3
MÉTODOS DE ACELERAÇÃO DO MODELO RL-NFHP

Este capítulo apresenta as alterações introduzidas no modelo de *Reinforcement Learning Neuro-Fuzzy Hierarchical Politree* (RL-NFHP), proposto por Figueiredo (2003), com o intuito de melhorar e acelerar o processo de aprendizado. Foram inseridas seis diferentes políticas de seleção da ação a ser executada pelo agente, sendo uma delas proposta no âmbito deste trabalho (*Q-DC-roulette*). Além disso, as seguintes alterações foram realizadas: implementação do método *early stopping* para a determinação do fim do processo de treinamento; desenvolvimento do método *eligibility trace* cumulativo; e implementação de um método de poda da estrutura, de forma a eliminar células desnecessárias. Ainda com a intenção de acelerar o processo de aprendizado, o código computacional do modelo foi inteiramente reescrito, com uma estrutura mais otimizada.

3.1
Políticas de seleção de ação

De forma a acelerar o processo de aprendizado do modelo RL-NFHP, seis diferentes políticas de seleção de ação foram implementadas e testadas (subseção 4.1.1): *ε-greedy*, *Q-roulette*, *DC-roulette*, *Q+DC-roulette* e *Q-DC-roulette*. Estas políticas de seleção são explicadas nas subseções a seguir.

3.1.1
Política *ε-greedy*

A política *ε-greedy* (Sutton & Barto, 1998) seleciona a ação associada à maior função de valor *Q* com probabilidade p = 1-ε, também chamada de seleção gulosa; e seleciona aleatoriamente uma ação qualquer com probabilidade p = ε. Esta política foi utilizada no algoritmo original de Figueiredo (2003).

O parâmetro ε da política de escolha de ação está diretamente relacionado com a exploração – *exploration* –. Valores elevados deste parâmetro geram

maiores oscilação dos valores de *Q*, pois realizam escolhas de ações mais ousadas. Valores baixos de ε levam a uma probabilidade de escolha maior das ações consideradas melhores, ou seja, usufrui-se mais do conhecimento aprendido – *exploitation* –. Isto pode acarretar o aprendizado de ações subótimas, gerando desempenho insatisfatório.

Uma fraqueza do método *ε-greedy*, quanto a explorar e usufruir, é o fato de a escolha sobre as ações ser uniforme. Ou seja, tem-se a mesma probabilidade de escolher uma ação ruim e uma quase ótima. Quando as piores ações são muito ruins, isto pode ser indesejado.

3.1.2
Política *Q-roulette*

Na política *Q-roulette*, com um número de vezes proporcional a (1-ε), a seleção da ação é realizada através de uma roleta de *Q(s,a)* baseada na distribuição de probabilidade dada pelas funções de valores *Q* (eq. 3.1); e com probabilidade p=*ε*, a seleção é feita de maneira aleatória. Nesta roleta, a ação gulosa continua com a maior probabilidade de seleção. Esta forma de seleção é mais conservativa, uma vez que as ações que apresentarem as maiores funções de valores *Q* terão maiores chances de serem escolhidas. Este tipo de método de seleção de ações é conhecido como *softmax* (Sutton & Barto, 1998).

$$P(a_i \mid s) = \frac{Q(s,a_i)}{\sum_k Q(s,a_k)} \qquad (3.1)$$

onde: $P(a_i|s)$ é a probabilidade de se escolher a ação a_i, dado que o agente está no estado *s*, $Q(s,a_i)$ é a função de valor da ação a_i e $\sum Q(s,a_k)$ é o somatório das funções de valores *Q* relativas às ações disponíveis para o agente quando o mesmo se encontra no estado *s*.

Outra versão da política *Q-roulette* baseada em potência, chamada de método de Gibbs-Boltzman (Sutton & Barto, 1998), pode ser empregada quando as funções valor *Q* possuem valores negativos ou nulos (eq. 3.2).

$$P(a_i \mid s) = \frac{b^{\frac{Q(s,a_i)}{\tau}}}{\sum_k b^{\frac{Q(s,a_k)}{\tau}}} \tag{3.2}$$

onde os parâmetros da base b (por exemplo $b=e$) e τ (por exemplo τ =100) devem ser ajustados. Para valores de τ elevados, a escolha das ações passa a ser equiprovável.

