Explorar x Explotar e Escolher
Resgate de Vítimas de Catástrofes Naturais, Desastres ou Grandes Acidentes

Lucas Pujol de Souza[1], **Saulo Bergamo**[1]

[1]Departamento de Informática (DAINF) – Universidade Tecnológica Federal do Paraná (UTFPR)
Curitiba – PR – Brazil

Abstract. *This paper describes an implemented solution for the development of two robotic agents that explore an environment with victims and rescue them within a certain time using artificial intelligence techniques. The results indicate that although the solution works, it can be improved for bug fixing and better performance of both agents.*

Resumo. *Este artigo descreve uma solução implementada para o desenvolvimento de dois agentes robôs que exploram um ambiente com vítimas e as resgatam dentro de um determinado tempo utilizando técnicas de inteligência artificial. Os resultados indicam que apesar da solução funcionar, ela pode ser aperfeiçoada para a correção de bugs e uma melhor performance de ambos os agentes.*

1. Introdução

O problema proposto é dividido em duas partes. Na primeira, o objetivo é explorar um ambiente com vítimas e guardar sua localização e gravidade. Para isso, o agente explorador desenvolvido deve balancear a busca local e global para conseguir localizar todas as vítimas dentro do tempo proposto. Além disso, o explorador deve salvar as coordenadas dos nós que contém paredes ou obstáculos que serão utilizados pelo socorrista para melhor aproveitar o ambiente. O balanceamento entre exploração e explotação é largamente discutido na elaboração de algoritmos evolucionários [Črepinšek et al. 2013]. Já na segunda, o agente socorrista deve escolher o melhor caminho com base nas informações concedidas pelo explorador com o objetivo de resgatar o máximo possível de vítimas no tempo estipulado. Os agentes são colocados em um ambiente de tamanho desconhecido e em um ponto aleatório, devendo assim construir suas próprias coordenadas para navegação.

2. Fundamentação Teórica

A construção dos agentes necessita da definição de alguns algoritmos que foram utilizados ou cogitados para a implementação:

2.1. Busca em profundidade

A busca em profundidade é um algoritmo de busca em árvore que começa pela raiz e explora o máximo possível de cada ramificação antes de retroceder para a próxima ramificação. Em um grafo, cada nó pode ter vários filhos, e também pode haver ciclos no grafo, o que significa que o algoritmo pode visitar o mesmo nó mais de uma vez. Para evitar entrar em um ciclo infinito, é necessário manter um registro dos nós que já

foram visitados. O algoritmo de busca em profundidade em grafos começa pelo nó inicial e explora o máximo possível de cada ramificação antes de retroceder para a próxima ramificação. Se encontrar um nó que já foi visitado anteriormente, o algoritmo não o explora novamente. No entanto, a busca em profundidade pode levar muito tempo para encontrar as vítimas se o ambiente for muito grande.

2.2. Busca em largura

A busca em largura é um algoritmo de busca em grafos que explora todos os nós em um nível antes de avançar para os nós do próximo nível. Em outras palavras, ela explora o grafo em largura, começando pelo nó inicial e avançando para todos os nós que estão a uma distância de um nível da raiz antes de avançar para o próximo nível. A busca em largura é uma técnica mais abrangente que a busca em profundidade, pois sempre encontra a solução mais próxima da raiz e a solução mais curta em termos de número de arestas percorridas. Isso ocorre porque o algoritmo explora todos os nós de um determinado nível antes de avançar para o próximo nível, garantindo que todas as soluções nesse nível tenham sido consideradas antes de avançar para o próximo nível. No entanto, a busca em largura pode ser mais lenta que a busca em profundidade, especialmente em grafos com muitos ramos.

2.3. Busca A*

A busca A* é um algoritmo de busca heurística que tenta encontrar o caminho mais curto entre um nó inicial e um nó objetivo em um grafo ou em um espaço de estados. Ela utiliza informações heurísticas para tentar guiar a busca em direção ao objetivo de forma mais eficiente do que outros algoritmos de busca não informados, como a busca em largura ou em profundidade. O algoritmo A* usa duas funções para guiar a busca: a função heurística e a função de custo. A função heurística estima a distância do nó atual até o nó objetivo, enquanto a função de custo avalia o custo total do caminho do nó inicial até o nó atual. O algoritmo calcula um valor de custo total para cada nó e escolhe o nó com o menor custo total para explorar em seguida. Uma das vantagens do algoritmo A* é que ele pode ser aplicado a problemas complexos com um grande número de estados possíveis, como jogos de tabuleiro ou planejamento de rotas em mapas [Rios et al. 2016]. Além disso, a heurística usada pode ser adaptada ao problema específico, permitindo que o algoritmo seja otimizado para um conjunto particular de circunstâncias. No entanto, a eficiência do algoritmo depende da qualidade da função heurística escolhida e da complexidade do problema a ser resolvido.

