Instituto Tecnológico de Aeronáutica

Divisão de Engenharia de Computação

CTC-20 Estruturas Discretas para Computação

Projeto Final

Implementação de algoritmos de busca de caminhos

Alunos:
Alexandre V. F. Muzio
Gabriel I. Magalhães
Matheus M. C. A.Camargo
Victor V. Pascoal

Prof. responsável:
Carlos H. Ribeiro

9 de julho de 2015

Descrição detalhada das atividades realizadas:

- Alexandre Muzio

  Criação da classe ImageMapCreator para geração de um mapa a partir de uma imagem de seus obstáculos. Implementação da estrutura de dados do grafo e seus respectivos testes do JUnit. Implementação dos algoritmos DFS, BFS, Dijkstra, A* e seus respectivos testes do JUnit e da classe responsável por gerar estatísticas.

- Gabriel Magalhães

  Implementação do protótipo da interface gráfica com o usuário (GUI). Criação da classe TextFileMapCreator que gera um mapa através de um arquivo de texto.

- Matheus Camargo

  Implementação do Navigation Mesh: divisor da área do mapa em um grafo de regiões retangulares, execução do A* para encontrar o melhor caminho de retângulos, divisão das arestas em determinado número de pontos, execução do A* para definir o caminho final. Implementação do visualizador do referido algoritmo, com interface para o usuário definir com o mouse os obstáculos.

- Victor Pascoal

  Criação da classe ConfigManager, um Singleton com um XML Parser para manejar os arquivos de configuração sem ser necessária a recompilação do código. Geração dos mapas de testes e compilação da tabela comparativa de estatísticas dos algoritmos implementados.

1 Motivação

Encontrar rotas eficientes entre dois pontos de forma rápida é um problema amplamente estudado em ciência da computação. As aplicações são diversas, variando desde a criação de inteligências artificiais para a indústria de jogos e a movimentação de robôs na vida real até o gerenciamento de rotas em aplicações baseadas em GPS, como o Google Maps.

Existem diversas soluções para esse problema, sendo a maioria fortemente baseada no estudo de grafos e algoritmos dentro dessas estruturas, o que está fortemente ligado aos conceitos estudados durante o curso de CTC-20 Estruturas Discretas para Computação. O presente trabalho é um estudo de algumas dessas soluções, incluindo comparações dos resultados (tempo de execução dos algoritmos e tamanho da rota encontrada) para diversos mapas.

No presente trabalho, foram estudados duas abordagens para o problema em questão.

A primeira delas consiste em uma série de algoritmos exatos baseados na iteração por pixels do mapa. Foram implementados e testados os seguintes algoritmos:

1. Breadth-First Search
2. Best-First Search
3. Depth-First Search
4. Algoritmo de Dijkstra
5. Algoritmo A*

A segunda solução consiste em uma abordagem por áreas, que gera uma solução em tempo computacional de execução consideravelmente menor, mas não gera, necessariamente, uma solução ótima.

Os algoritmos e as abordagens serão melhor detalhados na Seção 2.

2 Metodologia

O método a ser utilizado na resolução do problema descrito está intrinsecamente ligado a como o "mapa"é estruturado dentro do código. Nesse aspecto, foram propostas duas abordagens: partição em grid e partição em Navigation Mesh.

A primeira consiste na interpretação do mapa como um conjunto de pixels e a cada um desses é associado um valor booleano representando se o mesmo pode ser utilizado para formar um caminho ou se é um obstáculo. Essa implementação tem algumas desvantagens, como por exemplo: para um número pequeno de pixels, isto é, nós no grafo que representa o mapa, o caminho pode conter muitos "zigzags"em vez de curvas suaves. Assim, torna-se necessário o uso de uma quantidade significativa de pixels, gastando memória em excesso e aumentando o número de instruções a ser realizada para percorrer o caminho encontrado, sendo então desfavorável para aplicação em um robô real.

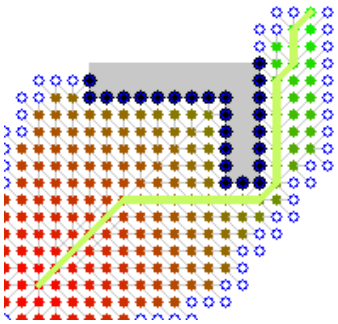


Figura 1: *Exemplo de A**

O mapa é lido a partir de três formas diferentes, como mostradas nas classes contidas no pacote *utils.gridMapCreator*, seja através de uma imagem, arquivo de texto ou via interface com o usuário. A Figura 2 mostra um protótipo da interface gráfica desenvolvida.