Quando a função de valor Q não assume valores negativos nem nulos, deve-se dar preferência à versão mais simples (eq. 3.1) com menor custo computacional. Neste trabalho, como a função Q foi inicializada com valores não nulos próximos a zero e o reforço só possui valores positivos, a versão mais simples da roleta Q foi adotada nos testes.

Não existe um consenso se o uso da seleção de ação baseada em *softmax* é melhor ou pior do que *ε-greedy*, sendo que o desempenho final depende da tarefa a ser executada pelo agente (Sutton & Barto, 1998).

3.1.3
Política *DC-roulette*

Na política *DC-roulette* (*Dual Counter Roulette*) proposta por Flesch (2009), a escolha da ação é associada ao maior *Q(s,a)* com probabilidade p = 1-ε, ou seja, uma seleção gulosa; e com probabilidade p = ε, esta seleção é baseada no número de visitas de cada ação (eq. 3.3), ou seja, uma roleta baseada em visita semelhante a *Q-roulette*.

$$P(a_i \mid s) = \frac{1 - \frac{C(s,a_i)}{\sum C(s,a_k)}}{N_a - 1} \tag{3.3}$$

onde: $P(a_i|s)$ é a probabilidade de se escolher a ação a_i dado que o agente está no estado s; $C(s,a_i)$ é o número de vezes que determinada ação foi escolhida (visitada) em ciclos anteriores; $\sum C(s,a_k)$ é o número de vezes que o agente passou pelo estado s, ou, de outra forma, o somatório das visitas de todas as ações escolhidas; e N_a o número de ações possíveis.

Observa-se que ações pouco visitadas possuem maior probabilidade de seleção, aumentando a exploração. Deve também ser observado que, neste caso, a roleta não leva em consideração as funções de valores *Q*.

3.1.3 Política *Q+DC-roulette*

A política *Q+DC-roulette* agrega as políticas *Q-roulette* e *DC-roulette*, por meio de suas roletas. A ação é escolhida através da *Q-roulette* com probabilidade p = 1-ε; e com probabilidade p = ε a ação é selecionada por *DC-roulette*.

Trata-se de uma política de escolha com extrema exploração, de um lado em relação ao valor *Q* – *Q-roulette* – e do outro em relação à visita – *DC-roulette* –. Em nenhum momento se escolhe diretamente a ação com maior função valor *Q*, ou seja, escolha gulosa. A ação gulosa não necessariamente possui a maior probabilidade de seleção, este fato pode gerar grandes oscilações da função valor *Q*, pois em um mesmo estado tem-se uma alta probabilidade de escolha de diferentes ações.

3.1.5 Política *Q-DC-roulette*

A política *Q-DC-roulette* é proposta neste trabalho como uma inovação, a qual visa mesclar os benefícios do aprendizado por meio das funções de valor *Q* com a possibilidade de exploração de acordo com o quanto cada ação foi visitada. Trata-se de uma mistura entre a *Dual Counter Roulette* e a *Q-roulette*. A seleção de ação com probabilidade p = ε é baseada numa distribuição de probabilidade conforme a eq. 3.4; e a ação é selecionada de forma gulosa no restante das vezes (p = 1-ε).

$$P(a_i \mid s) = \frac{\left(1 - \frac{\varphi.C(s,a_i)}{\sum C(s,a_k)}\right) Q(s,a_i)}{\sum \left(1 - \frac{\varphi.C(s,a_k)}{\sum C(s,a_k)}\right) Q(s,a_k)} \tag{3.4}$$

onde $P(a_i|s)$ é a probabilidade de se escolher a ação a_i dado que o agente está no estado s; $C(s,a_i)$ é o número de vezes que determinada ação foi escolhida (visitada) em ciclos anteriores; φ é uma constante de ponderação (por exemplo quando φ=1 não existe ponderação); $\sum C(s,a_k)$ é o número de vezes que o agente passou pelo estado s; e $Q(s,a_i)$ é a função de valor da ação a_i.

