Rafael Ribeiro de Carvalho

Robótica baseada em comportamento em Futebol de Robôs

Juiz de Fora
12/04/2013

Rafael Ribeiro de Carvalho

Robótica baseada em comportamento em Futebol de Robôs

Orientador: Raul Fonseca Neto

Universidade Federal de Juiz de Fora

Juiz de Fora
12/04/2013

Rafael Ribeiro de Carvalho

Robótica baseada em comportamento em Futebol de Robôs

Monografia submetida ao corpo docente do Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como parte integrante dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação. Juiz de Fora, 12 de abril de 2013:

Raul Fonseca Neto - Orientador
Doutor em Engenharia de Sistemas e Computação, UFRJ, 1990

Carlos Cristiano Hasenclever Borges
Doutor em Engenharia Civil, UFRJ, 1999

Custódio Gouvêa Lopes da Motta
Doutor em Engenharia Civil, UFRJ,2010

Juiz de Fora
12/04/2013

Resumo

Diante do problema de se criar robôs autônomos para execução de tarefas e solução de problemas, várias arquiteturas e estratégias foram propostas. Dentre essas abordagens, destaca-se atualmente a robótica baseada em comportamento, metodologia derivada da arquitetura *Subsumption*. Esse trabalho descreve brevemente s arquitetura *Subsumption* e anteriores, apresenta a robótica baseada em comportamento para então elaborar uma implementação na linguagem C a partir dessa metodologia para o Futebol de Robôs, um problema que apresenta alto dinamismo e complexidade.

Palavras-chaves: robótica, robótica baseada em comportamento, futebol de robôs.

Abstract

With the challenge of creating autonomous robots capable of task execution and problem solving, many strategies and architectures were proposed. Among those approachs, behavior-based robotics are the most used in recent years, derivated from Subsumption architecture. This work briefly decribes Subsumption architecture and its precursors, introduce behavior based robotics to then elaborate an implementation on C, based on this methodology, applied on Robot Soccer, a problem with high dynamism and complexity.

Keywords: robotics, behavior based robotics, robot soccer.

Sumário

1 Introdução

O estudo e desenvolvimento de sistemas autônomos, isto é, que não necessitam de intervenções humanas durante seu funcionamento (JONES, 2004), é um dos objetivos da área de Inteligência Artificial. Suas aplicações vão de cirurgias não assistidas por humanos a atividades nas quais a presença humana é inviável (robôs para estudo e exploração de outros planetas) ou demasiada arriscada (resgate de acidentados de locais perigosos) entre outras.

Como o desenvolvimento de um robô depende intrinsecamente do problema a ser resolvido, casos que envolvam aspectos mais abrangentes e desafiadores darão origem a modelos mais robustos, com potencial para serem aplicados em outras áreas. Sob esse ponto de vista, o futebol de robôs ganha cada vez mais espaço como área de pesquisa para a aplicação de sistemas autônomos, tanto pelo conjunto de características que o tornam um desafio complexo (alto dinamismo do ambiente, necessidade do uso de diversas áreas de Inteligência Artificial - Teoria de Jogos, Otimização, Heurísticas - bem como da Ciência da Computação - Processamento de Sinais, Computação Gráfica, Sistemas Distribuídos, etc.).

Várias arquiteturas já foram propostas para esse tipo de sistema, como *Sense-Act-Plan*, Controle Reativo e *Subsumption*. A metodologia da robótica baseada em comportamento, baseada na arquitetura *Subsumption*, se baseia na decomposição da estratégia de solução do robô em comportamentos primitivos e simples. A emersão de comportamentos mais complexos a partir destes primitivos simplifica o processo de criação da solução.

O objetivo desse trabalho é implementar, na linguagem C - a partir do uso de DLL's -, os comportamentos primitivos para futebol de robôs aplicando os conceitos de Robótica baseada em comportamento, capaz de realizar tarefas como levar a bola ao gol adversário ou impedir que a bola seja levada ao próprio gol. Usando a Robótica baseada em comportamento, deseja-se explicar cada um dos comportamentos básicos que o robô possuirá, bem como os possíveis comportamentos avançados que surgem da interação entre os mais simples. O teste desse sistema será feito a partir do simulador *Middle-League Simurosot*, em que as tarefas de sensoriamento e atuação já são dadas pelo próprio simulador, reduzindo o trabalho do programador à implementação da estratégia de jogo propriamente dita.

O trabalho foi organizado em mais cinco capítulos. O próximo trará de definições e conceitos usuais à robótica, além de um breve histórico das estratégias de inteligência artificial aplicadas a robôs. No capítulo seguinte será apresentada a robótica baseada em comportamento, com suas definições, princípios e exemplos em robôs de aplicações diversas. Na

sequência haverá o capítulo sobre a estratégia desenvolvida para futebol de robôs, mostrando a aplicabilidade da teoria para este problema, seguido pela análise dos resultados obtidos em um outro capítulo. Por fim, o capítulo de conclusão, avaliando a eficiência da robótica baseada em comportamento para futebol de robôs, tanto visando a implementação como o resultado final.

2 Conceitos e Histórico

2.1 Conceitos

Alguns conceitos e termos familiares àqueles que trabalham com robótica serão usados ao longo do trabalho e, portanto, serão exibidos a seguir.

2.1.1 Sensoriamento

Sensoriamento é a ação de obter informação do ambiente que cerca o agente, transformando a observação de uma grandeza física em um sinal elétrico proporcional(JONES, 2004). O robô pode, então, usar grandezas digitais para analisar o ambiente que o cerca. Algumas das grandezas de interesse para medição são intensidade de uma onda eletromagnética, como luz visível e infravermelho, volume de um sinal sonoro e força recebida.
A maior parte dos sensores está sujeita a alguns tipos de erros, como:

- imprecisão, quando a proporção entre a grandeza física e o sinal resultante não é constante ao longo do tempo;
- falso negativo, em que uma grandeza física deveria ser notada pelo sensor mas, por alguma falha do mesmo, este envia um sinal de que não há detecção da grandeza em questão; e
- falso positivo, analogamente ao anterior, quando o sinal diz existir alguma grandeza física que, na verdade, não está presente no momento.

Portanto, quando se trabalha com robôs, deve-se levar em conta que as informações obtidas pelo sistema de sensoriamento nem sempre pode condizer com a verdade e, sempre que possível, deve-se usar de informações redundantes ou outros recursos capazes de evitar que esse tipo de falha comprometa ou impeça o agente de realizar sua tarefa.

Alguns exemplos de sensores são: dispositivos infravermelhos, sonares (normalmente utilizados para informar a distância de objetos ou, pelo menos, a presença de algum deles a uma dada distância fixa), câmeras, termômetros.

2.1.2 Atuação

Atuação é a transformação de um sinal elétrico em uma grandeza física, compreendendo motores, alto-falantes, dispositivos luminosos, entre outros. Através de seus atuadores que o robô é capaz de realizar mudanças em seu estado, se movendo ao aplicar torque em suas rodas, e alterar o ambiente externo, usando um braço mecânico para mover objetos.

Analogamente ao item anterior, um atuador real (não ideal) pode não transformar um sinal em uma mesma grandeza física sempre. Ou talvez ainda a transformação seja feita da forma esperada mas o resultado obtido não (por exemplo, para um mesmo sinal elétrico para um motor que gira uma roda pode resultar na locomoção do robô ou não, dependendo da superfície em que se encontra).

2.1.3 Estado

Um estado do ambiente é a coleção de informações que o descreve. Um dado obtido através dos sensores não é, por si só, uma informação, mas o conjunto de dados obtidos através do sensoriamento carrega as informações sobre o ambiente. A análise desses é que gera a informação necessária para o robô definir o estado atual em que se encontra.

O estado objetivo é, portanto, o conjunto de informações que descreve o ambiente quando o mesmo está de acordo com a meta atual do agente.

O espaço de busca compreende, então, todos os possíveis estados que o ambiente pode assumir, podendo chegar a um número incomputável ou mesmo infinito de estados dependendo da natureza do problema.

Na prática, é um problema incomputável calcular todos os possíveis estados de um sistema real. Para tratar esses casos, abstrações são necessárias. Como exemplo, a medida contínua da distância entre dois objetos pode ser transformada para dois estados, “perto” e “longe”, criando-se uma função limiar que determina qual a distância limite entre essas duas classes.

2.1.4 Tarefa

Entende-se por tarefa como o conjunto de operações necessárias para transformar o estado atual do sistema no estado objetivo. Contudo, o estado objetivo não necessariamente é a solução final. Pode-se ter um estado objetivo como um passo para se chegar à solução final.

2.1.5 Robô

A definição de robô dada por Jones (2004) é: um dispositivo capaz de conectar sensoriamento a atuação de forma inteligente. Assim, um robô deve ser capaz de perceber o ambiente que o cerca e, portanto, dispositivos que armazenam informações apenas sobre seu estado

interno não são inclusos na definição. Robôs ainda devem ser capazes de interagir com o meio, excluindo sistemas que coletam informações externas mas que nada fazem para atingir um estado objetivo através de ações que possam mudar esse meio. E ainda devem conectar esses dois sistemas com um mínimo de complexidade.

Embora sem um rigor formal propriamente dito e não sendo aceita universalmente, esta é uma definição atendida por e coerente com os agentes corriqueiramente denominados robôs.