(a) *Layout principal da interface gráfica.* **(b)** *Layout de escolha dos algoritmos.*

Figura 2: *Interface gráfica.*

Em seguida, o mapa é instanciado como GridMap ou VertexMap, definidos no pacote *representations.maps*.

O pacote *representations.graph* contém a implementação de um grafo genérico, obtida utilizando o conceito de *Generics* do Java, fazendo uso de uma lista de adjacência e que pode ser utilizado nas duas abordagens supracitadas. Os tipos primitivos utilizados no projeto, como vértices e arestas do grafo, foram implementados no pacote *representations.primitives*.

Para determinar os caminhos entre dois pontos em um grafo, os seguintes algoritmos foram implementados e testados.

1. Breadth-First Search, ou Busca em Largura, é um algoritmo utilizado para grafos onde todas as arestas tem o mesmo custo. Durante a expansão, possuem prioridade os vértices com menor distância à raiz. Possui complexidade de tempo $O(|V| + |A|)$, onde $|V|$ e $|A|$ são, respectivamente, o número de vértices e arestas do grafo.

2. Depth-First Search, ou Busca em Profundidade é um algoritmo utilizado para encontrar o menor caminho entre dois pontos em um grafos onde todas as arestas tem o mesmo custo. Durante a expansão, possuem prioridade os vértices adjacentes ao último vértice visitado. Possui complexidade de tempo $O(|V| + |A|)$, onde $|V|$ e $|A|$ são, respectivamente, o número de vértices e arestas do grafo.

3. Best-First Search é um algoritmo de busca pelo menor caminho baseado em heurísticas, no qual se expande primeiro o nó com maior potencial segundo a heurística. A complexidade de tempo desse algoritmo depende da heurística [6].

4. Algoritmo de Dijkstra: O algoritmo de Dijkstra é utilizado para determinar-se o menor caminho entre dois pontos em um grafo com arestas de peso não negativo. Possui complexidade de tempo $O([|A| + |V|]log|V|)$, onde $|V|$ e $|A|$ são, respectivamente, o número de vértices e arestas do grafo [1].

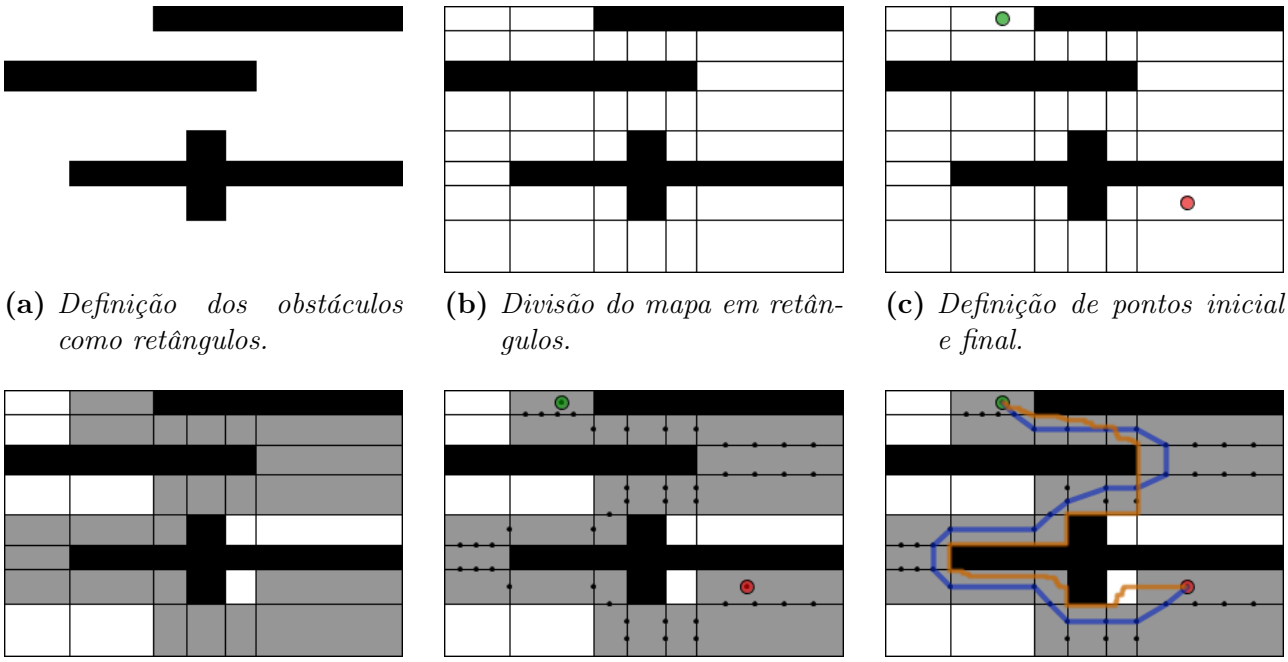
5. Algoritmo A*: O algoritmo A*, algoritmo da categoria Best-First Search, é amplamente utilizado para a determinação de rotas, baseando-se em heurísticas para as distâncias entre dois pontos. Para encontrar um caminho ótimo, é necessário que sua heurística seja admissível, isto é, que nunca superestime a distância mínima para chegar ao destino. A complexidade de tempo desse algoritmo depende da heurística. É importante citar que, no caso particular da heurística ser nula, trata-se do algoritmo de Dijkstra. A escolha da heurística é uma das etapas mais importantes em se tratando de problemas de otimização que utilizam algoritmos Best-First Search. Para o problema de pathfinding em grids, optou-se pelo uso da heurística Manhattan, já consagrada na literatura para resolução desse tipo de problemas [2].