A ideia é realizar uma exploração mais inteligente, dando menos importância à seleção de ações que tenham função valor Q alta mas que foram pouco visitadas, ou a ações muito escolhidas mas com baixo Q.

Vale ressaltar, para evitar dúvidas de interpretação, a forma como foi feita a identificação entre as distintas políticas de seleção de ação *Q+DC-roulette* e *Q-DC-roulette*, cuja diferenciação reside apenas nos sinais de '+' e '-'.

A figura 3.1 mostra uma comparação entre as probabilidades de escolha de ação para as diferentes políticas de seleção, usando um exemplo de quatro ações a_i possíveis de uma determinada polipartição para um dado estado s com os valores de $Q(s,a_i)$ e a quantidade de visitas $C(s,a_i)$ expostos da tabela 3.1.

Tabela 3.1: Valores de $Q(s,a_i)$ e a quantidade de visitas $C(s,a_i)$ para determinada polipartição em um estado s.

Ação	a_1	a_2	a_3	a_4
Valor Q(s,a_i)	100	50	10	10
Visita C(s,a_i)	800	200	10	200

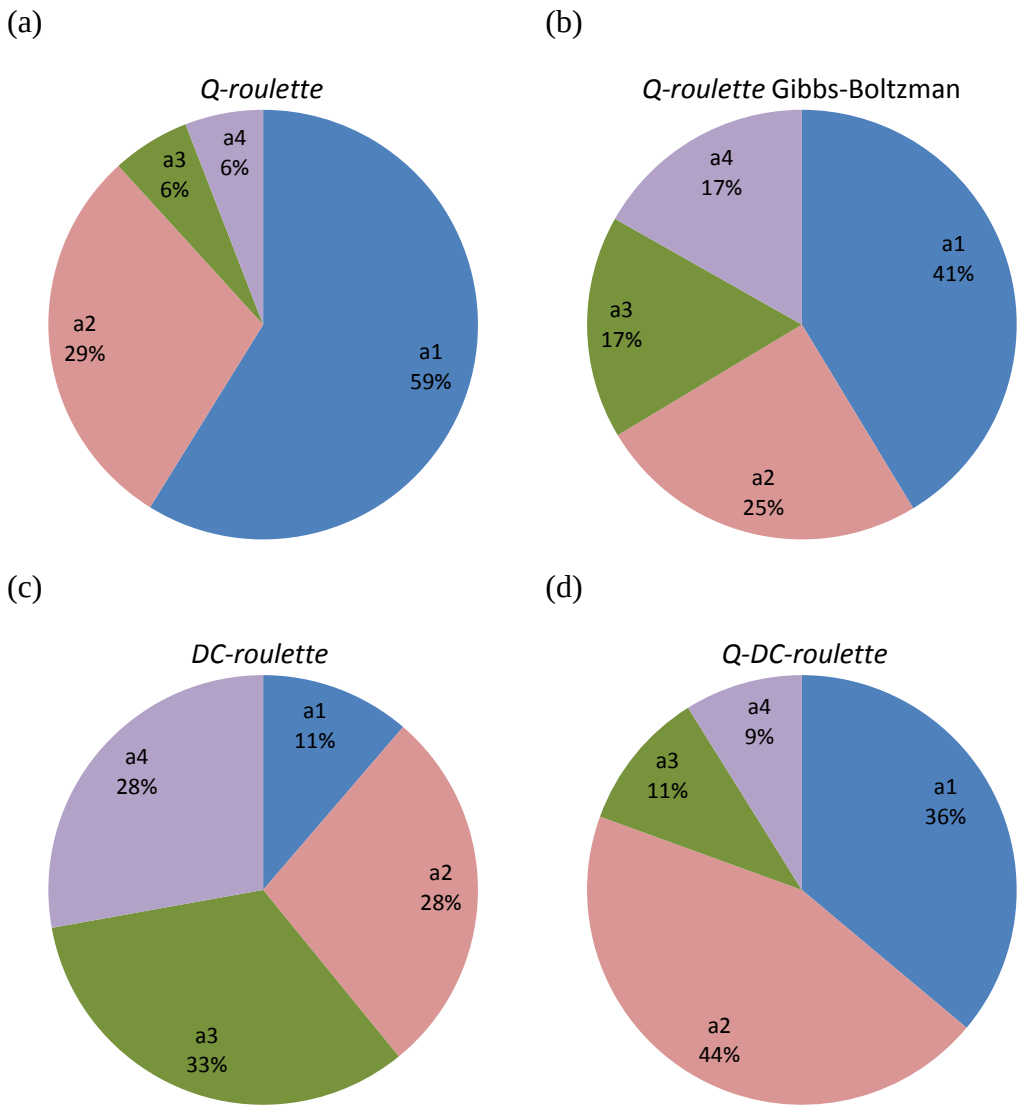

Figura 3.1: Probabilidade de seleção das ações da tabela 3.1 de acordo com as políticas: (a) *Q-roulette*; (b) *Q-roulette* Giggs-Boltzman; (c) *DC-roulette* e (d) *Q-DC-roulette*.

Verifica-se, pela figura 3.1, que a maior probabilidade de escolha da ação gulosa a_1 ocorre para as políticas *Q-roulette* e sua variação *Q-roulette* Giggs-Boltzman, a da ação a_2 ocorre na política *Q-DC-roulette*, enquanto que a maior chance de escolha da ação com menor visitação a_3 acontece na *DC-roulette*. A política *Q-roulette* Giggs-Boltzman tende a atenuar a diferença entre as probabilidades de escolha das ações em relação a *Q-roulette*. Na política *Q+DC-roulette* a ação é escolhida através da *Q-roulette* ($p = 1-\varepsilon$) ou da *DC-roulette* ($p = \varepsilon$).

3.2 *Early stopping*

