INSTITUTO POLITÉCNICO DE BEJA

Escola Superior de Tecnologia e Gestão

Licenciatura em Engenharia Informática

Estruturas de Dados e Algoritmos

Relatório


Damien Fialho, 11243

Eduardo de Sousa Fernandes, 12927

Beja

2013

1. Introdução

O ato de procurar o caminho mais curto para chegar ao nosso destino é um instinto primitivo, é por isso que engenheiros desenvolveram algoritmos de procura inteligente e eficiente.

A procura do caminho mais curto consiste em dar um passo de cada vez, mas para tal, deve-se analisar cada passo possível antes e avaliar qual o próximo passo a dar.

Neste trabalho será abordado e testado este problema a nível computacional através do algoritmo A* (A Estrela). Este algoritmo é uma extensão do algoritmo de Dijkstra do qual será feita uma breve introdução a fim de se perceber a base do funcionamento do algoritmo A*.

1.1 Objectivos e Motivação

Os objectivos deste trabalho são:

- o estudo do algoritmo A*;
- a criação de um software que implemente o algoritmo A*;
- a análise da performance do algoritmo A*;
- tirar conclusões acerca do algoritmo A* ;

O que motivou a elaboração deste trabalho foi a importância que estes algoritmos de procura inteligente e eficiente representam no mundo dos jogos, isto é, para implementar inteligência artificial num jogo, é importante conseguir calcular qual o caminho mais curto para ir de um ponto a outro.

1.2 Contributos

Para a elaboração deste trabalho foram utilizados conhecimentos adquiridos na pesquisa e no estudo efectuado sobre o algoritmo A* na internet.

1.3 Estrutura do Documento

Introdução - Abordagem breve sobre o que se trata, os objectivos, a motivação e os contributos relacionados com este trabalho.

Parte teórica - Abordagem teórica do algoritmo A*, isto é, as duas origens, do que se trata, para que serve, como funciona, conclusão e previsão dos resultados experimentais.

Parte experimental - Abordagem prática da elaboração deste trabalho, ou seja, como foi implementado o algoritmo A*, como foi testado, quais os resultados dos testes e breve análise e conclusão dos testes.

Conclusão - Síntese da totalidade do trabalho desenvolvido e breve crítica do mesmo.

Bibliografia - Referencias consultadas para a elaboração deste trabalho.

Anexos - Imagens auxiliares e código desenvolvido.

2. Parte Teórico

2.1 Introdução ao algoritmo A*

A fim de entender o que é o algoritmo A* é necessário entender o que é um algoritmo, quais as suas origens, qual a sua finalidade e como implementá-lo.

Um algoritmo consiste na sequência finita de instruções bem definidas necessárias para realizar uma tarefa.

O algoritmo A* consiste numa extensão do algoritmo Dijkstra criado pelo cientista de computação Edsger Dijkstra em 1956 e publicado em 1959. Este algoritmo soluciona a caminho mais curto do ponto num grafo e consiste num algoritmo que começa por procurar todos os vizinhos do node inicial, (um node é um ponto único que tem informação extra para além das suas coordenadas) e em seguida repete o processo para cada um desses vizinhos atribuindo-lhes um node parente e assim sucessivamente até chegar ao node final, isto é, até chegar ao seu destino.

2.2 Algoritmo A*

Agora que se conhecem as bases pode-se explicar o algoritmo A*.

Tal como o algoritmo de Dijkstra, este algoritmo tem como finalidade encontrar o caminho mais curto para chegar do ponto A ao ponto B. O que difere os dois algoritmos é o uso de heurísticas para calcular qual o caminho mais curto na implementação do algoritmo A*. A heurística consiste num cálculo da distância prevista de um node qualquer ao node final.

O algoritmo A* é constituído por duas listas, uma aberta e outra fechada, onde regista todos os nodes vizinhos dos nodes por onde passa e os nodes por onde passa respectivamente. A fim de seleccionar qual o node actual (node que está a ser verificado), este algoritmo coloca na lista fechada o node cujo resultado da soma da heurística de um node vizinho mais um custo definido na implementação, que representa a dificuldade de passar de um node para outro, for menor, esta soma é representada por: $f(n) = h(n) + g(n)$.

