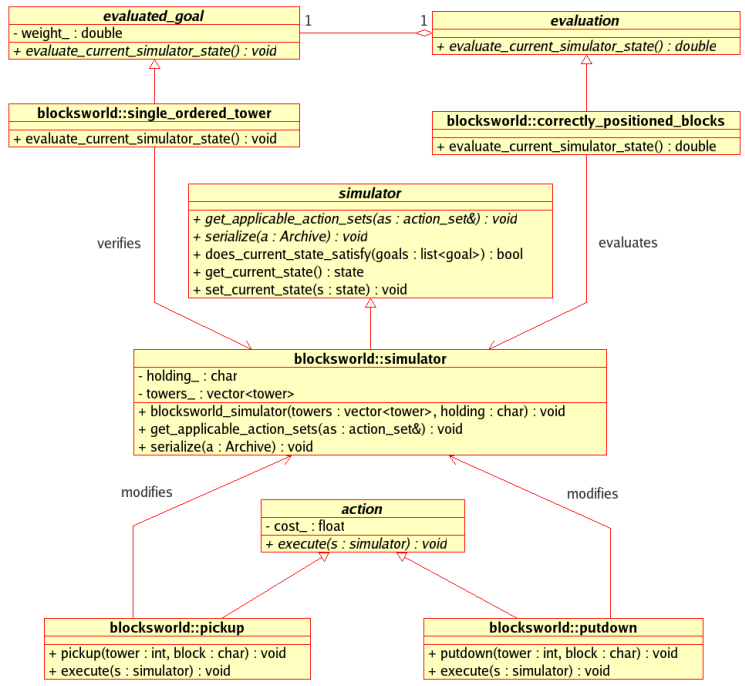


Figura 7.7: Instanciação do PLANSIM para solução do Blocksworld

## 7.2 Exemplo de instanciação

Esta Seção apresenta um exemplo de instanciação do PLANSIM para a solução de um problema clássico de planejamento e busca. Instanciações para a resolução dos problemas de Torres de Hanoi e *Tiles puzzle* podem ser obtidas no sítio do PLANSIM na Web. Uma instanciação para a construção de um planejador para o transporte de oleodutos é mostrada no capítulo 8.

### 7.2.1 Blocksworld

No domínio blocksworld [22], temos $n$ blocos inicialmente dispostos em $m$ pilhas. Cada bloco é associado a uma letra, e o objetivo é colocar todos os blocos em uma única pilha, de uma forma ordenada. Para movimentar os blocos temos um braço mecânico que pode segurar um único bloco. Apenas blocos que estão no topo das pilhas podem ser movimentados.

A Figura 7.7 mostra como o PLANSIM pode ser instanciado para resolver este problema. Apenas cinco classes são necessárias. A classe *blocksworld_simulator* define como o estado do sistema é representado e serializado. Em nosso caso optamos por representar o estado por um vetor de pilhas, cada pilha contendo uma lista de caracteres representando os

blocos. O bloco que está sendo segurado pelo braço mecânico é representado pelo atributo *holding_*. Durante a serialização armazenamos tanto o vetor de pilhas quanto o bloco que está sendo segurado no momento. Note que não é preciso serializar todos os membros do simulador, apenas aqueles que podem ser modificados durante a aplicações das ações definidas para o problema. Isto pode reduzir bastante a necessidade de memória durante a execução da busca.

Temos duas ações definidas, denominadas *pickup* para pegar um bloco do topo de uma pilha, e *putdown* para colocar o bloco no topo de uma pilha ou sobre a mesa. A geração das ações possíveis de serem aplicadas em um determinado estado, realizada pelo método *get_applicable_action_sets*, é tarefas simples para o simulador: se o braço está segurando algum bloco, temos uma ação *putdown* deste bloco para cada pilha. Caso contrário, temos uma ação *pickup* para cada pilha existente no estado.

Um procedimento muito parecido pode ser utilizado para facilmente gerar as ações possíveis em instâncias do Pipesworld, sem os problemas descritos na Seção 3.3.1.

O objetivo é representado por uma classe *single_ordered_tower*, que retorna verdade quando o estado do simulador contém uma pilha com todos os blocos ordenados. Definimos também uma função heurística simples através da classe *correctly_positioned_blocks*, que examina as pilhas e conta o número de blocos que estão posicionados corretamente em cada pilha. Ela retorna o número total de blocos menos o maior número de blocos já corretamente posicionados.