O fim do aprendizado, ou treinamento, ocorre quando a estrutura RL-NFHP deixa de mudar significativamente, ou seja, o agente aprende. Na prática, é difícil determinar o fim do treinamento. Pode-se treinar demasiadamente a estrutura, perdendo-se em termos de desempenho de generalização e em tempo de processamento, ou, ao contrário, treiná-la insuficientemente, impossibilitando o agente de cumprir seu objetivo ou de cumpri-lo de maneira satisfatória.

No modelo RL-NFHP original, o número de episódios necessários para o aprendizado era avaliado empiricamente (Figueiredo, 2003).

Baseado no conceito de *early stopping* praticado em Redes Neurais (Haykin, 1998), foi desenvolvido neste trabalho um procedimento de parada automática do aprendizado, que avalia a convergência do algoritmo. O treinamento é interrompido quando, durante o teste, no procedimento de validação, o número médio de passos em cada episódio aumenta e é menor que o número de passos máximo estipulado empiricamente. Este processo é semelhante ao aplicado a Redes Neurais, onde o treinamento é interrompido quando o erro de validação aumenta e é menor que um limiar definido. Esta validação é executada a cada N episódios.

3.3 *Elegibility trace* cumulativo

Foi implementado um método de rastros de eligibilidade (*eligibility trace*) cumulativo, onde o eco no passado faz com que a ação *n* passos a frente influencie os passos antigos. Este método de diferença temporal TD(λ) (subseção A.5.3) passa a atualizar os valores de *Q* de *n* estados passados.

Na prática, para facilitar a implementação computacional, o *eligibility trace* $e_t(s_t,a_t)$ foi ligeiramente modificado. Os pares estado-ação visitados são armazenados ordenadamente em um vetor de tamanho n+1, onde a última posição refere-se ao estado t+1. Este *eligibility* cumulativo desconsidera o fato que o estado anterior possa ter sido particionado e que a ação possa ter sido escolhida de maneira aleatória.