2.2 Histórico das Arquiteturas

Desde o nascimento da robótica, a idéia de robôs autônomos fascina pesquisadores. Essencialmente um robô autônomo seria capaz de gerar planos a partir de dados sensoriais obtidos por ele mesmo (BICHO, 1999), correspondendo a uma versão mecânica da inteligência apresentada por humanos para realizar uma mesma tarefa.

Mas simplesmente codificar robôs autônomos, sem princípios ou padrões, embora seja a tendência natural de iniciantes em robótica (JONES, 2004), pode levar a um código demasiadamente complexo, dificultando cada vez mais a inserção de novas funcionalidades. Assim, uma arquitetura que propõe um padrão organizacional para um controlador se faz necessária.

Entretanto a arquitetura impõe, além da organização, uma série de restrições para a resolução do problema (MATARIC, 1992). Assim, o conhecimento dos diferentes tipos de arquiteturas para controle de robôs proporciona não só diferentes formas de resolver um problema como também flexibilidade para a escolha de uma delas como arquitetura de um controlador.

2.2.1 Sense-Plan-Act

Os primeiros trabalhos nessa área utilizavam arquiteturas do tipo *Sense-Plan-Act*. Como o próprio nome sugere, a arquitetura basicamente corresponde a um laço infinito com três passos: sentir, ou seja, construir um modelo interno sobre o estado atual do mundo real; planejar, criando uma série de tarefas que, executadas, levam o estado atual do ambiente até o estado objetivo e; agir, colocando em prática cada um dos passos criados pela segunda fase.

A origem dessa arquitetura são os algoritmos de Inteligência Artificial tradicionais, que apresentam a mesma estrutura.

Essa solução é satisfatória para ambientes estáticos e previsíveis. Assim, o robô montava um modelo do ambiente através do sensoriamento, planejava as ações que deveria tomar e, uma vez feito o plano, executava cada uma das ações previstas até o fim do mesmo. Mas, como observado por Nilsson (1998), a eficiência desse tipo de arquitetura parte de pressupostos dificilmente encontrados em problemas reais, já que:

- O sensoriamento nunca é totalmente preciso, podendo levar diferentes estados do ambiente a uma mesma representação interna;
- A atuação também tem um grau de imprecisão, o que faz com que o novo estado obtido seja diferente do planejado, arruinando todo o resto do plano;
- Situações emergenciais podem requerer uma ação do robô quando o mesmo ainda está na fase de planejamento; e
- Nem sempre há recursos suficientes para a procura do estado objetivo dependendo do espaço de busca.

Uma forma de minimizar o problema causado pela imprecisão de sensores e atuadores é planejar uma sequência de ações, executar apenas a primeira delas e refazer o sensoriamento do ambiente, fazendo um novo plano a partir do estado atual.

2.2.1.1 STRIPS

STRIPS (*Stanford Research Institute Problem Solver*) se refere à linguagem formal de entrada para o planejador homônimo, sistema criado por Nils Nilsson e Richard Fikes e projetado para a arquitetura *Sense-Plan-Act*.

Esse consiste em construir uma pilha de objetivos e sempre tentar resolver o objetivo que se encontra no topo da mesma. Inicialmente coloca-se o objetivo final como primeiro elemento da pilha. Caso esse objetivo seja composto, decompõe-se o mesmo em objetivos parciais, empilhando cada um destes.

Quando o topo da pilha consiste em um objetivo atômico, ou seja, que não pode ser dividido em objetivos parciais, o sistema busca por uma regra aplicável. Um regra é uma tripla <PC, LR, LA> em que:

- PC é o conjunto de condições necessários para a aplicação da regra. Caso qualquer condição não seja satisfeita pelo estado atual do sistema, a regra não é aplicável para aquele estado.
- LR é a lista de remoção. Uma vez aplicada a regra, todos os itens pertencentes a essa lista devem ser retirados da representação do estado do sistema.
- LA é a lista de adição. Análoga à de remoção, todos os itens da lista de adição devem ser inseridos na representação do sistema quando a regra é aplicada. Pode-se entender que ambas a listas simbolizam as condições que deixam de/passam a existir quando a regra é executada.

O *STRIPS* utiliza o método da divisão-e-conquista. O algoritmo procura uma regra que possua, em sua lista de adição, o objetivo a ser alcançado. Uma vez encontrada, a lista de condições necessárias à aplicação da regra é adicionada aos objetivos. Para cada um desses novos objetivos, a mesma técnica é utilizada, até que se encontre uma regra cujo conjunto de condições seja atendido pelo estado atual do sistema. Aplica-se a sequência de ações planejadas, removendo e adicionando os novos elementos das respectivas listas ao estado do sistema, até que se atinja o objetivo original.

Um dos problemas é que não há garantia sobre a ordem de execução das instruções, existindo a possibilidade de um objetivo gerar mudanças no sistema que degenerem um objetivo já atingido.

2.2.1.2 RSTRIPS

Uma variação do STRIPS é o RSTRIPS, que resolve o problema de conflito entre objetivos intermediários através de mecanismos de regressão. Assim, se uma regra R1, que influencia uma condição C já alcançada e necessária para a regra R2 (ainda não aplicada), R1 não será aplicada. Há um rearranjo do plano, regredindo a condição C para uma condição C', através da regra que chega em C (RAINHA, 2007).

Mas o robô pode não ser o único a efetuar mudanças no ambiente em questão. Em ambientes dinâmicos e com mais agentes a elaboração de planos pode ser custosa - pela necessidade de vários planejamentos - ou impraticável, quando o tempo de planejamento supera o tempo em que o ambiente muda de estado sem ação alguma do agente em questão. Nesse tipo de caso, outras abordagens, como a arquitetura reativa, podem se mostrar mais aplicáveis, como será visto a seguir.

2.2.2 Robótica Reativa

As arquiteturas *Sense-Act-Plan* têm uma importante restrição: para começar a agir, o robô necessita construir primeiro todo o seu plano de ação. Além de causar uma sobrecarga a cada problema, todo o plano é descartado sempre que há uma mudança não esperada no estado atual do sistema.

A arquitetura reativa mantém o agente executando uma determinada ação até que uma informação relevante sobre o ambiente mude, alterando assim seu estado atual e, consequentemente, exigindo uma nova reação. Segundo Schoppers (1987), há uma forma de representar o comportamento do robô de forma a especificar reações apropriadas para todo possível estado do sistema com custo computacional moderado e sintetizado de forma automatizada.

2.2.2.1 Planos Universais

Os planos universais são expressões com a forma "sempre que uma situação que satisfaz a condição C surgir enquanto se busca o objetivo G, então a resposta apropriada é a ação A".

Nessa arquitetura, o estado atual do sistema é avaliado em tempo de execução e a resposta apropriada àquela situação é executada. Caso o ambiente mude de forma mais drástica não há replanejamento, há a reavaliação da situação, classificando como um novo estado e gerando a resposta apropriada.

Isso gera uma nova interpretação sobre planejamento. Como definiu Schoppers (1987), planejamento é a seleção, orientada a objetivo, de reações a possíveis estados do ambiente.

Para avaliar o estado do sistema em tempo de execução, reagindo de acordo com os planos universais, precisa-se de uma constante avaliação das informações sensoriais. Define-se então o ciclo (ou *loop*) de atualização o intervalo entre sucessivas coletas de dados pelos sensores.

2.2.2.2 Procedural Reasoning System

O *Procedural Reasoning System* (Sistema de Raciocínio Procedural) não só permite a criação e execução de planos como também possibilita a interrupção ou replanejamento quando necessário. Ao invés de gerar um plano completo desde o primeiro passo, replanejando quando um evento inesperado ocorre, o *PRS* intercala fases de planejamento e atuação, mantendo planos parciais ao longo da execução. Esse planejamento parcial é um dos fatores responsáveis pela alta reatividade dessa arquitetura.

Como descrito por Georgeff e Schoppers (1987), o *PRS* é baseado no modelo *believe-desire-intention*, constituído por uma base de dados - a representação daquilo que o agente acredita ser o mundo ao seu redor -, seus objetivos (desejos) e suas ações para atingir o objetivo atual, representado pela intenção.

A base de dados é a representação interna sobre o ambiente. Essa base é inicialmente povoada com definições feitas pelo usuário, incluindo propriedades sobre o próprio sistema, denominado expressões reflexivas.

Os objetivos são representados por uma coleção de condições a ser atingida, não só do ambiente mas também de estados internos do robô.

As intenções englobam tanto conhecimento sobre reação a diferentes cenários como ações para atingir determinados objetivos (RAINHA, 2007). Cada intenção descreve os passos do procedimento e uma especificação do ambiente para que tal intenção seja aplicável.

Esses sistemas reativos provaram sua utilidade sobre a solução de problemas em ambientes dinâmicos. Entretanto, a necessidade de representar condições do ambiente em expressões complexas, bem como a necessidade de módulos sequenciais e passagem linear da informação - do sensoriamento para o modelador, seguindo para um planejamento (ainda que parcial)

para enfim haver atuação - impediam o desenvolvimento de aplicações de tempo real, ou sistemas competindo com outros agentes por objetivos antagônicos.