2.3 Conclusão da parte teórica

Como extensão do algoritmo de Dijkstra, o algoritmo A* é muito mais eficiente na procura do caminho mais curto num grafo.

A eficiência mencionada anteriormente deve-se ao cálculo de qual o node que aparenta estar mais próximo do node final antes de executar a próxima instrução. Este cálculo é uma mais valia, uma vez que, é possível alcançar o destino com menos instruções não desperdiçando tempo em nodes que se encontram longe do node final.

2.4 Previsão dos testes

Uma vez que o algoritmo estiver implementado, serão realizados testes a fim de calcular a sua eficiência através do cálculo do tempo de execução do algoritmo A*.

Se o algoritmo for bem implementado, ao calcular o tempo de execução e inserir os dados dos testes num gráfico, é esperado que se obtenha uma recta linear crescente em que quanto maior for o número de nodes, maior será o tempo de execução.

3. Parte Experimental

3.1 Introdução ao projecto desenvolvido

Nesta fase será apresentado todo o trabalho realizado na elaboração deste projecto relacionado com a implementação computacional do algoritmo A*, na realização e análise do desempenho e as conclusões referentes a estas análises.

Poderá ser verificado que o tempo de execução depende do número de nodes em que fazemos a leitura, o que também é ser influenciado pelo método de cálculo das heurísticas.

3.2 Realização Experimental

A linguagem de programação: C# 5.0. Esta foi a linguagem de programação escolhida porque é simples e tem uma biblioteca predefinida muito vasta.

Ambiente de desenvolvimento: Microsoft Visual Studio 2012 Version 11 Update 4.

Sistemas operativos: Microsoft Windows 7 Ultimate;

Microsoft Windows 8.1 Professional.

Hardware: Intel Core i7-3840QM CPU @ 2.80GHz,

16 GB RAM;

Intel Core i7-4710MQ CPU @ 2.5GHz,

8 GB RAM.

3.3 Sistema Experimental

A documentação referente ao nosso software encontra-se na pasta "Documentation" e, tal como o manual de utilizador na pasta "Manual", ambos podem ser encontrados na página do Google code referente a este projecto abaixo apresentado.

(https://code.google.com/p/eda1314-11243-12927/source/browse/)

3.4 Medições Experimentais

As medições que serão realizadas dizem respeito ao tempo de execução que o software demora a encontrar o caminho mais curto entre dois pontos utilizando o algoritmo A* implementado na linguagem de programação C#.

No que diz respeito à precisão das medições, irão ser realizadas várias experiências com o mesmo número de nodes, serão calculadas as suas médias separadamente, e serão também realizadas várias experiências com diferente número de nodes, em que iremos comparar as média das médias previamente calculadas. Serão então apresentados os resultados num gráfico para melhor visualização e, por consequência, análise.

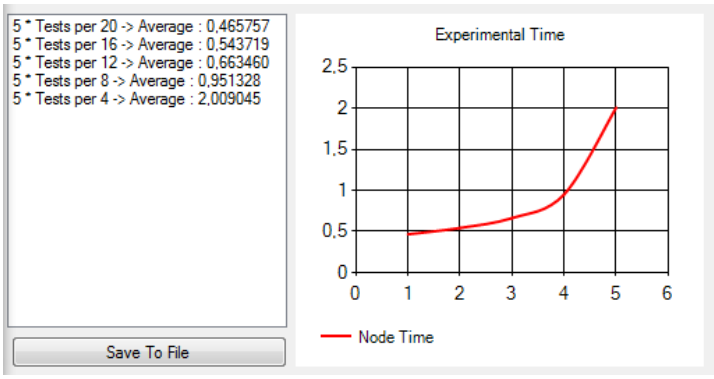
3.5 Protocolo Experimental

É nesta fase que será apresentado todo o trabalho efectuado, bem como as medições, os resultados das medições e uma breve reflexão desses resultados.