A função de valor *Q* é atualizada de maneira similar à subseção 2.5.6 conforme as equações 3.5 e 3.6:

- Se $R_{t+1} > R_t$:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\, e_t(s_t, a_t)\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{3.5}$$

- Se $R_t \geq R_{t+1}$:

$$Q(s_t, a_t) = (1 - fp)\; e_t(s_t, a_t)\; Q(s_t, a_t) \tag{3.6}$$

onde: o valor $Q(s_t, a_t)$ é atualizado a partir do seu valor atual; r_{t+1} é o reforço local imediato (este é o reforço da partição definido na subseção 2.5.5); o γ é um parâmetro que fixa um percentual da contribuição da função de valor *Q* associada à próxima ação a_{t+1} escolhida (termo $Q(s_{t+1}, a_{t+1})$) quando o sistema está no estado s_{t+1}; α é o parâmetro proporcional à contribuição relativa desta ação local na ação global (subseção 2.5.6); e $e_t(s_t, a_t)$ está associado à posição do vetor dos pares estado-ação.

O valor de $e_t(s_t, a_t)$, associado à posição do vetor, é '1' em $i=n$ e decresce para estados anteriores conforme a eq. 3.7.

$$e_t(s_t, a_t) = e^{\lambda(i-n)} \tag{3.7}$$

onde λ indica o fator de influência dos estados anteriores; $i \in [1, n]$ é a posição do vetor; e *n* o número de estados passados.

Este tipo de atualização, que considera diversos estados passados, requer maior processamento computacional, porém a velocidade de convergência é, em geral, maior. Existem algumas restrições ao uso de *eligibility trace* como, por exemplo, quando o algoritmo utiliza excessivamente políticas de escolha de ações *non-greedy* (Sun & Sessions, 2000).

3.4
Poda da estrutura RL-NFHP

Em alguns problemas, durante o processo de aprendizado, a estrutura RL-NFHP pode crescer demasiadamente, ficando com um número de células indesejado. Este crescimento, embora atenda aos critérios de particionamento apresentados na subseção 2.5.7, pode gerar pares estado-ação pouco visitados; não permitir que o modelo aprenda; e/ou aumentar o custo computacional.

Para tentar contornar este e outros problemas, foi desenvolvido um método de poda da estrutura. Ele consiste em remover células filhas, ou folhas, criadas e não visitadas.

A identificação destas células filhas espúrias ocorre por meio de um conceito de tempo de vida. É permitido a todas as células criadas um tempo de vida, em passos de iteração, durante o qual a célula deve receber pelo menos uma visita. Terminado este tempo, caso esta folha não tenha sido visitada, ela é eliminada, ou podada, da estrutura, voltando a polipartição pai a ser o que era antes do particionamento.

A verificação do tempo de vida das células filhas criadas é realizada ao final dos episódios.

3.5
Reescrita do algoritmo

O código computacional do algoritmo original foi inteiramente reescrito, melhorando estruturas cíclicas e recursivas. A concepção da organização dos objetos na linguagem Java foi repensada, generalizando o modelo e facilitando futuras modificações conforme mostrado na tabela 3.2. A figura 3.2 ilustra a estrutura em árvore representativa do grau de hierarquia entre as principais classes do modelo RL-NFHP da tabela 3.2. As setas indicam a subordinação entre elas, ou seja, as classes que estão inseridas no código das de maior hierarquia.

Para a escrita do programa em Java deste trabalho, utilizou-se o ambiente de desenvolvimento integrado NetBeans® versão 6.5.1.

Tabela 3.2: Arquivos do modelo RL-NFHP.