Para esse tipo de cenário uma nova arquitetura era necessária. Proposta e desenvolvida por BROOKS et al. (1991), a arquitetura *Subsumption* foi capaz de atender a esse tipo de situação. No próximo capítulo será apresentada essa arquitetura, bem como a abordagem diretamente derivada dela, a robótica baseada em comportamento.

3 Robótica Baseada em Comportamento

A robótica baseada em comportamento é uma metodologia para projetar robôs, concedendo-os comportamentos inteligentes bioinspirados (RAINHA, 2007). Essa metodologia é baseada na arquitetura reativa denominada *Subsumption*.

3.1 Subsumption

A arquitetura *Subsumption* é baseada em módulos de controle, capazes de comunicarem entre si, sendo cada um desses uma máquina computacional simples. Os módulos são hierarquicamente organizados, de forma que níveis superiores podem suprimir ou alterar informações sensoriais antes que estas cheguem aos módulos inferiores, ou ainda suprimir a saída destes.

A decomposição que se tinha até, representada na Figura 1 então subdividia o problema horizontalmente em unidades funcionais (RAINHA, 2007), em que cada camada é responsável por uma função no agente - percepção, planejamento, atuação. Em seu trabalho, BROOKS et al. (1991) define a modularização através de comportamento para cumprir uma tarefa,

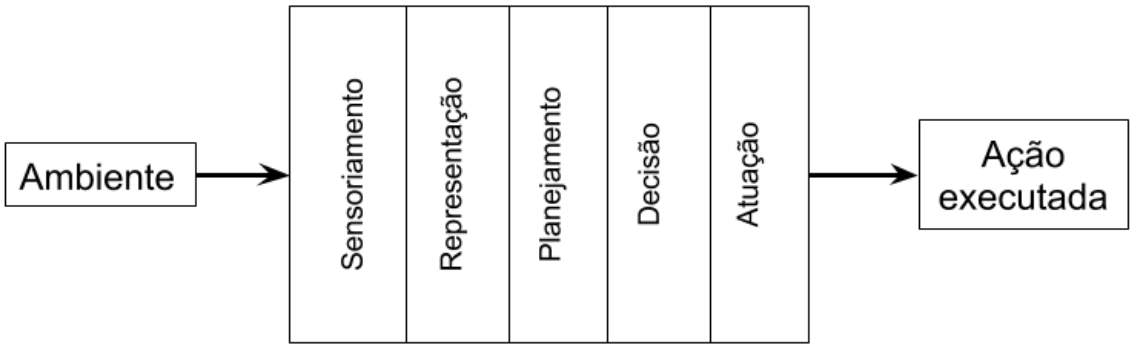

Figura 1 – Divisão esquemática horizontal utlizada pelas arquiteturas anteriores à *Subsumption*

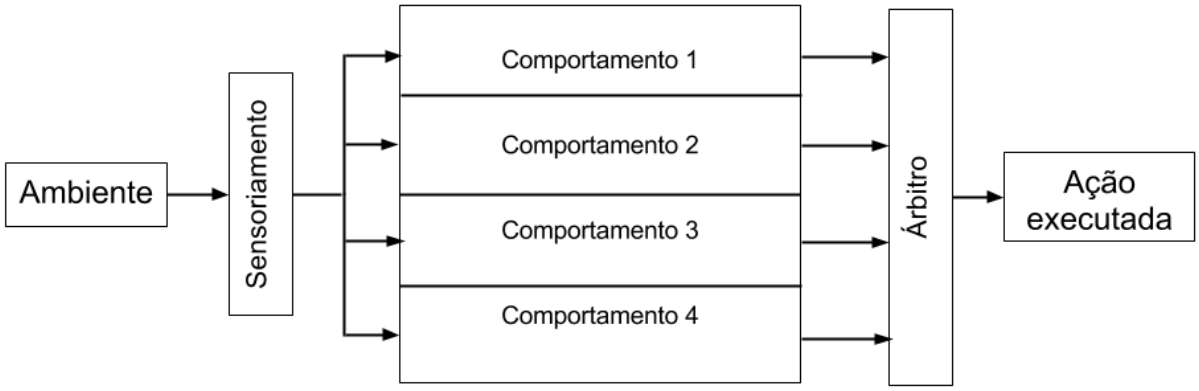

Figura 2 – Divisão orientada a comportamentos proposta por BROOKS et al. (1991)

como esquematizado na Figura 2. Dessa forma, a divisão passa a ser vertical, com todos os módulos trabalhando assincronamente a partir da entrada de sinais sensoriais.

Essa divisão vertical proporciona a execução paralela e assíncrona dos módulos. Isso permite a multiplicidade de objetivos, atribuindo cada objetivo desejado a um ou mais módulos diferentes. Os objetivos são atingidos de forma paralela, tendendo a um cenário final em que todos os objetivos são atendidos simultaneamente. Será visto mais à frente neste trabalho a resolução de conflitos que possam surgir a partir de objetivos que exijam ações diferentes, até mesmo antagônicas.

Além disso, a falha de um módulo não interfere na execução de módulos independentes dele, mantendo o funcionamento do agente , ainda que sub-ótimo. De forma análoga, a inserção de novos módulos independentes não muda a capacidade dos já implementados, garantindo uma capacidade de aditividade para novas funcionalidades sem a exigência do retrabalho em todo o sistema.

Segundo BROOKS et al. (1991), agentes autônomos e inteligentes devem ser capazes de:

- responder em tempo real a mudanças no ambiente dinâmico;
- ser robusta em relação a mudanças no ambiente, isto é, mudanças que afetem as propriedades do ambiente não devem fazer o sistema para de funcionar completamente, devendo funcionar da melhor forma possível em face de tais mudanças não projetadas;
- manter múltiplos objetivos e, de acordo com a situação atual, mudar o objetivo particular que estava perseguindo;
- realizar mudanças no mundo externo, tendo uma razão para existir.

Para implementar um sistema com as características da arquitetura *Subsumption*, são necessários vários elementos definidos pela metodologia baseada em comportamento, começando pelos sistemas de controle.

3.2 Sistemas de controle

As duas categorias mais amplas de sistemas de controle são chamados laço aberto e laço fechado (JONES, 2004). No primeiro, dá-se um comando e espera-se que o sistema obedeça-o.

Um controle de laço aberto é o tipo mais simples de controle, podendo ser suficiente caso haja a garantia de que o robô nunca encontrará condições variantes de ambiente. Para ambientes dinâmicos é necessário o uso de controles de laço fechado. Estes não só observam aquilo que o robô deve fazer como também continuamente avaliam como o robô o faz. Em outras palavras, o controle de laço fechado observa a saída do sistema para ajustar o comando de forma que o resultado seja o esperado.

3.2.1 Controles de laço aberto com parâmetros e estados

Um controle de laço aberto pode ser arbitrariamente complexo e dinâmico, bastando não se basear na saída do sistema para ser classificado como tal.

Suponha um robô-trem que trafega em um circuito dentro de uma plantação, funcionando como um espantalho. Para assustar os pássaros, deseja-se que o robô ande de tempos fazendo o maior barulho possível. Entretanto isso não deve ser feito de maneira contínua pois os pássaros podem se acostumar com a movimentação e passar a ignorar o robô.

Para que o sistema funcione de modo apropriado, são estabelecidos dois intervalos de tempo, um determinando quanto tempo o robô anda nos trilhos fazendo barulho e o outro para o tempo que fica parado. Na primeira ativação do controle, o robô começa a se movimentar. Quando o tempo de movimentação termina, ele espera pela outra quantidade de tempo até recomeçar o processo. O controle do robô, nessas condições, é dado por três parâmetros: a velocidade com que deve andar e os tempos de movimentação e espera. O ajuste desses parâmetros podem, portanto, mudar drasticamente como o robô se comporta.

Esse robô possui dois estados: ativo e inativo. Quando no primeiro, movimenta-se e faz barulho até que o tempo de movimentação seja atingido, mudando para o estado inativo. Este, por sua vez, é descrito pelo tempo que o robô deve ficar imóvel até que possa voltar ao estado anterior.

3.2.2 Controle *Bang-Bang*

Um controle *bang-bang* possui dois estados, mas, ao contrário do exemplo anterior, depende de informações da saída do sistema, sendo um controlador de laço fechado. Ele normalmente responde com um valor se a medida de uma quantidade de interesse está acima de um certo limite e um outro valor em caso contrário.

Embora simples, este controle pode servir, por exemplo, para regular o comportamento de um robô próximo a uma parede. Sabendo a distância do robô à parede, caso esta seja inferior a um limite, uma ação que force o robô a não se aproximar mais da parede (ou ainda que não permita que o mesmo tente atravessá-la) deverá ser tomada. Caso a distância seja maior que esse limite, essa preocupação passa a ser irrelevante e o robô fica livre para tomar outra ação qualquer.

Um importante conceito que emerge do controle *bang-bang* é o de histerese, ou seja, a tendência de um sistema de manter um certo comportamento na ausência do estímulo que o gerou. Se o limite imposto pelo controle for pontual, qualquer variação pequena cujo intervalo contenha o limite de ativação fará com que o sistema oscile entre comportamentos com frequência potencializada. Caso isso não seja previsto, há a possibilidade do sistema ficar sobrecarregado e se tornar menos eficiente com as constantes mudanças entre os estados do controle.

