Práctica 2: Resolución de problemas y búsqueda

Javier Herrer Torres

Octubre 2020

1. Introducción

El trabajo consiste básicamente en la realización de experimentos y recopilación de información relevante de la búsqueda relacionada con aspectos de eficiencia. Se realizarán con el 8-puzzle búsquedas ciegas (BFS e IDS) y búsquedas informadas A* con las heurísticas de fichas descolocadas y Manhattan. Se utilizará la búsqueda A*, y se comparará la eficiencia de los algoritmos mostrando el número de nodos generados y el factor de ramificación efectivo b* para distintas profundidades.

2. Tareas realizadas

2.1. Métrica de número de nodos generados

Se ha añadido la métrica de nodos generados en las clases NodeExpander en aima.core.search.framework, e IterativeDeepeningSearch tomando como modelo la métrica de nodos expandidos. En la clase NodeExpander se ha hecho uso de una nueva métrica nodesGenerated que será incrementada al final de cada iteración del bucle for del método expandNode. En la clase IterativeDeepeningSearch se realiza algo similar en el método search para iteración del bucle.

```
/**
 * Returns the children obtained from expanding the specified node
 * in the specified problem.
 *
 * @param node
 *            the node to expand
 * @param problem
 *            the problem the specified node is within.
 *
 * @return the children obtained from expanding the specified node
 * in the specified problem.
 */
public List<Node> expandNode(Node node, Problem problem) {
```

```
    List<Node> childNodes = new ArrayList<Node>();

    ActionsFunction actionsFunction =
        problem.getActionsFunction();
    ResultFunction resultFunction =
        problem.getResultFunction();
    StepCostFunction stepCostFunction =
        problem.getStepCostFunction();

    for (Action action :
            actionsFunction.actions(node.getState())) {
        Object successorState =
            resultFunction.result(
                node.getState(), action);
        double stepCost =
            stepCostFunction.c(
                node.getState(), action, successorState);
        childNodes.add(
            new Node(
                successorState, node, action, stepCost));
        metrics.set(
            METRIC_NODES_GENERATED,
            metrics.getInt(METRIC_NODES_GENERATED) + 1);
    }
    metrics.set(
        METRIC_NODES_EXPANDED,
            metrics.getInt(METRIC_NODES_EXPANDED) + 1);

    return childNodes;
}
```

2.2. Implementación del método de bisección

La implementación de la clase Biseccion se incluyó en aima.core.util.math. Permite obtener los ceros de la función de bisección por aproximaciones sucesivas, donde b* es el factor de ramificación efectivo, N el número de nodos generados y d la profundidad de la solución:

$$0 = \frac{b^*(1-(b^*)^d)}{1-b^*} - N$$

2.3. Generación de experimentos

Se han generado 100 experimentos aleatorios de la profundidad deseada y calculado la media de los nodos generados. Se ha utilizado la clase *GenerateInitialEightPuzzleBoard* suministrada para generar aleatoriamente estados iniciales

y estados finales de la profundidad deseada. El problema del método *random* es que ejecuta acciones aleatorias desde el estado inicial, donde d es la profundidad deseada sin generar estados repetidos, pero el que se hayan dado d pasos al estado final no garantiza que no haya caminos más cortos al estado final. Por lo que antes de pasar al siguiente experimento, se comprueba si la solución óptima es del coste deseado. En caso de no cumplirse, se repetirá el experimento hasta que se cumpla. Únicamente se incluirán en la media los experimentos válidos.