Arquivo	Descrição
Action.java	Classe contendo o vetor de comandos dos atuadores.
ActiveCell.java	Classe que armazena cada célula ativa no ciclo com informações sobre grau de ativação das partições e reforço.
Cell.java	Classe célula da estrutura RLNFHP.
Collision.java	Classe com o ponto de colisão do robô Khepera e o obstáculo.
Controller.java	Interface com métodos obrigatórios de um controlador.
Display.java	Classe relacionada com a visualização da simulação.
Domain.java	Classe com os domínios das entradas utilizada em cada célula do modelo RL-NFHP.
Drawing.java	Classe relacionada com a visualização da simulação.
KeyboardInput.java	Classe para captura de informações pelo teclado.
Line.java	Classe que descreve uma linha.
Log.java	Classe contendo informações relevantes relacionada com a visualização da simulação.
Main.java	Programa principal.
ManualController.java	Classe contendo um controlador manual.
Obstacle.java	Classe que descreve um obstáculo.
RLNFHP.java	Classe com o modelo RL-NFHP.
Simulation.java	Interface com métodos obrigatórios de um simulador.
SimulationKhepera.java	Classe com o simulador do robô Khepera.
SimulationMountain.java	Classe com o simulador carro na montanha.
ValueFunctionQ.java	Classe da função valor Q para cada ação dentro da partição. Armazena informações estatísticas da função valor Q e das visitas.
Interface5Entries.vi	Programa de interface entre computador e robô real.
entryKhepera.txt	Arquivo com os parâmetros de entrada para a simulação do robô Khepera.
entryMountain.txt	Arquivo com os parâmetros de entrada para a simulação do carro na montanha.
fag.txt	Arquivo de permissão de leitura e escrita dos arquivos arquivo_sensores.txt e arquivo_comandos.txt.
arquivo_sensores.txt	Arquivo com os valores de leitura dos sensores do robô.
arquivo_comandos.txt	Arquivo com os comandos para os atuadores dados pelo controlador.

A implementação consiste em um programa principal (Main.java) responsável pelas estruturas cíclicas dos passos em cada episódio e da quantidade de episódios durante o treinamento do modelo RL-NFHP modificado (RLNFHP.java), assim como pelo armazenamento das informações relevantes para posterior análise. Este programa inicializa um controlador (no caso, RLNFHP.java) e uma simulação benchmark (no caso, SimulationKhepera.java ou SimulationMountain.java) com os parâmetros fornecidos pelos arquivos de entrada (entryKhepera.txt ou entryMountain.txt, respectivamente). Ele intermedia a passagem da leitura dos sensores da simulação para o controlador e o comando dos atuadores do controlador para a simulação durante os passos. Este programa

principal também é responsável por obter o melhor controlador durante a fase de aprendizado, quando são executados diversos treinamentos.