Na segunda abordagem, o mapa pode ser entendido como um conjunto de polígonos convexos, onde os nós do grafos correspondem aos centros geométricos desses polígonos. Em um primeiro momento é feita uma busca pelo conjunto de polígonos que contêm a rota mais eficiente ao destino e, em um segundo momento, é descoberto quais os pontos de cruzamento do caminho com as arestas de cada polígono. As divisões permitem a criação de um grafo com dimensões muito menores do que as de um grafo contendo todos os pixels, de modo que encontrar uma rota satisfatória se torna uma tarefa menos custosa em termos de tempo de execução e de memória. Esse procedimento pode ser representado pelo pseudocódigo a seguir e visualizado graficamente na Figura 3.

```
procedure navMeshAlgorithm(obstacleList, mapSize) {
    rectangleGraph = divideMap(obstacleList, mapSize);
    rectanglePath = aStar(rectangleGraph);
    pointGraph = rectanglePathToEdgePoints(rectanglePath);
    pointPath = aStar(pointGraph);
    return pointPath;
}
```

Na primeira linha do código toma-se cada aresta de cada obstáculo e prolonga-se ela até os limites do mapa, formando assim uma matriz de retângulos. Dessa matriz, tomam-se os centros geométricos de cada retângulo e gera-se um grafo, que é passado para o algoritmo de busca A*, que gerará o caminho mostrado em cinza. Tendo esse caminho, usa-se a função *rectanglePathToEdgePoints* para gerar um conjunto de pontos sobre as arestas em comum dos retângulos cinzas. Ela impõe uma limitação para o espaço entre esses pontos de 10 a 25 pixels, para que eles fiquem bem distribuídos. Com isso, é possível executar mais uma vez o A* e então obter o caminho final em azul. A título de comparação, tem-se em laranja o caminho encontrado pelo algoritmo de busca padrão, executado num grafo contendo cada pixel como nó. A solução gerada por essa abordagem não é, necessariamente, a solução ótima. Porém, é, em geral, uma aproximação viável e satisfatória, como mostram os resultados da Seção 4.

O conceito de testes, no âmbito da engenharia de computação, é fundamental para a criação e desenvolvimentos de aplicações extensas. Dessa forma, optou-se pela incorporação de *Unit Testing* ao projeto. A framework de testes escolhida foi JUnit para a realização de testes na maioria sos algoritmos e dos tipos primitivos implementados, possibilitando verificar se a cada alteração no código os testes eram completados corretamente, facilitando a etapa de *debugging*. A plataforma Eclipse possui integração com o JUnit e torna possível selecionar se o programa ou os testes irão ser executados.



