**Laporan Tugas Kecil 3  
Strategi Algoritma**

Dibuat Oleh:

Andi Marihot Sitorus

13522138

# **Analisis**

Program ini dibuat dengan konsep priority queue dan juga graf. Pertama, file txt yang berisi kata dalam bahasa Inggris akan disortir menjadi file-file yang berisi kata dengan panjang yang sama. Setiap kata memiliki jumlah tetangga dan seluruh tetangganya dibawah kata tersebut. Tetangga dari sebuah kata merupakan kata lain yang hanya berbeda satu huruf saja, sesuai dengan konsep permainan word ladder. Kata disimpa dalam sebuah Node yang berisi kata, tetangga(children), dan juga cost yang disesuaikan untuk tiap algoritma. Pencarian *path* akan dimulai dari mengecek tetangga dari startWord yang dimasukkan ke priority queue. Pencarian akan berhenti jika priority queue kosong (tidak ada *path* yang ditemukan) atau elemen priority queue yang di cek berisi endWord.

Hal yang membedakan ketiga algoritma adalah cost yang dijadikan acuan dalam priority queue. Setiap kata yang memiliki cost lebih rendah tidak akan dieksplorasi ulang. Dalam UCS, costnya merupakan banyak huruf yang berbeda pada elemen priority queue dengan startWord. Contohnya, jika word adalah startWord, maka plan akan memiliki cost sebesar 4. Dalam Greedy Best First Search, costnya merupakan kesamaan kata pada elemen priority queue dengan endWord sehingga mengambil langkah terbaik setiap saat. Dalam A\*, costnya merupakan perbedaan antara startWord dengan endWord dikurangi dengan perbedaan startWord dengan elemen priority queue ditambah kesamaan elemen priority queue dengan endWord. Lebih jelasnya :

A\*cost = diffCount(startWord,endWord)-diffCount(Elmnt,startWord)+sameCount(Elmnt, endword)

Dengan begitu, akan didapatkan kata yang paling dekat dengn startWord dan juga endWord secara bersamaan.

Definisi f(n) dan g(n) biasanya merujuk pada fungsi evaluasi yang digunakan dalam algoritma pencarian seperti A\* atau UCS. f(n) adalah nilai total dari g(n) (biaya yang telah dikeluarkan untuk mencapai simpul saat ini) dan h(n) (heuristik yang memberikan perkiraan biaya dari simpul saat ini ke tujuan). Sedangkan g(n) adalah biaya yang telah dikeluarkan untuk mencapai simpul saat ini. Ini sering digunakan untuk menghitung biaya yang telah dikeluarkan dari simpul awal ke simpul saat ini dalam algoritma pencarian.

Heuristik yang digunakan pada algoritma A\* dikatakan admissible jika tidak pernah melebih biaya sebenarnya untuk mencapai tujuan. Dengan kata lain, heuristik admissible selalu memperkirakan biaya yang tidak pernah melebihi biaya sebenarnya. Ini sesuai dengan definisi admissible dari salindia kuliah yang menekankan bahwa heuristik harus konservatif dan tidak memperkirakan biaya yang lebih tinggi dari yang sebenarnya.

Pada kasus word ladder, algoritma UCS (Uniform Cost Search) dan BFS (Breadth-First Search) berbeda dalam prinsip kerja mereka meskipun keduanya menghasilkan urutan node dan path yang sama. UCS mempertimbangkan biaya yang berbeda untuk setiap langkah, sementara BFS memperlakukan semua langkah dengan biaya yang sama. Namun, dalam hal menghasilkan urutan node dan path, keduanya akan sama.

Secara teoritis, algoritma A\* lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder karena A\* menggunakan heuristik untuk memandu pencarian, yang memungkinkan algoritma untuk menghindari penelusuran jalur yang tidak produktif. Ini berarti A\* dapat menemukan solusi dengan mempertimbangkan lebih sedikit node dibandingkan dengan UCS dalam kasus word ladder.

Secara teoritis, algoritma Greedy Best First Search tidak menjamin solusi optimal untuk persoalan word ladder. Meskipun Greedy Best First Search memilih simpul yang paling dekat dengan tujuan saat ini, itu tidak mempertimbangkan biaya total dari simpul awal ke tujuan. Oleh karena itu, algoritma ini tidak dapat menjamin bahwa solusi yang dihasilkan adalah yang terbaik secara keseluruhan.