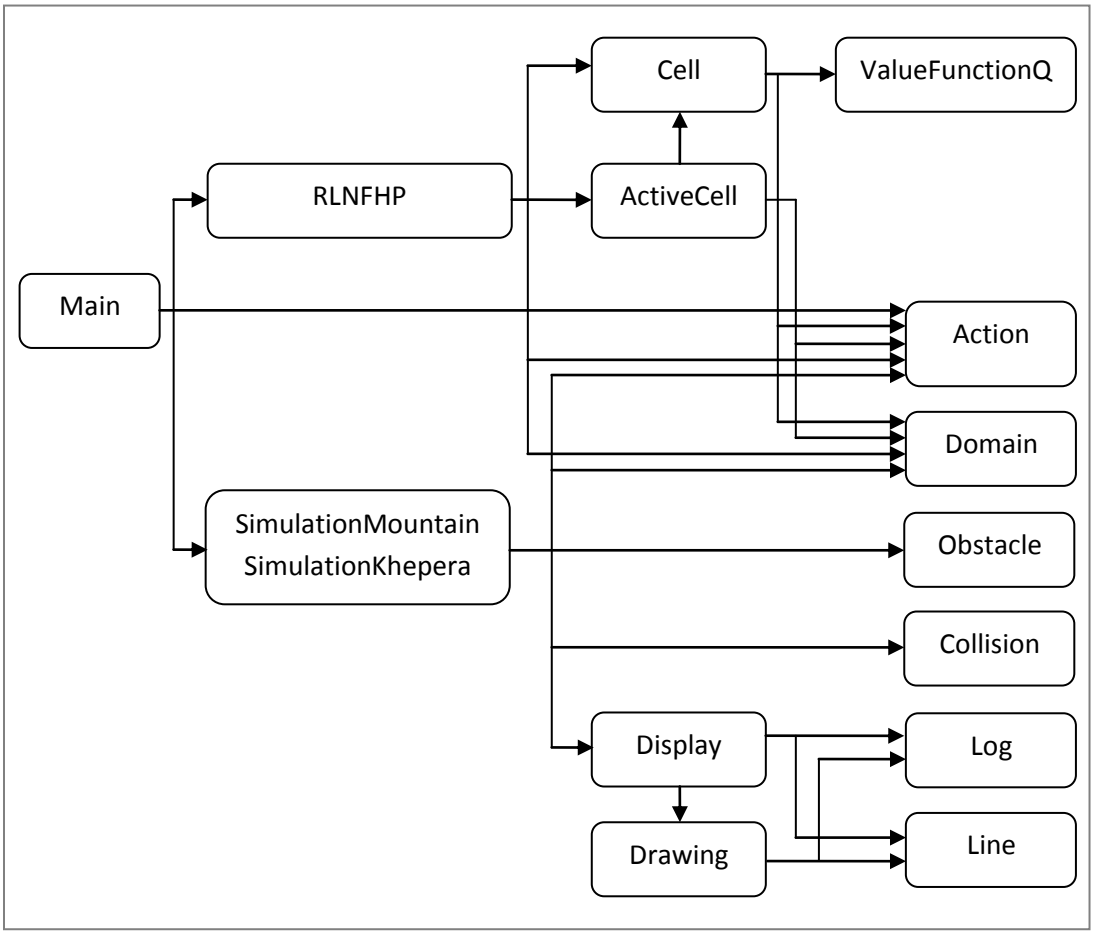

Figura 3.2: Hierarquia entre as classes do modelo RL-NFHP

O controlador utilizando o modelo RL-NFHP modificado (RLNFHP.java) é genérico, podendo receber problemas de uma a seis entradas e dar instruções, ou comandos, para qualquer número de atuadores. Para isto foram desenvolvidas classes que abstraem o número de entradas (Domain.java) e o número de atuadores (Action.java). A estrutura RL-NFHP está armazenada sob a forma de um vetor de células (Cell.java). Dentro de cada polipartição de cada célula está especificado seu subdomínio (Domain.java), os valores *Q* (ValueFunctionQ.java) para cada ação possível e o índice do filho (caso esta polipartição tenha sido particionada). O controlador pode ser invocado pelo programa principal a cada passo de duas formas: treino e teste. No treinamento da estrutura RL-NFHP, o controlador é responsável por: determinar quais as células ativas (ActiveCell.java) para um dado vetor de entradas; escolher as ações de cada polipartição final de

acordo com a política estabelecida (seção 3.1); calcular a ação resultante (Action.java) mediante os pesos das funções de pertinência dos conjuntos fuzzy (subseção 2.5.3); retropropagar o reforço (subseção 2.5.5) – informação recebida juntamente com o vetor de estados – atualizando os valores *Q* das células ativas do passo anterior (subseção 2.5.6); e particionar polipartições que atendam aos critérios impostos (subseção 2.5.7). Durante a fase de testes, o controlador deve somente determinar as células ativas e calcular a ação resultante, a qual é ponderada pelos conjuntos fuzzy das ações com maiores valores *Q*. Vale ressaltar que o controlador também é o responsável por fornecer ao programa principal todas as informações relevantes que devem ser persistidas em arquivo.

O módulo do simulador é específico ao problema em questão (seções 4.1 e 4.2). Ele deve ser capaz de reproduzir a dinâmica do problema, através de equações, fornecendo as entradas, obtida pelos sensores, e o reforço para uma dada execução dos comandos (Action.java) impostos pelo controlador. No caso do Khepera (seção 4.2), o simulador deve testar colisões (Collision.java) entre o robô e os obstáculos do ambiente (Obstacle.java) e tratá-las, além de fornecer dados que emulem as leituras dos sensores. Cabe também ao simulador gerar informações gráficas (por exemplo, figuras 4.2, 4.8 e 4.9) da simulação em curso (Display.java).

Qualquer outro controlador ou simulador pode ser escrito sem alteração do restante do código, desde que implemente as interfaces de controle (Controller.java) e de simulação (Simulaton.java), respectivamente.

Por não terem sido utilizadas diretamente, diversas outras classes que foram implementadas não estão descritas neste trabalho.