Software criado: O software desenvolvido consiste num software que lê um ficheiro PGM, o converte para uma imagem com extensão Bitmap e apresenta essa imagem numa janela. Nesta janela é apresentada a imagem e juntamente com botões de funcionamento que permitem a execução dos testes previstos no capítulo anterior (3.4).

Medições: Para a medição dos tempos de execução, foram efectuados cinco testes com diferente número de nodes, os quais são constituídos por cinco testes com medições com o mesmo número de nodes, nomeadamente: 20, 16, 12, 8 e 4.

Após a medição dos tempos de execução destes testes pode-se concluir que, tal como previsto, o tempo de execução aumenta dependendo da quantidade de medições efectuadas. Embora foi previsto um comportamento mais exponencial por parte dos valores obtidos nas medições.



Os valores observados na lista representam as médias dos testes elaborados com o mesmo número de nodes e o gráfico representa o comportamento da média com os resultados experimentais.

4. Conclusão

O estudo, a implementação e análise do algoritmo A* foram elaborados com sucesso, isto é, o software desenvolvido ficou funcional e foram efectuados os testes previstos com sucesso.

O algoritmo A* é sem sombra de dúvida um pathfinder muito inteligente e eficiente, porém, é complicado entender o bom funcionamento deste algoritmo a fim de o implementar com sucesso.

É muito provável que este algoritmo volte a ser implementado várias vezes pela nossa parte, devido ao simples facto de ser um dos pathfinder mais eficientes actualmente.

5. Bibliografia

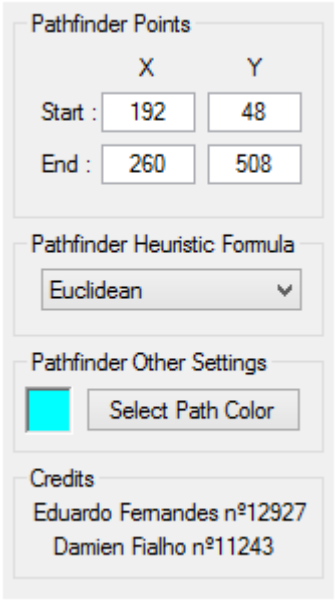
Algoritmo de Dijkstra: http://en.wikipedia.org/wiki/Dijkstra's_algorithm

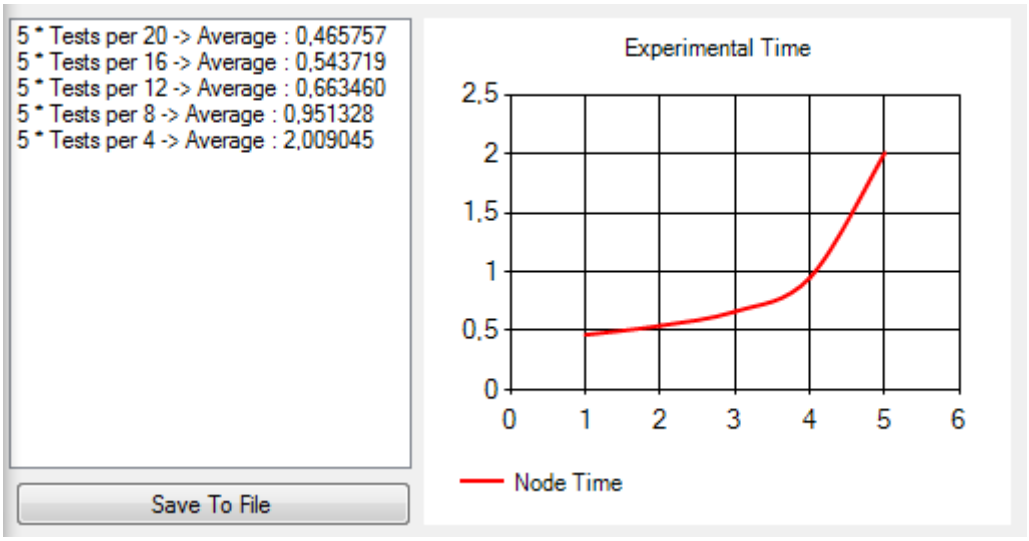
Algoritmo A*: http://en.wikipedia.org/wiki/A*_search_algorithm