# **Source Code**

## **Algoritma UCS**

1. import java.util.\*;
2. public class UCS {
3. public static Map<List<String>, Integer> findPath(List<Node> nodes, Node startNode, Node endNode) {
4. String startWord = startNode.getWord();
5. PriorityQueue<Node> prioQue = new PriorityQueue<>((n1, n2) -> Integer.compare(n1.UCScost(startWord), n2.UCScost(startWord)));
6. Map<String, Integer> explored = new HashMap<>();
7. Map<String, String> parentMap = new HashMap<>();
8. prioQue.offer(startNode);
9. while (!prioQue.isEmpty()) {
10. Node current = prioQue.poll();
11. String currentWord = current.getWord();
12. int currentCost = current.UCScost(startWord);
13. if (currentWord.equals(endNode.getWord())) {
14. Map<List<String>, Integer> ret = new HashMap<List<String>, Integer>();
15. ret.put(constructPath(parentMap,startNode, endNode), explored.size());
16. return ret;
17. }
18. if (!explored.containsKey(currentWord) || currentCost > explored.get(currentWord)){ {
20. explored.put(currentWord, currentCost);
21. for (String child : current.getChildren()) {
22. Node currentChild = Tools.getNode(child, nodes);
23. int newCost = currentChild.UCScost(startWord);
24. if (!explored.containsKey(child) || newCost > explored.get(child)) {
25. prioQue.offer(currentChild);
26. if(!parentMap.containsKey(currentChild.getWord()))    {
27. parentMap.put(child, currentWord);
28. }
29. }
30. }
31. }
32. }
33. }
34. return new HashMap<>();
35. }
36. private static List<String> constructPath(Map<String, String> parentMap,Node startNode, Node endNode) {
37. List<String> path = new ArrayList<>();
38. String currentWord = endNode.getWord();
39. while (currentWord != null) {
40. path.add(0, currentWord);
41. currentWord = parentMap.get(currentWord);
42. }
43. return path;
44. }
46. }

## **Algoritma Greedy Best First Search**

1. import java.util.\*;
2. public class GBFS {
3. public static Map<List<String>, Integer> findPath(List<Node> nodes, Node startNode, Node endNode) {
4. PriorityQueue<Node> prioQue = new PriorityQueue<>((n1, n2) -> Integer.compare(heuristic(n1, endNode), heuristic(n2, endNode)));
5. Map<String, Integer> explored = new HashMap<>();
6. Map<String, String> parentMap = new HashMap<>();
7. prioQue.offer(startNode);
9. while (!prioQue.isEmpty()) {
10. Node current = prioQue.poll();
11. String currentWord = current.getWord();
12. int currentCost = heuristic(current, endNode);
13. if (currentWord.equals(endNode.getWord())) {
14. Map<List<String>, Integer> ret = new HashMap<>();
15. ret.put(constructPath(parentMap, startNode, endNode), explored.size());
16. return ret;
17. }
18. if (!explored.containsKey(currentWord) || currentCost > explored.get(currentWord)) {
19. explored.put(currentWord, currentCost);
20. for (String child : current.getChildren()) {
21. Node currentChild = Tools.getNode(child, nodes);
22. int newCost = heuristic(currentChild, endNode);
23. if (!explored.containsKey(child) || newCost > explored.get(child)) {
24. prioQue.offer(currentChild);
25. if (!parentMap.containsKey(currentChild.getWord())) {
26. parentMap.put(child, currentWord);
27. }
28. }
29. }
30. }
31. }
32. return new HashMap<>();
33. }
34. private static List<String> constructPath(Map<String, String> parentMap, Node startNode, Node endNode) {
35. List<String> path = new ArrayList<>();
36. String currentWord = endNode.getWord();
37. while (currentWord != null) {
38. path.add(0, currentWord);
39. currentWord = parentMap.get(currentWord);
40. }
41. return path;
42. }
44. private static int heuristic(Node node, Node endNode) {
45. return node.GBFScost(endNode.getWord());
46. }
47. }

## **Algoritma A\***

import java.util.\*;

public class Astar {

    public static Map<List<String>, Integer> findPath(List<Node> nodes, Node startNode, Node endNode) {

        String startWord = startNode.getWord();

        String endWord = endNode.getWord();

        PriorityQueue<Node> prioQue = new PriorityQueue<>((n1, n2) -> Integer.compare(n1.Astarcost(startWord,endWord), n2.Astarcost(startWord,endWord)));

        Map<String, Integer> explored = new HashMap<>();

        Map<String, String> parentMap = new HashMap<>();

        prioQue.offer(startNode);

        while (!prioQue.isEmpty()) {

            Node current = prioQue.poll();

            String currentWord = current.getWord();

            int currentCost = current.Astarcost(startWord, endWord);

            if (currentWord.equals(endWord)) {

                Map<List<String>, Integer> ret = new HashMap<>();

                ret.put(constructPath(parentMap, startWord, endWord), explored.size());

                return ret;

            }

            try{

            if (!explored.containsKey(currentWord) || currentCost > explored.get(currentWord)) {

                explored.put(currentWord, currentCost);

                for (String child : current.getChildren()) {

                    Node currentChild = Tools.getNode(child, nodes);

                    int newCost = currentChild.Astarcost(startWord, endWord); // cost so far

                    if (!explored.containsKey(child) || newCost > explored.get(child)) {

                        prioQue.offer(currentChild);

                        if (!parentMap.containsKey(currentChild.getWord())) {

                            parentMap.put(child, currentWord);