2.4. Algoritmos Genéticos

Os algoritmos genéticos são uma técnica de otimização baseada em princípios evolutivos da seleção natural. Eles são usados para resolver problemas de otimização complexos, onde existem muitas soluções possíveis e não há uma solução claramente melhor que as outras. O algoritmo começa com uma população de soluções candidatas que são representadas por cromossomos, geralmente codificados como cadeias de bits. Essas soluções são avaliadas de acordo com um critério de aptidão (fitness) que reflete o quão bem cada solução atende ao problema a ser resolvido. Em seguida, o algoritmo usa operadores genéticos, como reprodução, mutação e seleção natural, para gerar uma nova geração de soluções candidatas a partir da população atual. O operador de reprodução é usado

para combinar dois cromossomos e gerar um novo cromossomo, enquanto o operador de mutação é usado para modificar aleatoriamente um cromossomo existente. A seleção natural é usada para escolher os indivíduos mais aptos da população atual para sobreviver e produzir descendentes para a próxima geração. O processo é repetido várias vezes, gerando novas gerações de soluções a cada iteração. Com o passar do tempo, a população de soluções candidatas tende a convergir para uma solução ótima ou sub-ótima, dependendo da qualidade do conjunto de soluções iniciais e da eficácia dos operadores genéticos utilizados. [de Lacerda and De Carvalho 1999]

3. Metodologia

A solução para o problema proposto foi elaborada na linguagem Python, utilizando como base o simulador VictimSim.

3.1. Agente Explorador

Para o agente explorador, foi utilizada uma busca em profundidade com backtracking para explorar o ambiente e guardar as informações de paredes e vítimas, tanto sua localização quanto seu estado. O processo de retorno para a base utiliza uma aproximação do A* com a heurística da distância euclidiana, definida como a soma da raiz quadrada da diferença entre x e y em suas respectivas dimensões. Se o explorador utilizar 85% do tempo ou estiver tão longe que a sua distância euclidiana em números absolutos seja equivalente a 15% do tempo restante, ele deve retornar para base.

3.2. Agente Socorrista

O agente socorrista utiliza um algoritmo genético para aprender qual a melhor solução. Para isso, a gravidade da vítima é determinante como prioridade de atendimento/resgate, ou seja, o agente sempre resgata primeiramente todas as vítimas em estado crítico e posteriormente aquelas em estado instável, potencialmente instável e estável nesta ordem. O socorrista utiliza os mesmos condicionais que o explorador para retornar para a base antes que o tempo acabe.

4. Resultados e análise

A implementação dos agentes foi bem sucedida, com as vítimas sendo encontradas e resgatadas na maioria das vezes. No entanto, existem algumas ressalvas:
1. O agente explorador apresenta um bug e entra em loop indo e voltando de dois quadrados adjacentes. A causa é desconhecida e não foi solucionada.
2. O agente socorrista para de funcionar subitamente caso seu tempo restante seja muito curto.

4.1. Métricas

Nos dados de treino, as métricas alcançadas foram as seguintes:
PVE (Porcentual de vítimas encontradas): 69,05%
PTE (Porcentual do tempo de exploração utilizado): 86,83%
VEG (Pontuação de vítimas encontradas ponderada por classe de gravidade): 0,72
PEG (Porcentagem de graves encontrados): 67%
PVS (Porcentual de vítimas salvas): 30,95%

PTS (Porcentual do tempo gasto pelo socorrista): 58,25%
VSG (Pontuação de vítimas salvas ponderada por classe de gravidade): 0,32
PSG (Porcentagem de graves salvos): 32%

No teste 2, as métricas alcançadas foram:
PVE: 100%
PTE: 58,6%
VEG: 1
PEG: 100%
PVS: 100%
PTS: 38,12%
VSG: 1
PSG: 100%

No teste 3:
PVE: 100%
PTE: 15,66%
VEG: 1
PEG: 100%
PVS: 100%
PTS: 23%
VSG: 1
PSG: 100%

Notou-se uma eficácia e eficiência melhor nos arquivos de teste.

5. Conclusões

Apesar de funcionar, a solução encontrada não é a ideal. Os bugs encontrados podem prejudicar sua execução e causar travamentos, colocando em cheque seus objetivos. O agente socorrista poderia ser melhorado com a aplicação de uma função fitness que considerasse maior gravidade e menor distancia simultaneamente ou então que considerasse o custo total do caminho percorrido. Além disso, o problema poderia ser sanado com a aplicação de algoritmos gulosos, algoritmo de Dijkstra de menor caminho ou ate mesmo o algoritmo A* que evitariam movimentos desnecessários observados nesta solução.. Além disso, o problema do socorrista com a falta de tempo poderia ser sanado com a aplicação de algoritmos gulosos, algoritmo de Dijkstra ou até mesmo o algoritmo A*. Novas modelagens com o objetivo de mitigar os problemas encontrados tornariam os agentes mais confiáveis e estáveis, com uma taxa de sucesso próxima ao cem por cento.

Referências

Črepinšek, M., Liu, S.-H., and Mernik, M. (2013). Exploration and exploitation in evolutionary algorithms: A survey. *ACM computing surveys (CSUR)*, 45(3):1–33.

de Lacerda, E. G. and De Carvalho, A. (1999). Introdução aos algoritmos genéticos. *Sistemas inteligentes: aplicaçoes a recursos hıdricos e ciências ambientais*, 1:99–148.

Rios, M. L., Neto, F. S., and Netto, J. F. M. (2016). Análise e comparação dos algoritmos de dijkstra e a-estrela na descoberta de caminhos mínimos em mapas de grade. In *Anais do I Encontro de Teoria da Computação*, pages 887–890. SBC.

Apêndice I - Instruções para execução

Para rodar o código, é necessária a instalação do Python no computador. Caso não possua, baixe em https://www.python.org/downloads/.

1- Baixe o arquivo “codigo.zip” e o extraia em uma pasta

2- Dentro da pasta, clique com o botão direito do mouse e escolha “Abrir no terminal”

3- Digite o seguinte comando: python3 main.py "arquivo", onde arquivo é a pasta com os testes que deseja rodar (TESTE2, TESTE3).