3.3 Comportamentos

Um sistema de controle por si gera um robô capaz de fazer apenas uma tarefa. Para realizar tarefas mais complexas, o robô deve se comportar de diferentes maneiras sob as diferentes circunstâncias em que se encontra. Surge então o conceito de comportamentos.

3.3.1 Comportamentos primitivos

Segundo Jones (2004), comportamentos primitivos possuem duas partes: um controle que transforma dados oriundos do sensoriamento em um saída através dos atuadores e um gatilho que determina quando a ativação desse comportamento é apropriada. Visto de fora, o sistema não sabe se a ausência do sinal de ativação interrompe a geração de saída pelo controle daquele comportamento ou se simplesmente sua saída é ignorada.

3.3.1.1 Comportamento servo e balístico

Tipicamente, um comportamento servo emprega um controle de laço fechado como componente de controle, sujeito a interrupções caso haja mudanças no ambiente em que o mesmo

está inserido. Já um comportamento balístico, como o nome sugere, uma vez disparado, segue um caminho de execução conhecido até que complete sua execução. Um comportamento balístico pode ser considerado como um pequeno plano dentro da arquitetura baseada em comportamentos.

Por sua natureza, os comportamentos balísticos devem ser usados com cuidado, pois impedirão a liberação o controle do sistema a outro comportamento até que tenham terminado sua sequência de operações. Dependendo do momento em que um comportamento balístico for executado, sua completude pode não ser garantida, e assim o sistema travará.

3.3.2 Máquina de estados finitos

Comportamentos puramente do tipo servo não guardam qualquer informação sobre o passado, baseando-se apenas na informação atual sobre o sistema e o ambiente para determinarem uma saída. Comportamentos desse gênero são ditos sem estado (JONES, 2004). Entretanto, em alguns comportamentos existe a necessidade de armazenamento de informação, caracterizando-o como uma máquina de estados finitos.

Controles com memória podem ser representados por máquinas de estados finitos. Estes definem os possíveis estados do sistema bem como as transições válidas de um estado a outro. A utilização de máquina de estados permite criar comportamentos com memória, evitando ou solucionando problemas nos quais o comportamento oscila entre um número finito de estados sem se aproximar regularmente de seu objetivo.

3.3.3 Sobrecarga de comportamentos

Para gerenciar complexidade de forma eficiente, é importante evitar a sobrecarga de comportamentos. A ideia é escrever uma coleção de comportamentos simples, cada qual gerenciando uma situação em específico. Por sobrecarga de comportamentos

Suponha um comportamento A projetado para lidar com um evento 1 e um comportamento B para um evento 2. Suponha ainda que a resposta a ser dada pelo robô deva ser qualitativamente diferente quando os eventos 1 e 2 acontecem ao mesmo tempo (ou com relativa proximidade) do que quando ocorrem de forma isolada. Ao invés de tornar mais complexo o comportamento A para tratar o caso da proximidade desses eventos, uma solução melhor seria criar um terceiro comportamento, C, cujo gatilho é a ocorrência dos dois eventos. Assim o sistema tende a ficar mais simples e mais robusto.

3.4 Arbitragem

Para perseguir vários objetivos, o robô precisa de vários comportamentos primitivos. Mas, caso mais de um comportamento seja disparado ao mesmo tempo, potencialmente gerando diferentes ordens para o robô, deve-se ter uma política que determine como essas ordens simultâneas devem ser tratadas. à essa política dá-se o nome de arbitragem.

Como mostrado na figura 2, a saída dos comportamentos é ligada ao árbitro e sempre que há competição entre comportamentos por um recurso, é papel do árbitro selecionar o comportamento que controlará aquele atuador. Para cada recurso que possa ser "disputado" por diferentes comportamentos deve-se ter um árbitro associado.

3.4.1 Arbitragem de prioridade fixa

Há uma estratégia simples para garantir que os comandos mais apropriados a um momento sejam executados. Dá-se prioridades fixas aos comportamentos, de acordo com sua importância para o sistema atingir seu objetivo, e quando houver comandos incompatíveis sendo ordenados, o comportamento de maior prioridade retém o controle dos atuadores do robô.

Para sistemas com mais de um árbitro, deve-se levar em conta que para diferentes recursos as prioridades de uso podem diferir também. Assim, cada árbitro deve dar uma prioridade para cada comportamento levando em conta o recurso com o qual está associado.

4 Aplicação

O controlador, de arquitetura *Subsumption*, desenvolvido através da metodologia baseada em comportamento foi projetado de baixo para cima (*bottom-up*): comportamentos mais primitivos são implementados primeiro, e a interação e arbitragem desses faz com que comportamentos de mais alto nível surjam. Com os comportamentos definidos, pode-se criar as tarefas, unidades lógicas que emergem a partir da arbitragem. Por fim é feito um treinador (*coach*) que designa diferentes tarefas para os robôs ao longo do jogo.

A estratégia foi elaborada de acordo com a interface fornecida pelo simulador para futebol de robôs *Middle League Simorosot*, descrito a seguir:

4.1 Simulador

O simulador utilizado é o *Middle League Simurosot*, distribuído pela FIRA *Federation of International Robot-soccer Association* e a inserção de estratégias nesse simulador é feita a partir da geração de uma biblioteca de vínculo dinâmico (DLL, do inglês *Dynamic-link library*). A escolha de um simulador já desenvolvido permitiu o foco na elaboração do controlador, além de ser mantido por uma federação internacional e já ter sido utilizado em outros trabalhos científicos, como em Haobin et al. (2009).

4.1.1 Sensoriamento e Atuação

Neste simulador, o sensoriamento é feito pelo mesmo, que informa ao controlador a posição e rotação atuais de cada robô e da bola, além do número de gols de cada time. Essas informações são atualizadas a uma taxa média de 60 vezes por segundo. Portanto, algoritmos de visão computacional para reconhecimento dos robôs não são necessários.

Quando a posição de cada robô é dada pela posição relativa deste a uma mesma origem, tem-se posicionamento global. Essa representação facilita não só a interpretação dos estados de jogo dependentes de posicionamento (distância a outro robô, ou invasão de uma área não permitida), como também facilita os comportamentos de movimentação em geral.

Em sistemas de visão e posicionamento locais, cada robô calcula a posição relativa dos outros elementos em relação à sua própria posição. Para isso, o robô, dotado de uma câmera individual, mapeia pontos relativos do campos (marcadores fora do campo ou marcas do campo, como o centro do mesmo) que estejam em seu alcance visual e calcula sua distância

relativa àquele marcador. Como esse sistema depende da visibilidade, ocorre com frequência situações em que um robô não sabe onde se encontra os robôs adversários, alguns dos companheiros ou até mesmo a bola. Em alguns casos, é possível que o robô perca seu referencial relativo e não saiba onde está no campo.

Sistemas de posicionamento global são, até o ponto permitido pelo sistema de posicionamento utilizado, mais confiáveis que sistemas locais. Mas sistemas globais envolvem altos custos computacionais e econômicos, como um sistema de visão computacional através de uma câmera em cima do campo ou um sistema GPS. Portanto, para robôs implementados é mais comum o uso de sistemas locais de visão e posicionamento, o que diminui a aplicabilidade do controlador desenvolvido nesse trabalho em outros tipos de robôs.

Também é responsabilidade do simulador a representação das interações físicas entre os elementos do jogo. Assim, a atuação é feita apenas com a força com que o motor acelera cada uma de suas rodas. Impedindo ações como reposicionamentos forçados pelo do controlador, mudando a posição do robô sem usar o movimento das rodas, por exemplo, o simulador garante que o controlador tem maior correspondência com um controlador para robôs reais.

4.1.2 Interface

A interface entre a estratégia criada e o simulador é feito através da implementação da estratégia como uma DLL. A própria *FIRA* oferece um projeto base, incluso no simulador, para permitir o começo do desenvolvimento. O esqueleto básico da DLL é o seguinte:

```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
                     ) {
   switch (ul_reason_for_call) {
   case DLL_PROCESS_ATTACH:
   case DLL_THREAD_ATTACH:
   case DLL_THREAD_DETACH:
   case DLL_PROCESS_DETACH:
      break;
   }

   return TRUE;
}
extern "C" STRATEGY_API void Create(Environment *env){
}
extern "C" STRATEGY_API void Destroy(Environment *env) {
```

```
}
extern "C" STRATEGY_API void Strategy(Environment *env) {
   switch (env->gameState) {
   case 0:
   case FREE_BALL:
   case PLACE_KICK:
   case PENALTY_KICK:
   case FREE_KICK:
   case GOAL_KICK:
      break;
   }

}
```

A função *DllMain* é a função da Dll que o sistema operacional executa quando a mesma é anexada em algum processo. Para a maioria dos casos neste simulador, incluindo este trabalho, não há necessidade de alteração no conteúdo desta função.

Na função *Create*, a estratégia deve alocar todos os recursos de memória necessários para sua execução. No trabalho aqui descrito, esta alocação compreende a memória necessária para detectar se o robô permaneceu muito tempo numa mesmo posição, como será explicado em breve. Analogamente, a função *Destroy* é chamada no momento em que a estratégia é liberada pelo simulador e deve desalocar toda a memória alocada para si, bem como terminar qualquer processo de escrita em arquivos, fechando-os.

