Relatório Buscas Heurísticas

Alunos:

Guilherme Henrique Lourenço dos Santos, RA: 1305069.
Haroldo Shigueaki Teruya, RA: 1305301.

O trabalho foi desenvolvido utilizando:

- HTML5;
- Biblioteca JavaScript JQuery;
- CSS3 com SASS;
- Biblioteca JavaScript D3;

Para a visualização do tabuleiro foi implementada a “classe” “TapesSimulator” com o apoio da biblioteca D3. Para embaralhar o tabuleiro a função “sortBlocks” é chamada. A resolução de forma aleatória é feita pela função “randomSolution”. E as buscas heurística 1, heurística 2 e heurística pessoal são implementadas respectivamente pelas funções “heuristic1Process”, “heuristic2Process” e “heuristicCustomProcess”.

- TapesSimulator:
  - Leva dois argumentos, o primeiro para identificar em qual div será adicionada imagem resultado e a segunda deve ser um vetor de vetores como:
    - “[[0, 1, 2], [3, 4, 5], [6, 7, 8]]”
  - Código:

```
var TapesSimulator = function(chart, tapes){
    var cellSize = 100;
    var array_size = 5;
    var width = 500;
    d3.select(chart).html("");
    this.svg = d3.select(chart).append("svg")
        .attr("height", 400)
    this.g = this.svg.append("g");
        //.call(zoom);
    this.g_tape = this.g.selectAll("g")
        .data(tapes)
        .enter().append("g")
        .attr(
            "transform", function(d, i) {
                return "translate(0, "+ i * cellSize + ")";
            });

    this.g_rect = this.g_tape.selectAll("g")
        .data(function(d,i) {
            return d;
        })
```

```
            .enter().append("g")
            .attr("class", "tape")
            .attr(
                "transform", function(d, i) {
                    return "translate(" + i * cellSize + ",0)";
                });

        g_rect.append("rect")
            .attr("class", function(d, i){
                if (d == ""){
                    return "cur_element";
                }
            })
            .attr("width", cellSize)
            .attr("height", cellSize);
        g_rect.append("text")
            .attr("x", function(d) { return (cellSize/2) -3; })
            .attr("y", cellSize / 2)
            .attr("dy", ".35em")
            .text(function(d) { return d });
    };
```

- sortBlocks:
    - Leva um argumento sendo o número de movimentos para embaralhar
    - Código:

```
function sortBlocks(times)
{
        setTimeout(function()
        {
                lock = true;
                shuffleBlocks();
                if(--times)sortBlocks(times);
                else
                {
                        _moviments = 0;
                        refreshMovimentsLabel();
                        lock = false;
                }
        }, 50)
}
```

- randomSolution:
    - Leva um argumento sendo o número de movimentos para tentar encontrar a solução. Para quando encontrar a solução ou esgotam-se os movimentos

- Código:

```
function randomSolution(times)
{
	setTimeout(function()
	{
		shuffleBlocks();
		if(--times && !checkBlocks())randomSolution(times);
	}, 1)
}
```