(a) *Definição dos obstáculos como retângulos.*

(b) *Divisão do mapa em retângulos.*

(c) *Definição de pontos inicial e final.*

(d) *Busca de melhor caminho dentre os retângulos.*

(e) *Divisão das arestas comuns em pontos.*

(f) *Resultado da busca final com caminho em azul.*

Figura 3: *Visualização das etapas do algoritmo proposto usando Navigation Mesh.*

Foi criada uma interface com o usuário, onde é possível criar um mapa personalizado contendo caminhos e obstáculos, definir um ponto inicial e final e rodar os algoritmos implementados, bem como disponibilizar as estatísticas, tempos de execução do DFS, BFS, Dijkstra, A* e Navigation Mesh, dessa descoberta de rota. Os pacotes graphicInterface, com dependência do graphicInterface.buttonListeners, e draw são os responsáveis por essa etapa de criação da GUI.

3 Material e Avaliação

Todo o projeto foi desenvolvido na linguagem Java utilizando a plataforma Eclipse. Para a parte gráfica do software, foi utilizada a biblioteca Processing [3].

Como consulta de implementação, utilizou-se dois principais recursos, a biblioteca *JGraphT* [4] e o curso de Algoritmos 2 de Princeton [5].

O esquema a seguir mostra a estrutura de pastas (pacotes do Java) na raiz do projeto. A estrutura completa pode ser encontrada no Apêndice.

```
pathfinder
├── algorithm
├── draw
├── graphicInterface
├── main
└── representations
```

```
└── statistics
└── utils
```

Ao todos foram 57 arquivos *.java* na pasta pathfinder e mais 13 arquivos *.java* de testes que totalizavam 30 unit tests. Para critérios de avaliação dos resultados, foram utilizadas as seguintes métricas:

1. O tempo de execução do algoritmo
2. A distância total do caminho encontrado

Ambos os parâmetros foram medidos para diversos casos de teste, contendo mapas de diferentes tamanhos e diferentes densidades de obstáculos. Cada mapa foi testado para cada um dos algoritmos de busca de rota descritos na Seção 2.

Em seguida, comparou-se a performance da abordagem Navigation Mesh com a do algoritmo de busca de rotas que apresentou a melhor performance segundo nossos testes.

4 Resultados e análise

Como métrica de distância, optou-se por usar a distância Manhattan, por se adequar muito bem como quantificador de distâncias em grids bi-dimensionais.

Utilizou-se 6 mapas, construídos pela interface gráfica desenvolvida, para analisar o desempenho dos algoritmos. A Figura 4, por exemplo, representa a imagem correspondente ao mapa *testMap4*.

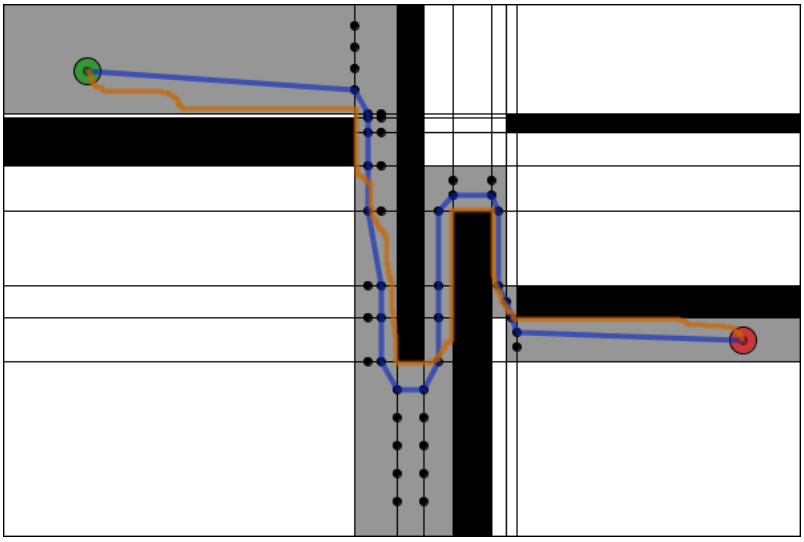


Figura 4: *Exemplo de mapa analisado*

As tabelas 1 até 6 demonstram os resultados obtidos pelas execuções do algoritmos.