Algoritmo A*: http://theory.stanford.edu/~amitp/GameProgramming/

Anexos

Imagens da interface do software elaborado:






Pepino-A-Star .: Main Menu :.
Find Path
Close

Código source do algoritmo A* elaborado:

```
namespace Pepino_A_Star
{
    /// <summary>
    /// The Node Class
    /// </summary>
    public class Node
    {
        public double _cost;
        public double _heuristic;
        public double _total;

        public Vector2 _pos;

        public List<Node> _neightbors;

        public Node _parent;

        /// <summary>
        /// Node Constructor
        /// </summary>
        /// <param name="pos">Node Position</param>
        public Node(Vector2 pos)
        {
            this._neightbors = new List<Node>();
            this._pos = pos;
            this._cost = 0;
            this._heuristic = 0;
        }

        /// <summary>
        /// Calculates the Total f()
        /// </summary>
        public void CalcTotal()
        {
            _total = _cost + _heuristic;
        }

    }

    /// <summary>
    /// @author Eduardo Fernandes nº12927
    /// @author Damien Fialho nº11243
    ///
    /// @date 06/06/1024
    /// @code https://code.google.com/p/eda1314-11243-12927/
    ///
    /// The Pathfinder Class. Created by us. Inspired by
http://theory.stanford.edu/~amitp/GameProgramming/
    /// </summary>
    ///
    public static class AStarPathFinder
    {

        /// <summary>
        /// Calculates the Cost based on the color given
        /// </summary>
```

```
        /// <param name="color">The color, from 0 to 255</param>
        /// <returns></returns>
        public static double CalculateCost(byte color)
        {
           double STEPS = 220;
           return (double)(255 - STEPS) / color;
        }

        /// <summary>
        /// Checks if the position is outside the picture.
        /// </summary>
        /// <param name="_pos">The Position to check</param>
        /// <returns></returns>
        public static bool IsOutsidePicture(Vector2 _pos)
        {
            return (_pos.X - 1 < 0 || _pos.Y - 1 < 0 || _pos.Y + 1 >
GlobalStuff.Height || _pos.X + 1 > GlobalStuff.Width);
        }

        /// <summary>
        /// Creates the Nod Neighbors
        /// </summary>
        /// <param name="_nod">The Node</param>
        /// <param name="_finalNode">The End Node</param>
        public static void CreateNeighbors(Node _nod, Node _finalNode)
        {
            _nod._neightbors.Clear();

            for (int X = -1; X < 2; X++)
                for (int Y = -1; Y < 2; Y++)
                {

                    if (GlobalStuff._heuristicMODE == 0)
                        if ((X == 0 && Y == 0) || (X == -1 && Y == -1) || (X == -
1 && Y == 1) || (X == 1 && Y == 1) || (X == 1 && Y == -1)) continue; // Do not
add center.

                    if (GlobalStuff._heuristicMODE == 1)
                        if (X == 0 && Y == 0) continue;

                    int XNew = _nod._pos.X + X;
                    int YNew = _nod._pos.Y + Y;

                    if (IsOutsidePicture(new Vector2(XNew, YNew))) continue;

                    Node _neigh = new Node(new Vector2(XNew, YNew));

                    if (GlobalStuff._heuristicMODE == 0)
                        _neigh._heuristic = ManhattenH(_neigh, _finalNode);
                    else
                        _neigh._heuristic = Euclidean(_neigh, _finalNode);

                    _neigh.CalcTotal();

                    _nod._neightbors.Add(_neigh);
                }

        }
        /// <summary>
        /// Finds the Path between the start node and the end node
        /// </summary>
        /// <param name="_start">Start Node</param>
```

```
        /// <param name="_end">End Node</param>
        /// <param name="Steps">Each Time Step</param>
        /// <returns>The Average</returns>
        public static double FindPath(Node _start, Node _end, int Steps)
        {

            Dictionary<int, Node> _openList = new Dictionary<int, Node>();
            Dictionary<int, Node> _closedList = new Dictionary<int, Node>();
            List<double> _nodeTimes = new List<double>();

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();


            int Infc = 0;

            Node _current = _start;
            _openList[Get1DVector(_start._pos.X, _start._pos.Y)] = _start;