- heuristic1Process:
  - Busca heurística em uma camada:
    - openList = lista dos possíveis movimentos a serem feitos.
    - closeList = lista dos movimentos já realizados. Utilizado para evitar “loops”.
    - createNode: método para criar o nó inicial.
    - backtracking: está variável é utilizado para sinalizar se voltamos uma jogada ou não.
  - Código:

```
function heuristic1Process()
{
	_animationInterval = 1;
	_moviments = 0;
	openList = null;
	openList = [];
	closeList = null;
	closeList = [];
	createNode(parent, _blocks);

	heuristic1(false);
}

function heuristic1(backtracking)
{
	// if no node is in the openList
	if( openList.length <= 0 ) return;

	// get the last elment of the openList
	node = openList.pop();

	if( node == null ) return;

	// verify if the current node (position of the blocks) is complete (right positions).
	if(isComplete(node)) return;
```

```
	// doing a movement
	_moviments++;

	// have to verify if it is not backtracked to not create repetitives nodes
(movements)
	if (!backtracking)
	{
		// create possible movements (child nodes)
		createChildNodes(node);

		// store the current node (movement), used to avoid loops
		closeList.push(node);
	}

	backtracking = false;

	// we have to verify if the current node has onlu one way (child nodes)
	var hasOneWay = false;

	// verify if we have a repretition of movement (loop of nodes) for each
child node of the current node (possible movements)
	for(var j = 0; j < node.getChildNodes().length; j++)
	{
		var repeated = false;
		if( !node.getChildNodes()[j].getVisited() )
		{
			for(var i = 0; i < closeList.length; i++)
			{
				repeated = true;
				for(var x = 0; x < 3 && repeated; x++)
				{
					for(var y = 0; y < 3 && repeated; y++)
					{

if(node.getChildNodes()[j].getBlocks()[x][y] != closeList[i].getBlocks()[x][y])
						{
							repeated = false;
						}
					}
				}
				if(repeated)break;
			}

			// verified if exist a repretition, if not, advance to the
next move (node child).
```

```
                        if(!repeated)
                        {
                                hasOneWay = true;

                                // advance a movement node (child).
                                openList.push(node.getChildNodes()[j]);
                                break;
                        }
                }
        }

        // if not exist child node (movement) possible.
        if (!hasOneWay)
        {
                // set node visited (to not repeat this movement)
                node.setVisited(true);

                // we have to back to parent (backtracking)
                openList.push(node.getParent());

                backtracking = true;
        }

        heuristic1(backtracking);
}
```

- heuristic2Process:
  - Busca heurística em duas camadas:
    - openList = lista dos possíveis movimentos a serem feitos.
    - closeList = lista dos movimentos já realizados. Utilizado para evitar “loops”.
    - createNode: método para criar o nó inicial.
    - backtracking: está variável é utilizado para sinalizar se voltamos uma jogada ou não.
  - Código:

```
function heuristic2Process()
{
        _moviments = 0;
        openList = null;
        openList =[];
        closeList = null;
        closeList = [];
        createNode(parent, _blocks);

        heuristic2(false);
}
```

```
function heuristic2(backtracking)
{
	// if no node is in the openList
	if( openList.length <= 0 ) return;

	// get the last elment of the openList
	node = openList.pop();

	if( node == null ) return;

	// verify if the current node (position of the blocks) is complete (right
positions).
	if(isComplete(node)) return;

	// doing a movement
	_moviments++;

	// have to verify if it is not backtracked to not create repetitives nodes
(child nodes)
	if (!backtracking)
	{
		// store the current node (movement), used to avoid loops
		closeList.push(node);

		// create possible movements (child nodes)
		createChildNodes(node);

		// In this for, creating a second layer of possible movemets of
the possible movements (child nodes for each child node)
		for(var i = 0; i < node.getChildNodes().length; ++i)
		{
			createChildNodes(node.getChildNodes()[i]);
		}

		// temporary node list, this node list will be the possible
movements of the current node
		var temporaryNodeList = [];

		// populating the temporaryNodeList with the possible next
movements of the next movements (second layer)
		for(var i = 0; i < node.getChildNodes().length; ++i)
		{
			for(var j = 0; j <
node.getChildNodes()[i].getChildNodes().length; j++)
			{
```

```
node.getChildNodes()[i].getChildNodes()[j].setParent(node);

temporaryNodeList.push(node.getChildNodes()[i].getChildNodes()[j]);
                        }
                }
                node.setChildNodes(temporaryNodeList);
        }

        backtracking = false;

        // we have to verify if the current node has onlu one way (child nodes)
        var hasOneWay = false;

        // verify if we have a repretition of movement (loop of nodes) for each
child node of the current node (possible movements)
        for(var j = 0; j < node.getChildNodes().length; j++)
        {
                var repeated = false;
                if( !node.getChildNodes()[j].getVisited() )
                {
                        for(var i = 0; i < closeList.length; i++)
                        {
                                repeated = true;
                                for(var x = 0; x < 3 && repeated; x++)
                                {
                                        for(var y = 0; y < 3 && repeated; y++)
                                        {

if(node.getChildNodes()[j].getBlocks()[x][y] != closeList[i].getBlocks()[x][y])
                                                {
                                                        repeated = false;
                                                }
                                        }
                                }
                                if(repeated)break;
                        }

                        // verified if exist a repretition, if not, advance to the
next move (node child).
                        if(!repeated)
                        {
                                hasOneWay = true;

                                // advance a movement node (child).
                                openList.push(node.getChildNodes()[j]);
```

```
                                    break;
                                }
                            }
                        }

                        // if not exist child node (movement) possible.
                        if (!hasOneWay)
                        {
                            // set node visited (to not repeat this movement)
                            node.setVisited(true);

                            // we have to back to parent (backtracking)
                            openList.push(node.getParent());

                            backtracking = true;
                        }

                        heuristic2(backtracking);
                    }
```