Algoritmo	Tempo de execução (ms)	Tamanho da rota (pixels)	Tempo relativo (%)	Tamanho relativo (%)
DFS	61	17542	13	100
BFS	454	474	100	3
Dijkstra	278	474	61	3
A*	170	474	37	3
NavMesh	2	516	0.44	3

Tabela 1: *testMap1 (300x200 pixels)*

Algoritmo	Tempo de execução (ms)	Tamanho da rota (pixels)	Tempo relativo (%)	Tamanho relativo (%)
DFS	61	13696	38	100
BFS	42	472	26	4
Dijkstra	160	472	100	4
A*	155	472	97	4
NavMesh	1.8	486	1.1	3

Tabela 2: *testMap2 (300x200 pixels)*

Algoritmo	Tempo de execução (ms)	Tamanho da rota (pixels)	Tempo relativo (%)	Tamanho relativo (%)
DFS	255	58324	16	100
BFS	438	2056	27	3
Dijkstra	1620	2056	100	3
A*	1051	2056	64	3
NavMesh	1.9	2512	0.11	4

Tabela 3: *testMap3 (600x400 pixels)*

Algoritmo	Tempo de execução (ms)	Tamanho da rota (pixels)	Tempo relativo (%)	Tamanho relativo (%)
DFS	374	79521	10	100
BFS	619	925	17	1
Dijkstra	3696	925	100	1
A*	871	925	23	1
NavMesh	2	987	0.054	1

Tabela 4: *testMap4 (600x400 pixels)*

Algoritmo	Tempo de execução (ms)	Tamanho da rota (pixels)	Tempo relativo (%)	Tamanho relativo (%)
DFS	1446	411534	9	100
BFS	3630	2102	24	0.5
Dijkstra	14891	2102	100	0.5
A*	997	2102	7	0.5
NavMesh	5	2310	3.3	0.6

Tabela 5: *testMap5 (1900x1000 pixels)*

Algoritmo	Tempo de execução (ms)	Tamanho da rota (pixels)	Tempo relativo (%)	Tamanho relativo (%)
DFS	6621	527480	47	100
BFS	3774	3668	27	0.7
Dijkstra	14136	3668	100	0.7
A*	12628	3668	9	0.7
NavMesh	8	4722	0.063	0.9

Tabela 6: *testMap6 (1900x1000 pixels)*

Fica evidenciado, por meio das estatísticas geradas pelos seis mapas de teste, que a solução por Navigation Mesh é computacionalmente mais eficiente em termos de tempo de execução, mas encontra um caminho sub-ótimo. No contexto de games, por exemplo, o Navigation Mesh parece ser o algoritmo mais indicado pois gera uma solução satisfatoriamente aproximada com tempo de execução da ordem de centenas de vezes por segundo.

O algoritmo DFS não é eficiente pois sempre busca investigar o primeiro vértice adjacente sem qualquer análise. Ele não é aceitável no contexto de busca em grafo pois, muitas vezes, não se tem o grafo a ser investigado completamente em memória, sendo construído iterativamente. Isso acontece nos casos em que o grafo a ser investigado é muito grande.

Além disso, é interessante observar que, para o caso do grid, em que todas as arestas tem comprimento unitário, a implementação do Dijkstra é essencialmente equivalente a do BFS. No entanto, como o algoritmo do Dikjstra é implementado com uma *Priority Queue*, que possui operações de adição e remoção em O(logN), em contraste com as operações de adição e remoção do BFS, que são em tempo constante. Devido a isso, pode-se observar um melhor desempenho do BFS nos resultados.

5 Conclusões

No presente trabalho, foram estudadas duas metodologias para solucionar o problema de encontrar uma rota entre dois pontos em um dado mapa.

A primeira delas consistiu em algoritmos exatos que produzem soluções ótimas, iterando pelos pixels do mapa por meio de uma série de algoritmos. Foram comparadas as performances dos algoritmos e os resultados foram coerentes com o esperado segundo uma revisão da literatura. O

algoritmo que produziu o caminho ótimo foi o A*, no qual a heurística utilizada foi a distância Manhattan entre um dado ponto e o ponto de destino.

A segunda abordagem consistiu na implementação do sistema Navigation Mesh. Uma comparação com os resultados do algoritmo de melhor performance da primeira abordagem, A*, mostra que o tempo de execução dessa solução é consideravelmente menor, principalmente para mapas grandes. Uma outra vantagem é que a quantidade de pontos que determinam o caminho resultante é bem menor, o que se traduz em uma trajetória mais suave. No entanto, o caminho gerado por essa solução não é, necessariamente, o menor.