                        }

                    }

                }

            }} catch(Exception e){}

        }

        return new HashMap<>();

    }

    private static List<String> constructPath(Map<String, String> parentMap, String startWord, String endWord) {

        List<String> path = new ArrayList<>();

        String currentWord = endWord;

        while (currentWord != null && !currentWord.equals(startWord)) {

            path.add(0, currentWord);

            currentWord = parentMap.get(currentWord);

        }

        if (currentWord != null && currentWord.equals(startWord)) {

            path.add(0, currentWord);

        }

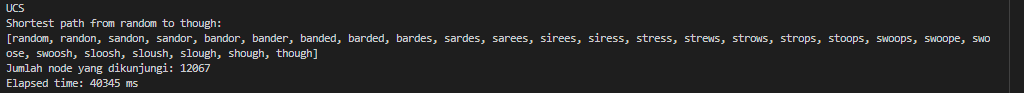
        return path;

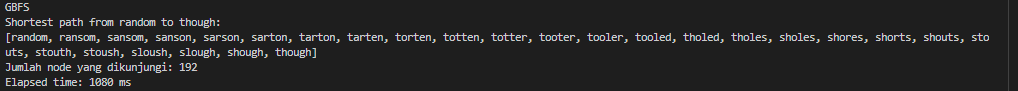
    }

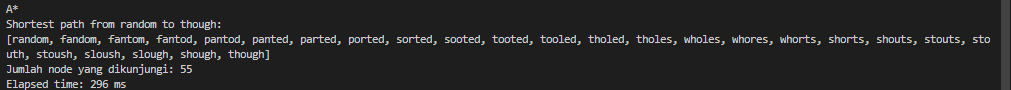
}

# **Test**

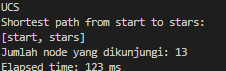
## **Start: random, End : though**

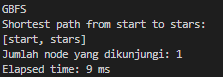


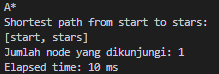




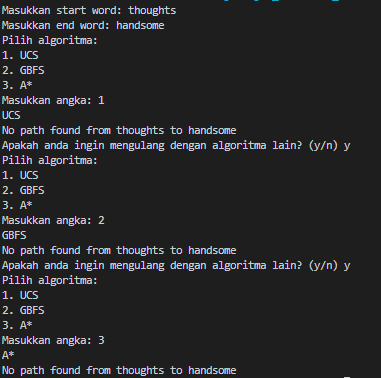
## **Start: Start, End: Stars**



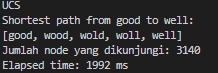


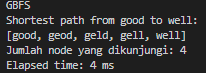


## **Start: thoughts, End: handsome**



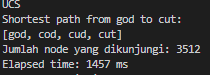
## **Start : good, End: well**

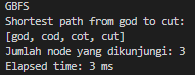


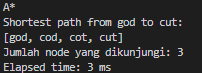




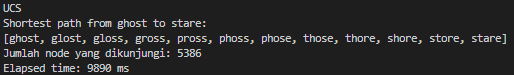
## **Start: god, End: cut**

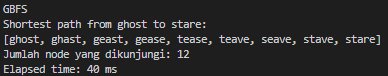


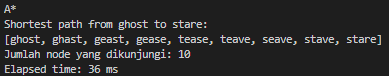




## **Start: ghost, End: stare**







# **Perbandingan solusi UCS, GBFS, dan A\***

Algoritma yang paling optimal merupakan A\*, lalu diikuti GBFS, dan yang terakhir UCS. Pada bagian test, dapat dilihat bahwa UCS seringkali memberikan solusi yang lebih panjang. Lalu pada test pertama, A\* memberikan solusi yang lebih optimal daripada GBFS. Namun pada panjang kata yang pendek dan perbedaan start dan end yang sedikit, A\* dan GBFS sama optimalnya. Hal itu disebabkan oleh jalur pencarian yang semakin rumit apabila panjang kata dan perbedaan start ke end semakin besar.

Waktu eksekusi yang paling lama adalah UCS karena menelusuri node-node yang tidak diperlukan. Sedangkan untuk A\* dan GBFS, pada panjang kata rendah dan perbedaan start ke end sedikit, A\* dan GBFS menunjukkan waktu eksekusi yang hampir sama. Namun pada test 1 yang memiliki kata yang cukup panjang dan perbedaan start dan end yang besar, A\* menjadi jauh lebih cepat dari GBFS.

Memori yang digunakan algoritma UCS paling besar karena menelusuri node-node yang tidak diperlukan. Lalu diikuti oleh GBFS dan yang paling cepat adalah A\*. GBFS menggunakan memori yang lebih besar karena hanya menggunakan endWord sebagai acuannya sehingga kadang dapat memasuki node yang kurang optimal. Sedangkan A\*, mempertimbangkan endWord sekaligus dengan startWord, sehingga penggunaan memorinya sangat optimal.

# **Pranala ke Repository**

<https://github.com/MelonSeed9/Tucil3_13522138.git>