La figura 1 corresponde al resultado de la ejecución de EightPuzzlePract2.java. Se puede observar que para el algoritmo de búsqueda IDS, a partir de la profundidad d=10, se cancelan los siguientes experimentos debido al alto coste (se representa mediante «0»).

```
public static void experimento(String searchName, int prof_ini,
    int profundidad, int numExp, long[] resultados,
    double[] effectiveB) {
    Search search = null;

    double mediaNodosGenerados = 0;
    double mediaNodosExpandidos = 0;
    int experimento = 0;

    System.out.println(searchName);

    if (searchName.compareTo("IDS") == 0 && profundidad > 10) {
        profundidad = 10;
    }

    for (int pasos = prof_ini; pasos <= profundidad; pasos++) {
        experimento = 1;
        while (experimento <= numExp) {
            EightPuzzleBoard initialState = new EightPuzzleBoard(
                GenerateInitialEightPuzzleBoard.randomIni());
            EightPuzzleBoard finalState =
                GenerateInitialEightPuzzleBoard.random(
                    pasos,
                    new EightPuzzleBoard(initialState));
            EightPuzzleGoalTest.setGoalState(
                new EightPuzzleBoard(finalState));

            if (searchName.compareTo("BreathFirstSearch") == 0) {
                search = new BreadthFirstSearch();
            } else if (searchName.compareTo("IDS") == 0) {
                search = new IterativeDeepeningSearch();
            } else if (searchName.compareTo("A* misplaced") == 0) {
                search = new AStarSearch(
                    new GraphSearch(),
```

```
                    new MisplacedTilleHeuristicFunction2());
            } else if (searchName.compareTo("A* Manhatan") == 0) {
                search = new AStarSearch(
                    new GraphSearch(),
                    new ManhattanHeuristicFunction2());
            }

            eightPuzzleSearch(search, initialState, finalState,
                "Busqueda " + searchName);

            if (_depth == pasos) {
                mediaNodosGenerados += _generatedNodes;
                mediaNodosExpandidos += _expandedNodes;

                experimento++;
            }
        }

        resultados[pasos] = Math.round(mediaNodosGenerados/numExp);
        Biseccion bf = new Biseccion();
        bf.setDepth(_depth);
        bf.setGeneratedNodes((int)resultados[pasos]);

        effectiveB[pasos] =
            bf.metodoDeBiseccion(1.000000000000001, 4, 1E-10);
        mediaNodosExpandidos = Math.round(mediaNodosExpandidos/numExp);

        System.out.format("Depth: %3d %6d Nodos b*: %f3.2\n",  _depth,
            resultados[pasos], effectiveB[pasos]);
    }
}
```

2.4. Cambios en clases de heurísticas

Las heurísticas *ManhattanHeuristicFunction* y *MisplacedTilleHeuristicFunction* están pensadas para un único estado objetivo. Es por ello que se han realizado los siguientes cambios:

- Se ha modificado la clase *EightPuzzleGoalTest* haciendo la variable *goal* estática, y añadiendo los métodos *setGoalState* y *getGoalState*.
- La clase *EightPuzzleBoard* extiende de *EightPuzzleGoalTest* para poder acceder a *goal* fácilmente.
- Nuevas clases *ManhattanHeuristicFunction2* y *MisplacedTilleHeuristicFunction2* que consideran para cada experimento el estado final correspondiente.

	Nodos Generados				b*			
d	BFS	IDS	A*h(1)	A*h(2)	BFS	IDS	A*h(1)	A*h(2)
2	8	11	6	5	2,37	2,85	2,00	1,79
3	26	44	15	14	2,56	3,14	2,06	2,00
4	64	143	28	25	2,51	3,15	1,96	1,89
5	134	425	45	39	2,40	3,11	1,85	1,79
6	257	1207	70	58	2,29	3,05	1,78	1,71
7	474	3374	105	81	2,21	3,01	1,72	1,65
8	852	9490	155	109	2,15	2,99	1,68	1,59
9	1471	27321	226	146	2,09	2,97	1,65	1,55
10	2502	78465	341	195	2,05	2,96	1,63	1,53
11	4151	0	511	257	2,00	0,00	1,62	1,50
12	6808	0	773	343	1,97	0,00	1,61	1,48
13	11106	0	1175	462	1,94	0,00	1,60	1,47
14	18094	0	1803	631	1,91	0,00	1,59	1,46
15	29330	0	2795	837	1,89	0,00	1,59	1,45
16	46665	0	4267	1129	1,87	0,00	1,58	1,44
17	74154	0	6677	1511	1,85	0,00	1,58	1,43
18	115493	0	10286	2078	1,83	0,00	1,58	1,43
19	179364	0	15720	2800	1,81	0,00	1,58	1,42
20	270535	0	24641	3687	1,79	0,00	1,58	1,42
21	400145	0	37591	4887	1,78	0,00	1,57	1,41
22	574511	0	58729	6720	1,76	0,00	1,57	1,41
23	803385	0	89576	9181	1,74	0,00	1,57	1,41
24	1091888	0	136588	12221	1,72	0,00	1,57	1,41

Figura 1: Resultado de la ejecución de EightPuzzlePract2.java

3. Conclusiones

El factor de ramificación efectivo b* mide la calidad de la heurística. Es una buena medida de la utilidad de la heurística. Un buen valor de b* es 1.

De la figura 1 podemos obtener que de las dos heurísticas h(1) y h(2), siendo la primera el número de piezas en posición errónea, y la segunda la distancia Manhattan; la heurística h(2) es mejor que la h(1).