Essa implementação do Navigation Mesh poderia ser aprimorada de algumas maneiras futuramente:

- Melhorar a divisão inicial do mapa em retângulos, fazendo com que as arestas só gerem linhas dividindo o mapa até encontrar um obstáculo, em vez de elas se estenderem até os limites do mapa;
- Aceitar diferentes polígonos como obstáculos, incluindo formas côncavas;
- Dividir também o mapa em polígonos genéricos e não apenas retângulos;
- Checar se os pontos inicial e final já não estão livres de obstáculos considerando uma linha reta entre eles;
- Suavizar o caminho encontrado, caso isso seja importante para a aplicação.

De modo geral, podemos concluir que o sistema Navigation Mesh é muito satisfatória se a solução ótima não é estritamente necessária. A abordagem é, portanto, muito útil em diversas áreas de aplicação, em especial a robótica e a indústria de games.

Referências

[1] Cormen, T. H. *Introduction to Algorithms, 3rd Edition.* The MIT Press, United States, 2009.

[2] from Red Blog Games, A. A*'s use of the heuristic, 2011.

[3] Fry, B. Language reference (api) / processing 2+.

[4] Naveh, B. Jgrapht, 2015.

[5] Robert Sedgewick, K. W. Algorithms, 4th edition, 2015.

[6] Russell, S. *Artificial Intelligence: A Modern Approach.* Pearson, United States, 2009.

6 Apêndice

O código completo e comentado pode ser encontrado no link: `https://github.com/gabrielilharco/PathFinder`.

A estrutra de pastas completa está representada abaixo.

```
pathfinder
├── algorithm
│   ├── AbstractPathFinder.java
│   ├── AbstractShortestPathFinder.java
│   ├── AStarSearch.java
│   ├── BestFirstSearch.java
│   ├── BreadthFirstSearch.java
│   ├── DepthFirstSearch.java
│   ├── DijkstraShortestPath.java
│   ├── Heuristic.java
│   ├── IAlgorithm.java
│   ├── MapAreaDivider.java
│   ├── RectanglePathToEdgePoints.java
│   ├── RecursiveDepthFirstSearch.java
│   └── WaypointAlgorithm.java
├── draw
│   ├── DisplayFrame.java
│   ├── FileDrawer.java
│   ├── GUIDrawer.java
│   ├── IDrawer.java
│   ├── MainPApplet.java
│   └── WaypointDrawer.java
├── graphicInterface
│   ├── Backgrounds.java
│   ├── ChooseAlgorithmJPanel.java
│   ├── CreateMapJPanel.java
│   ├── ImageButton.java
│   ├── ImagePanel.java
│   ├── MainJFrame.java
│   ├── MainJPanel.java
│   ├── buttonListeners:
│   ├── ChooseAlgorithmButtonListener.java
│   ├── CreateMapButtonListener.java
│   └── MainButtonListener.java
├── main
│   └── Main.java
└── representations
    └── graph
```

```
    │   ├── AbstractUndirectedGraph.java
    │   ├── AdjacencyList.java
    │   ├── Graph.java
    │   ├── IGraph.java
    │   ├── Path.java
    │   └── WeightedGraph.java
    ├── maps
    │   ├── GridMap.java
    │   └── VertexMap.java
    └── primitives
        ├── Edge.java
        ├── LineInfo.java
        ├── Point.java
        ├── Rectangle.java
        └── Vertex.java
├── statistics
│   └── Benchmark.java
└── utils
    ├── graphCreator
    │   ├── AbstractGraphCreator.java
    │   ├── GridGraphCreator.java
    │   └── WaypointGraphCreator.java
    ├── gridMapCreator
    │   ├── AbstractGridMapCreator.java
    │   ├── GUIGridMapCreator.java
    │   ├── ImageGridMapCreator.java
    │   ├── RectangleGridMapCreator.java
    │   └── TextFileGridMapCreator.java
    ├── vertexMapCreator
    │   ├── AbstractVertexMapCreator.java
    │   └── TextFileVertexMapCreator.java
    ├── ConfigManager.java
    └── Pair.java
```