A função *Strategy* é onde o controlador é realmente executado. A cada atualização do simulador é feita uma chamada a esta função, que deve atribuir os valores para as velocidades das rodas de cada robô de seu time. Como o simulador espera, a cada quadro, toda a atualização da estratégia, controladores que demoram mais do que o necessário na fase de planejamento e execução da estratégia não são prejudicados pela má performance, sendo um fator negativo avaliado no simulador.

O simulador ainda permite o posicionamento manual dos robôs antes do início da partida e sempre que o juiz sinaliza uma evento que paralisa o jogo, como uma falta ou um reinício por falta de movimento da bola. Após a permissão do juiz para prosseguimento do jogo, os robôs não podem ser manipulados pelos controladores humanos, dependendo exclusivamente da estratégia empregada para realizar suas tarefas.

4.2 Comportamentos

Os comportamentos são a unidade básica de inteligência em um sistema de robótica baseada em comportamento. Com comportamentos mais complexos surgindo da interação entre comportamentos primitivos, é importante que estes sejam robustos e eficientes em suas atribuições para que a interação entre comportamentos gere o resultado esperado.

4.2.1 GoTo

O primeiro comportamento implementado foi o de levar um robô até uma localização, denominado neste trabalho *GoTo*. Como o sistema dispõe de posicionamento global do robô, implementar esse comportamento significa representar vetorialmente a distância entre o robô e o alvo, para depois comparar a direção desejada com a direção atual do robô. Como o robô pode acelerar de forma máxima tanto diretamente para a frente quanto para a direção contrária, deve-se considerar que, caso a rotação necessária seja maior do que 90°, o robô deve acelerar para trás. Matematicamente, sendo $\vec{P_r}$ a posição do robô, $\vec{P_a}$ a posição do alvo e $\vec{\theta_r}$ a orientação do robô:

$$\vec{\theta_a} = S\vec{P_r} - \vec{P_a} \tag{4.1}$$

$$cos(\theta) = \vec{\theta_r} \cdot \vec{\theta_a} \tag{4.2}$$

$$(V_r, V_l) = \begin{cases} (V_{\max} * (1 - \theta), V_{\max}) & \text{se } \theta > 0 \\ (V_{\max}, V_{\max} * (1 + \theta)) & \text{se } \theta < 0 \end{cases} \tag{4.3}$$

Essa sistema é recalculado a cada atualização do simulador. Caso o alvo mude de direção, os novos valores calculados para a velocidade das rodas esquerda e direita serão readaptados e o robô irá em direção ao novo alvo, sem a necessidade de planejamento de caminhos.

Entretanto, o comportamento criado causa uma histerese: se o alvo permanecer fixo tempo suficiente para que o robô chegue até lá, este passará diretamente pelo alvo. Ao passar pelo alvo e avaliar que está logo a frente dele, o comportamento acelerará o robô no movimento contrário, até que o mesmo passe novamente sobre o local, entrando num ciclo que caracteriza a histerese robótica.

Ainda assim, para diversos comportamentos a aceleração máxima em direção a um alvo é mais eficiente que uma aceleração proporcional à distância do robô ao alvo. Ao invés de tratar a histerese dentro do comportamento, a escolha de projeto foi por manter o comportamento e tratar externamente, através de outros comportamentos de maior prioridade, os casos em que o alvo escolhido está próximo do robô o suficiente para que seja melhor mantê-lo parado.

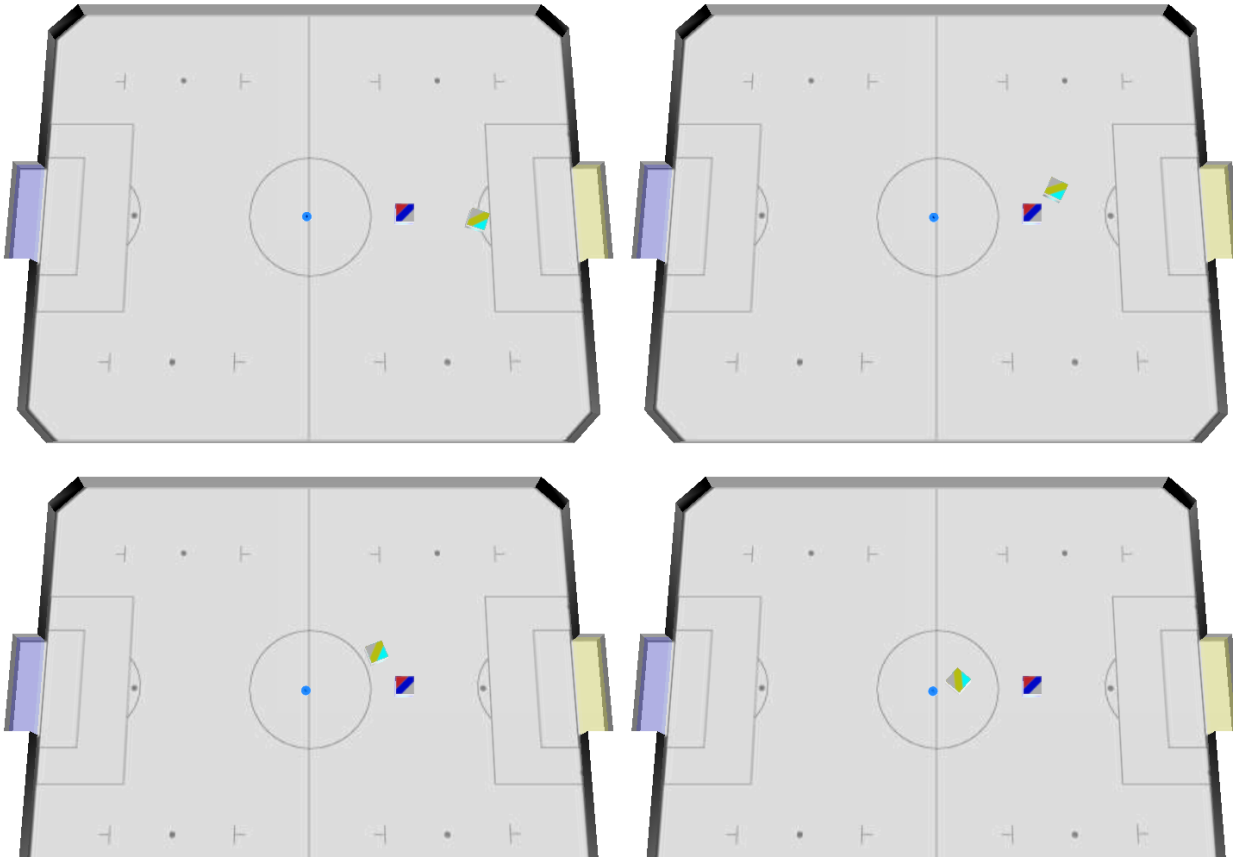

Figura 3 – Robô executando o comportamento Avoid para chegar até a bola

4.2.2 Avoid

Em alguns, é necessário que o robô evite contato com outros objetos para não gerar penalidades ou deixar o time em situação desfavorável. Como exemplo, um robô pode querer tirar a bola de perto da pequena área e, no processo, acabar empurrando o goleiro do próprio time, prejudicando o time. O comportamento *Avoid* impede que o robô se choque com a bola ou outro robô, avaliando quais objetos devem ser evitados.

Quando ativo, esse comportamento avalia se qualquer outro objeto está no caminho entre o robô e seu destino. Em caso afirmativo, avalia-se então se o obstáculo em questão deve ser evitado e, se esta for a situação encontrada, há o recálculo das velocidades das rodas para que o robô circule em volta do alvo, evitando contato com o mesmo enquanto se aproxima da posição pretendida, como exemplificado na figura 3.

A escolha dos objetos a serem evitados muda de acordo com o estado do jogo: se o robô está à frente da bola, esta passa a ser um obstáculo evitado para que o robô, enquanto tenta se posicionar atrás da bola, não a empurre para trás, favorecendo o adversário.

4.2.3 Stall

Com frequência o robô adota um comportamento que o faz ficar preso, seja, num adversário, companheiro ou até mesmo na borda do campo. Nesses casos, o comportamento *Stall* retira o robô dessa situação girando-o.

Esse é um comportamento que, uma vez ativado, faz com que o robô dê uma pequena sequência de giros, de forma a livrá-lo da obstrução encontrada. Como é um giro em torno do seu próprio eixo, há reduzida chance de que este movimento também seja impedido, e a nova rotação final faz com que o robô se livre da situação de paralisação quando o comportamento termina.

Considerando a base teórica, este é um comportamento balístico, já que uma vez disparado o robô não executará outro comando até que o comportamento todo seja terminado. Como é a ação de fuga mais simples e que funciona independentemente do cenário, este comportamento possui a maior prioridade. Comportamentos como esse devem ser projetados com cautela, devido à sua capacidade de tomar o controle do agente a qualquer momento e não devolvê-lo até o encerramento de suas ordens.

Para determinar se houve uma paralisação do robô, a informação sensorial de posição é somada à informação da saída de velocidade nas rodas: caso o robô esteja em uma mesma posição durante 1 segundo enquanto a saída de outros comportamentos desejam movê-lo desse ponto, *Stall* é disparado. Com isso, comportamentos que propositalmente mantenham o robô em uma dada posição não disparam esse comportamento.