- heuristicCustomProcess:
    - Busca heurística em três camadas inspirada na heurística 2:
        - openList = lista dos possíveis movimentos a serem feitos.
        - closeList = lista dos movimentos já realizados. Utilizado para evitar “loops”.
        - createNode: método para criar o nó inicial.
        - backtracking: está variável é utilizado para sinalizar se voltamos uma jogada ou não.
    - Código:

```
function heuristicCustomProcess(){
    _moviments = 0;
    openList = null;
    openList =[];
    closeList = null;
    closeList = [];
    createNode(parent, _blocks);

    heuristicCustom(false);
}

function heuristicCustom(backtracking)
{
    // if no node is in the openList
    if( openList.length <= 0 ) return;
```

```
		// get the last elment of the openList
		node = openList.pop();

		if( node == null ) return;

		// verify if the current node (position of the blocks) is complete (right
positions).
		if(isComplete(node)) return;

		// doing a movement
		_moviments++;

		// have to verify if it is not backtracked to not create repetitives nodes
(child nodes)
		if (!backtracking)
		{
			// store the current node (movement), used to avoid loops
			closeList.push(node);

			// create possible movements (child nodes)
			createChildNodes(node);

			// In this for, creating a second layer of possible movemets of
the possible movements (child nodes for each child node)
			for(var i = 0; i < node.getChildNodes().length; ++i)
			{
				createChildNodes(node.getChildNodes()[i]);

				// In this for, creating a third layer of possible movemets
of the second layer
				for(var j = 0; j <
node.getChildNodes()[i].getChildNodes().length; j++)
				{

createChildNodes(node.getChildNodes()[i].getChildNodes()[j]);
				}
			}

			// temporary node list, this node list will be the possible
movements of the current node
			var temporaryNodeList = [];

			// populating the temporaryNodeList with the possible next
movements of the next movements of the next movements (third layer)
			for(var i = 0; i < node.getChildNodes().length; ++i)
```

```
                {
                        for(var j = 0; j <
node.getChildNodes()[i].getChildNodes().length; j++)
                        {
                                for( var k = 0; k <
node.getChildNodes()[i].getChildNodes()[j].getChildNodes().length; k++ )
                                {

node.getChildNodes()[i].getChildNodes()[j].getChildNodes()[k].setParent(node
);

temporaryNodeList.push(node.getChildNodes()[i].getChildNodes()[j].getChild
Nodes()[k]);
                                }
                        }
                }
                node.setChildNodes(temporaryNodeList);
        }

        backtracking = false;

        // we have to verify if the current node has onlu one way (child nodes)
        var hasOneWay = false;

        // verify if we have a repretition of movement (loop of nodes) for each
child node of the current node (possible movements)
        for(var j = 0; j < node.getChildNodes().length; j++)
        {
                var repeated = false;
                if( !node.getChildNodes()[j].getVisited() )
                {
                        for(var i = 0; i < closeList.length; i++)
                        {
                                repeated = true;
                                for(var x = 0; x < 3 && repeated; x++)
                                {
                                        for(var y = 0; y < 3 && repeated; y++)
                                        {

if(node.getChildNodes()[j].getBlocks()[x][y] != closeList[i].getBlocks()[x][y])
                                                {
                                                        repeated = false;
                                                }
                                        }
                                }
                                if(repeated)break;
```

```
                        }

                        // verified if exist a repretition, if not, advance to the
next move (node child).
                        if(!repeated)
                        {
                                hasOneWay = true;

                                // advance a movement node (child).
                                openList.push(node.getChildNodes()[j]);
                                break;
                        }
                }
        }

        // if not exist child node (movement) possible.
        if (!hasOneWay)
        {
                // set node visited (to not repeat this movement)
                node.setVisited(true);

                // we have to back to parent (backtracking)
                openList.push(node.getParent());

                backtracking = true;
        }

        heuristicCustom(backtracking);
}
```