            while (true)
            {
                int X = _current._pos.X;
                int Y = _current._pos.Y;

                if (X == _end._pos.X && Y == _end._pos.Y )
                {
                    _end._parent = _current;
                    break;
                }

                _current = GetMinTotal(_openList);
                _openList.Remove(Get1DVector(_current._pos.X, _current._pos.Y));
                _closedList[Get1DVector(_current._pos.X, _current._pos.Y)] =
_current;

                CreateNeighbors(_current, _end);

                foreach (Node neigh in _current._neightbors)
                {

                    int INDX = Get1DVector(neigh._pos.X, neigh._pos.Y);

                    if (_closedList.ContainsKey(INDX)) continue;

                    if (_openList.ContainsKey(INDX))
                    {
                        double NewCost = _current._cost +
CalculateCost(GetPGMData(neigh._pos.X, neigh._pos.Y));

                        if (NewCost < _openList[INDX]._cost)
                        {

                            _openList[INDX]._parent = _current;
                            _openList[INDX]._cost = NewCost;
                            _openList[INDX].CalcTotal();

                        }
                    }
                    else
                    {

                        neigh._parent = _current;
```

```
                        neigh._cost = _current._cost +
CalculateCost(GetPGMData(neigh._pos.X, neigh._pos.Y));
                        neigh.CalcTotal();

                        _openList[INDX] = neigh;
                    }

                }

                if ((Infc % Steps) + 1 == 1)
                {
                    _nodeTimes.Add((double)stopwatch.ElapsedMilliseconds);
                }

                Infc++;
            }

            _nodeTimes.Add((double)stopwatch.ElapsedMilliseconds); // Add the
last time.

            double totas = 0;

            foreach (double sad in _nodeTimes)
                totas += sad;

            return (totas / Infc);

        }

        /// <summary>
        /// Converts 2D to 1D
        /// </summary>
        /// <param name="x"></param>
        /// <param name="y"></param>
        /// <returns>The Vector</returns>
        ///
        public static int Get1DVector(int x, int y)
        {
            return (y * GlobalStuff.Width + x);
        }

        /// <summary>
        /// Gets the stored PGM Data
        /// </summary>
        /// <param name="x"></param>
        /// <param name="y"></param>
        /// <returns>The Color</returns>
        public static byte GetPGMData(int x, int y)
        {
            return (byte)GlobalStuff._pgmData[(y * GlobalStuff.Width + x)];
        }

        /// <summary>
        /// Manhatten Heuristic
        /// </summary>
        /// <param name="_start">Node to Check</param>
        /// <param name="_end">End Node</param>
        /// <returns></returns>
        public static double ManhattenH(Node _start, Node _end)
        {
            double dx = Math.Abs(_start._pos.X - _end._pos.X);
            double dy = Math.Abs(_start._pos.Y - _end._pos.Y);
```

```
            return (dx + dy);
        }

        /// <summary>
        /// Euclidean Heuristic
        /// </summary>
        /// <param name="_start">Node to Check</param>
        /// <param name="_end">End Node</param>
        /// <returns></returns>
        public static double Euclidean(Node _start, Node _end)
        {
            double dx = Math.Abs(_start._pos.X - _end._pos.X);
            double dy = Math.Abs(_start._pos.Y - _end._pos.Y);

            return Math.Sqrt(dx * dx + dy * dy);
        }

        /// <summary>
        /// Gets the Min Node total on the List.
        /// </summary>
        /// <param name="_list">The List to Check from</param>
        /// <returns>The Min Node</returns>
        public static Node GetMinTotal(Dictionary<int,Node> _list)
        {
            Node MAX = new Node(new Vector2());
            MAX._total = double.MaxValue;

            foreach (Node _nd in _list.Values)
            {
                if (_nd == null) continue;

                if (_nd._total < MAX._total)
                    MAX = _nd;
                else
                    continue;
            }

            return MAX;
        }

    }
}
```