4.2.4 Push

A função do comportamento *Push* é empurrar a bola, direcionando-a para um alvo. O comportamento analisa a posição da bola, do robô e do alvo para onde a bola deve ser empurrada, usando a linha definida pelas posições da bola e do alvo para definir uma posição atrás da bola. Caso o robô esteja próximo ao alvo, o comportamento GoTo recebe como alvo o gol adversário. Assim, o robô se deslocará em direção ao gol com a bola em seu caminho, fazendo com que o robô empurre a bola durante o movimento.

Quando o robô não está na posição correta para empurrar, seu objetivo passa a ser a posição atrás da bola. Para isso ele sempre tenta desviar de outros robô, para não atrapalhar companheiros do time que também estejam defendendo, e também da bola, já que empurrá-la enquanto se está à frente da bola pode dar vantagem ao time adversário.

Se a bola estiver entre o robô e o alvo e os três alinhados, basta o robô ir ao encontro do alvo. Caso a bola escape durante o movimento, o próprio comportamento reposicionará o robô atrás da mesma, já que a condição anterior do robô não estar alinhado atrás da bola passa a ser verdadeira.

O código que implementa a descrição deste comportamento é:

```
void Push(Robot* robot, Environment *env) {
   Vector3D behindBall, *ball;
   double hipot;
   behindBall.x = FLEFTX;
   behindBall.y = GMIDDLE;
   hipot = distance(&behindBall, &env->currentBall.pos);
   ball = &env->currentBall.pos;
   behindBall.x = ball->x + 5 * (ball->x - FLEFTX) / hipot;
   behindBall.y = ball->y - 5 * (GMIDDLE - ball->y) / hipot;
   if (distance(&robot->pos, &behindBall) > 4.0) {
      if (robot->pos.x < ball->x) {
         behindBall.x += 15.0;
         if (fabs(robot->pos.y - ball->y) < 5.0) {
            if (behindBall.y > GMIDDLE) {
               behindBall.y -= 10.0;
            } else {
               behindBall.y += 10.0;
            }
         }
      }
      GoTo(robot, &behindBall, env);
   } else {
      behindBall.x = FLEFTX ;
      behindBall.y = GMIDDLE;
      GoTo(robot, &behindBall, env);
   }
}
```

4.2.5 Intercept

Intercept é o comportamento que posiciona o robô entre a bola e um alvo, normalmente o gol que deve ser protegido.

Esse comportamento guarda a informação sensorial da atualização atual e da anterior, projetando esse deslocamento para inferir a direção para a qual a bola está sendo empurrada. Caso essa direção passe pela região a ser protegida, o robô calcula a posição a ser adquirida para ficar entre a bola e a região defendida.

A determinação de qual ponto o robô deverá ocupar sobre a reta dependerá do papel que o mesmo estiver representando no time, como será visto adiante.

4.2.6 StayAway

StayAway faz com que o robô se limite a uma região do campo, podendo se mover livremente na parte irrestrita, mas impedido de entrar em zonas determinadas.

À medida em que o robô se aproxima da fronteira da região permitida, a saída para a velocidade de suas rodas é reduzida ou zerada, para que o robô não invada a área proibida. Este comportamento possui prioridade maior que a de outros comportamentos, pois evitar infrações as leis do jogo é mais importante do que desempenhar qualquer outra tarefa, já que punições são aplicáveis nessas situações.

O robô então fica o mais próximo do solicitado por outros comportamentos sem invadir o espaço restrito. Essa proximidade é útil para que, caso a restrição seja liberada, o robô esteja na melhor posição possível para atender o outro comportamento. É o que acontece, por exemplo, com um robô que está impedido de entrar na área adversária existe um companheiro lá e somente um atacante pode ocupar a área adversária em um dado instante de tempo. Assim que o outro robô sair da área, o primeiro estará livre para completar o ataque por estar próximo durante todo o evento, como mostra a figura 4.

4.3 Arbitragem com prioridade variável

Os comportamentos descritos formam o núcleo da estratégia. Entretanto, se os parâmetros de cada comportamento forem diferentes, tem-se a capacidade de desenvolver robôs que atuam com diferentes papéis, sem a necessidade de projetar comportamentos diferentes para cada um deles. Papéis diferentes necessitam de diferentes prioridades para os comportamentos, o que é alcançado a partir de de um esquema de arbitragem variável.

Mesmo em esquemas de arbitragem com prioridade variável, alguns comportamentos possuem prioridade maior em todos os papéis possíveis. É o caso do comportamento *StayAway*, já que todos os papéis possuem regras de posicionamento que devem ser obedecidas acima dos objetivos individuais.

4.3.1 Attacker

O primeiro papel projetado foi o *Attacker*. Nele, o robô procura empurrar a bola sempre que possível, com alta prioridade para o comportamento *Push*. O *StayAway* mantém o robô forá da grande área do próprio time, mas deixa o robô livre para entrar na área adversária.

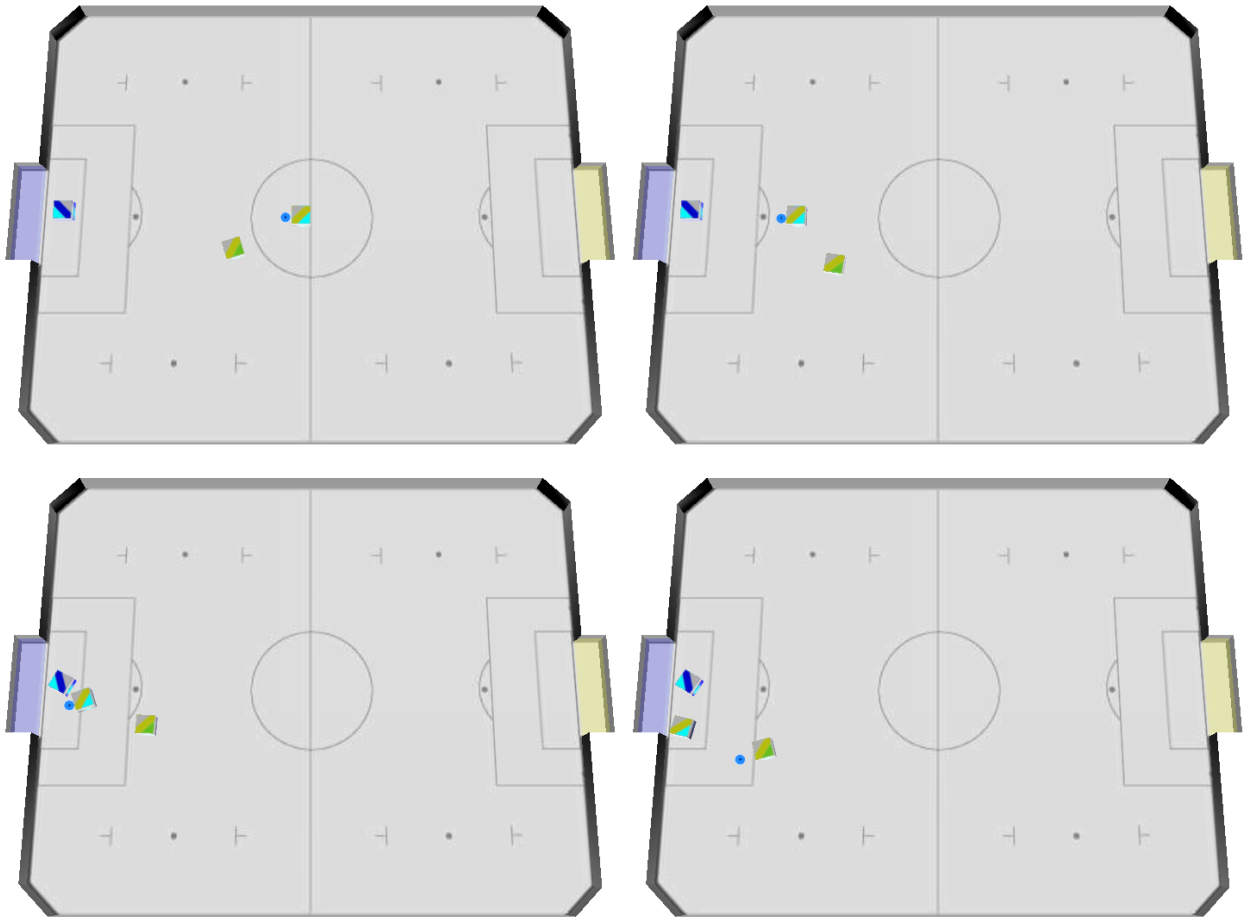

Figura 4 – StayNearby num primeiro momento faz com que o robô roxo saia do caminho do vermelho, e logo depois o mantém perto da área para recuperar a posse de bola após o ataque do vermelho ser defendido pelo time amarelo

Dada a restrição de apenas um mebro do time dentro da área adversária em qualquer instante de tempo, dois robôs não podem exercer a função de atacante ao mesmo tempo. Isso não significa que um mesmo robô deve ser o atacante durante toda a partida, como será visto na próxima seção.

4.3.2 Defender

Defender é o papel responsável por defender o time, objetivando evitar ou atrapalhar ataques adversários. Utilizando o *StayAway* para mantê-los dentro da grande área, os robôs possuem maior prioridade para o comportamento *Intercept*.

Como até 3 robôs podem permanecer na grade área, sendo apenas um permitido dentro da pequena área, três robôs podem ser defensores ao mesmo tempo. O robô que permanece dentro da pequena área é comparável a um goleiro, mas o papel do goleiro é exatamente o mesmo dos defensores da grande área, com diferença apenas na área de atuação. Logo, não

há a necessidade de especializar o goleiro em um papel diferente, apenas usá-lo como um *Defender* responsável pela pequena área.

4.3.3 Support

Support é um papel auxiliar, tendo como função acompanhar o atacante, estando à frente ou atrás do mesmo, para dificultar o roubo de bola pelo time adversário e maximizar a chance do time recuperá-la quando necessário.

Para permitir esse comportamento, um novo sensoriamento é inserido, que informa qual função os outros robôs do time estão executando. Considerando todo o time como um único sistema, este passa a ser um sensoriamento interno.

Atribuindo esses papéis a cada um dos jogadores, obtém-se um time funcional capaz de jogar o futebol de robôs. Contudo, a atribuição estática desses papéis no início do jogo é ineficiente considerando que o robô mais adequado para um ataque é aquele que possui a bola no momento. Um melhor funcionamento do time pode ser atingido atribuindo-se diferentes papéis ao robô ao longo do jogo, sendo esta atribuição responsabilidade da unidade denominada treinador.

4.3.4 Treinador

Através de uma arbitragem de prioridade variável, pode-se ter diferentes sequências de ativamento dos comportamentos que, quando observados em um nível de abstração maior, são os responsávei pelo desempenho de diferentes funções pelos robôs.

A unidade responsável por determinar o papel a ser desenvolvido por cada um dos robôs foi denominada *Treinador* dada sua analogia com a função deste de determinar a função de cada jogador em um time. Do ponto de vista da robótica, essa unidade é um árbitro que, dado o estado atual do sistema, deve determinar qual o papel a ser desempenhada por cada robô no time.

Externamente essa unidade parece envolver alta complexidade para trocar papéis em tempo real. Mas, considerando o treinador como um comportamento abstrato, podemos simplificar a implementação do mesmo.

Esse comportamento tem com informação sensorial a posição dos elementos do time, assim como o resto do sistema. Mas sua atuação ocorre sobre o próprio sistema: a saída é a atribuição de papéis a cada um dos robôs. Poderia-se considerar que o treinador não é um comportamento já que a atuação deve transformar o ambiente externo, mas a notabilidade da mudança é tal que mesmo um agente externo, observador ou mesmo time adversário, pode notar a mudança de papéis, sendo efetivamente uma atuação sobre o ambiente de jogo como um todo.

5 Conclusão

A demanda por robôs autônomos e inteligentes é crescente. Essa demanda gera a criação de arquiteturas mais avançadas: exigi-se mais do *hardware*, com processadores que, em média, dobram de capacidade de processamento a cada dois anos (MOORE, 1965), bem como do software. No segundo caso, novas arquiteturas são desenvolvidas de forma a ampliar não só a eficiência do robô como também a compreensão sobre a inteligência necessária para a realização de tarefas sem intervenção humana.

A partir do estudo sobre a robótica baseada em comportamento e outras arquiteturas antecessoras, foi possível avaliar qual teria maior aplicabilidade para um problema altamente dinâmico e imprevisível como o Futebol de Robôs. O estudo de arquiteturas anteriores não só serve como referência histórica da origem da robótica baseada em comportamento, como também fornece importantes definições, técnicas e ferramentas para aumentar a capacidade original dessa arquitetura.

Uma vez que essa base teórica foi estabelecida, o próximo passo foi o desenvolvimento de comportamentos básicos para o funcionamento de um robô. A partir da identificação dos diferentes comportamentos que o robô necessita para realizar cada uma de suas atividades, foram estabelecidos os comportamentos básicos que foram implementados nesse trabalho. O desenvolvimento e teste unitários de cada comportamento contribuíram para que cada unidade funcionasse de acordo com o especificado.

Por fim, observou-se a interação desses comportamentos básicos durante a execução de tarefas mais avançadas. A execução simultânea dos comportamentos primitivos permitiu a emersão de comportamentos mais complexos, como o suporte para a perda de bola por um companheiro, ou a defesa em grupo sem a disputa entre os robôs por um único ponto. O uso dos mesmos comportamentos, com parâmetros diferentes, para diferentes papéis para cada um dos robôs mostrou a flexibilidade da robótica baseada em comportamento para a simulação do futebol de robôs, sinalizando o potencial de se aplicar com sucesso essa arquitetura em uma implementação real do futebol de robôs.

5.1 Trabalhos Futuros

Uma sugestão natural para trabalho futuro a partir deste trabalho é a aplicação do mesmo em robôs reais, realizando as adaptações necessárias (embora a expectativa é de que não

sejam necessárias muitas mudanças, pelo simulador apresentar um nível aceitável de realismo) para que o robô tenha alto desempenho mesmo na presença de ruídos e falhas que não são representadas no simulador. Há ainda a oportunidade de se desenvolver outras estratégias, seguindo a arquitetura baseada em comportamento ou outra que se julgue aplicável, para comparação tanto em simuladores quanto com robôs mecânicos. Por fim, sugere-se ainda a comparação deste algoritmo com outros algoritmos da Inteligência Artificial Clássica em diferentes tarefas, como o $A*$ em tempo real, com o objetivo de comparar a complexidade espacial (de memória) e temporal (instruções executadas), além de avaliar a eficiência da execução da tarefa dentro do domínio de futebol de robôs.

APÊNDICE A – Código Fonte

```
// Strategy.cpp : Defines the entry point for the DLL application.

#define STRATEGY_EXPORTS

#include "stdafx.h"
#include "Strategy.h"
#include <math.h>
#include <stdlib.h>

BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
                     ) {
   switch (ul_reason_for_call) {
   case DLL_PROCESS_ATTACH:
   case DLL_THREAD_ATTACH:
   case DLL_THREAD_DETACH:
   case DLL_PROCESS_DETACH:
      break;
   }

   return TRUE;
}
const double PI = 3.1415923;
char myMessage[200]; //big enough???

void GoTo(Robot *robot, Vector3D *target, Environment *env);
void Stall(Robot *robot, Environment *env);
void Push(Robot* robot, Environment *env);
void Avoid(Robot* robot, Environment *env);
double inline distance(Vector3D *v1, Vector3D *v2);
```

```
void StayAway(Robot *robot, Vector3D *upperLeft, Vector3D *lowerRight,
    Environment *env);


extern "C" STRATEGY_API void Create(Environment *env) {
   int i;
   StrategyData *stratData = (StrategyData*) malloc(sizeof(StrategyData));

   for (i = 0; i < 5; i++) {
      stratData->lastPosition[i].x = env->home[i].pos.x;
      stratData->lastPosition[i].y = env->home[i].pos.y;
      stratData->lastLeftVel[i] = env->home[i].velocityLeft;
      stratData->lastRightVel[i] = env->home[i].velocityRight;
      stratData->stallCount[i] = 0;
      stratData->stalled[i] = 0;
   }

   env->userData = (void*) stratData;
}

extern "C" STRATEGY_API void Destroy(Environment *env) {
   free(env->userData);
}


extern "C" STRATEGY_API void Strategy(Environment *env) {
   // updating stratData
   switch (env->gameState) {
   case 0:
      // default
      Vector3D UL, LR;
      UL.x = env->home[0].pos.x - 5;
      UL.y = env->home[0].pos.y + 5;
      LR.x = env->home[0].pos.x + 5;
      LR.y = env->home[0].pos.y - 5;
      Push(&env->home[1], env);
      StayAway(&env->home[1], &UL, &LR, env);
      Avoid(&env->home[1], env);
      Stall(&env->home[1], env);
```

```
      Push(&env->home[0], env);
      Avoid(&env->home[0], env);
      Stall(&env->home[0], env);
      break;

   case FREE_BALL:
   case PLACE_KICK:
   case PENALTY_KICK:
   case FREE_KICK:
   case GOAL_KICK:
      break;
   }

   int i;
   StrategyData* stratData = (StrategyData*) env->userData;

   for (i = 0; i < 5; i++) {
      stratData->lastPosition[i].x = env->home[i].pos.x;
      stratData->lastPosition[i].y = env->home[i].pos.y;
      stratData->lastLeftVel[i] = env->home[i].velocityLeft;
      stratData->lastRightVel[i] = env->home[i].velocityRight;
   }
}

void Velocity(Robot *robot, int vl, int vr) {
   robot->velocityLeft = vl;
   robot->velocityRight = vr;
}

double inline AngleReduction(double ang) {
   while (ang > 180) {
      ang -= 360;
   }

   while (ang < -180) {
      ang += 360;
   }

   return ang;
```

```
}

double inline RadToDegree(double ang) {
   return (180.0 * ang) / PI;
}

double inline DegreeToRad(double ang) {
   return (PI * ang) / 180;
}

void GoTo(Robot *robot, Vector3D *target, Environment *env) {
   double gain = 60;
   Vector3D robotToTarget;
   double targetAngle;
   robotToTarget.x = target->x - robot->pos.x;
   robotToTarget.y = target->y - robot->pos.y;
   targetAngle = RadToDegree(atan2(robotToTarget.y, robotToTarget.x));
   targetAngle -= robot->rotation;
   targetAngle = AngleReduction(targetAngle);

   if (targetAngle > -90 && targetAngle < 90) {
      if (targetAngle > 0) {
         robot->velocityRight = MAX_VEL;
         robot->velocityLeft = MAX_VEL - (targetAngle * gain / 90.0);
      } else {
         robot->velocityRight = MAX_VEL + (targetAngle * gain / 90.0);
         robot->velocityLeft = MAX_VEL;
      }
   } else {
      if (targetAngle > 0) {
         robot->velocityRight = -MAX_VEL;
         robot->velocityLeft = -MAX_VEL + ((180 - targetAngle) * gain / 90.0);
      } else {
         robot->velocityRight = -MAX_VEL + ((180 + targetAngle) * gain / 90.0);
         robot->velocityLeft = -MAX_VEL;
      }
   }
}
```

```
double inline distance(Vector3D *v1, Vector3D *v2) {
   return sqrt((v1->x - v2->x) * (v1->x - v2->x) + (v1->y - v2->y) * (v1->y -
       v2->y));
}

void SubVector3D(Vector3D *output, Vector3D *v1, Vector3D *v2) {
   output->x = v1->x - v2->x;
   output->y = v1->y - v2->y;
}

void SetLine(double rot, Vector3D *pos, double *a, double *b) {
   rot = DegreeToRad(rot);
   *a = tan(rot);
   *b = pos->y - (*a) * pos->x;
}

void Avoid(Robot *robot, Environment *env) {
   double max_distance = 15.0;
   //double max_angle = 15.0;
   double max_angle = 68.4 - MAX_VEL * 0.42;
   double rotation;
   double robotRotation;
   int i, reversed = 0;
   robotRotation = robot->rotation;

   if (robot->velocityLeft < 0 && robot->velocityRight < 0) {
      robotRotation += 180;
      reversed = 1;
   }

   for (i = 0; i < 5; i++) {
      if (distance(&env->opponent[i].pos, &robot->pos) < max_distance) {
         rotation = RadToDegree(atan2(env->opponent[i].pos.y - robot->pos.y,
             env->opponent[i].pos.x - robot->pos.x));
         rotation = robotRotation - rotation;
         rotation = AngleReduction(rotation);

         if (rotation > -max_angle && rotation < 0) {
            if (reversed) {
```

```
            robot->velocityLeft *= (-rotation / max_angle);
         } else {
            robot->velocityRight *= (-rotation / max_angle);
         }

         max_distance = distance(&env->opponent[i].pos, &robot->pos);
      } else if (rotation > 0 && rotation < max_angle) {
         if (reversed) {
            robot->velocityRight *= (rotation / max_angle);
         } else {
            robot->velocityLeft *= (rotation / max_angle);
         }

         max_distance = distance(&env->opponent[i].pos, &robot->pos);
      }
   }
}

#ifdef BLUE
void Push(Robot* robot, Environment *env) {
   Vector3D behindBall, *ball;
   double hipot;
   behindBall.x = FLEFTX;
   behindBall.y = GMIDDLE;
   hipot = distance(&behindBall, &env->currentBall.pos);
   ball = &env->currentBall.pos;
   behindBall.x = ball->x + 5 * (ball->x - FLEFTX) / hipot;
   behindBall.y = ball->y - 5 * (GMIDDLE - ball->y) / hipot;
   if (distance(&robot->pos, &behindBall) > 4.0) {
      if (robot->pos.x < ball->x) {
         behindBall.x += 15.0;
         if (fabs(robot->pos.y - ball->y) < 5.0) {
            if (behindBall.y > GMIDDLE) {
               behindBall.y -= 10.0;
            } else {
               behindBall.y += 10.0;
            }
         }
```

```
      }
      GoTo(robot, &behindBall, env);
   } else {
      behindBall.x = FLEFTX ;
      behindBall.y = GMIDDLE;
      GoTo(robot, &behindBall, env);
   }
}
#else
void Push(Robot* robot, Environment *env) {
   Vector3D behindBall, *ball;
   double hipot;
   behindBall.x = FRIGHTX;
   behindBall.y = GMIDDLE;
   hipot = distance(&behindBall, &env->currentBall.pos);
   ball = &env->currentBall.pos;
   behindBall.x = ball->x - 5 * (FRIGHTX - ball->x) / hipot;
   behindBall.y = ball->y - 5 * (GMIDDLE - ball->y) / hipot;

   if (distance(&robot->pos, &behindBall) > 4.0) {
      if (robot->pos.x > ball->x) {
         behindBall.x -= 5.0;

         if (fabs(robot->pos.y - ball->y) < 5.0) {
            if (behindBall.y > GMIDDLE) {
               behindBall.y -= 5.0;
            } else {
               behindBall.y += 5.0;
            }
         }
      }
      GoTo(robot, &behindBall, env);
   } else {
      behindBall.x = FRIGHTX ;
      behindBall.y = GMIDDLE;
      GoTo(robot, &behindBall, env);
   }
}
#endif
```

```
void StayAway(Robot *robot, Vector3D *upperLeft, Vector3D *lowerRight,
    Environment *env) {
   Vector3D projection;

   if (robot->pos.y > upperLeft->y) {
      projection.y = upperLeft->y;
   } else if (robot->pos.y < lowerRight->y) {
      projection.y = lowerRight->y;
   } else {
      projection.y = robot->pos.y;
   }

   if (robot->pos.x < upperLeft->x) {
      projection.x = upperLeft->x;
   } else if (robot->pos.x > lowerRight->x) {
      projection.x = lowerRight->x;
   } else {
      projection.x = robot->pos.x;
   }

   float dist = distance(&robot->pos, &projection);

   if (dist < 0.1) {
      GoTo(robot, lowerRight, env);
   } else if (dist < 5) {
      robot->velocityLeft = 0;
      robot->velocityRight = 0;
   }
}

void Stall(Robot *robot, Environment *env) {
   int robotIndex;
   StrategyData* stratData = (StrategyData*) env->userData;

   for (robotIndex = 0; robotIndex < 5; robotIndex++) {
      if (&env->home[robotIndex] == robot) {
         break;
      }
```

```
   }

   if (stratData->stalled[robotIndex] == 1) {
      robot->velocityLeft = 125;
      robot->velocityRight = -125;
      stratData->stallCount[robotIndex]--;

      if (stratData->stallCount[robotIndex] <= 0) {
         stratData->stalled[robotIndex] = 0;
      }

      return;
   } else if ((fabs(robot->pos.x - stratData->lastPosition[robotIndex].x) <
       STALL_DISTANCE) && (fabs(robot->pos.y -
       stratData->lastPosition[robotIndex].y) < STALL_DISTANCE) &&
       ((fabs(robot->velocityLeft)) > 0.9 * MAX_VEL ||
       (fabs(robot->velocityRight)) > 0.9 * MAX_VEL)) {
      stratData->stallCount[robotIndex]++;

      if (stratData->stallCount[robotIndex] > STALL_MAX_COUNT) {
         stratData->stalled[robotIndex] = 1;
         return;
      }
   } else {
      stratData->stallCount[robotIndex] = 0;
   }
}
```

Referências

BICHO, E. Dynamic approach to behavior-based robotics: Design, specification, analysis, simulation and implementation. 1999. Disponível em: <http://hdl.handle.net/1822/210>. Citado na página 10.

BROOKS, R. et al. New approaches to robotics. *Science*, v. 253, n. 5025, p. 1227–1232, 1991. Citado 3 vezes nas páginas 14, 15 e 16.

GEORGEFF, A. L. L. M. P.; SCHOPPERS, M. J. *Reasoning and Planning In Dynamic Domains: An Experiment With A Mobile Robot*. 333 Ravenswood Ave., Menlo Park, CA 94025, Apr 1987. Citado na página 13.

HAOBIN, S. et al. Offensive formation strategy and control algorithm in simurosot (simulation of soccer robot)[j]. *Journal of Northwestern Polytechnical University*, v. 6, p. 035, 2009. Citado na página 21.

JONES, J. L. *Robot Programming*: A practical guide to behavior-based robotics. New York: McGraw-Hill, 2004. 293 p., 24 cm. ISBN 0071427783. Citado 7 vezes nas páginas 6, 8, 9, 10, 17, 18 e 19.

MATARIC, M. Behavior-based control: Main properties and implications. In: *In Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*. [S.l.: s.n.], 1992. p. 46–54. Citado na página 10.

MOORE, G. E. Cramming More Components onto Integrated Circuits. *Electronics*, IEEE, v. 38, n. 8, p. 114–117, abr. 1965. ISSN 0018-9219. Disponível em: <http://dx.doi.org/10-.1109/jproc.1998.658762>. Citado na página 31.

NILSSON, N. J. *Artificial intelligence: a new synthesis*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN 1-55860-467-7. Citado na página 10.

RAINHA, R. F. *Histórico de programação de robôs*. 66 f. Monografia (Graduação) — Universidade Federal de Juiz de Fora, Minas Gerais, 2007. Citado 3 vezes nas páginas 12, 13 e 15.

SCHOPPERS, M. *Universal Plans for Reactive Robots in Unpredictable Environments*. 1987. Citado 2 vezes nas páginas 12 e 13.