LAPORAN
TUGAS BESAR STRATEGI ALGORITMA

"Perbandingan Algoritma *backtracking* dan Dynamic Programming dalam Menyelesaikan Permainan *Rubik's Cube*"

Disusun Oleh:

IF-47-EXT

Vitria Anggraeni 103012380501
Fahri Alfiansyah 103012380508
Gid Achmad Ahlul Fadli 103012380509

PROGRAM STUDI S1 INFORMATIKA
FAKULTAS INFORMATIKA
UNIVERSITAS TELKOM
BANDUNG
2024

Pembagian Tugas

Vitria Anggraeni	Dokumentasi dan penyusunan laporan
Fahri Alfiansyah	Analisis dan perbandingan algoritma
Gid Achmad Ahlul Fadli	Implementasi algoritma *backtracking* dan *dynamic programming*

1. Deskripsi Studi Kasus

Permainan *Rubik's Cube* adalah teka-teki mekanis yang menantang pemain untuk memutar-mutar sisi-sisinya hingga setiap sisi memiliki satu warna seragam. Menyelesaikan permainan ini membutuhkan strategi yang efektif dan efisien. Dalam tugas besar ini, kami membandingkan dua algoritma, *backtracking* dan *dynamic programming*, untuk menyelesaikan permainan *Rubik's Cube* 2x2x2. Tujuan utama kami adalah untuk mengevaluasi keefektifan dan efisiensi kedua algoritma tersebut dalam menyelesaikan teka-teki ini.

2. Strategi Algoritma yang Dipilih

- **Algoritma *backtracking***

  Backtracking adalah teknik algoritma yang mencoba membangun solusi secara incremental. Dalam konteks *Rubik's Cube* 2x2x2, algoritma ini mencoba semua kemungkinan gerakan hingga menemukan solusi. Jika suatu gerakan tidak mengarah ke solusi, algoritma akan mundur (backtrack) dan mencoba gerakan lainnya. Berikut merupakan langkah-langkah penerapan dari algoritma *backtracking*:

  1. Mulai dari keadaan awal *Rubik's Cube*.
  2. Lakukan gerakan satu per satu.
  3. Setelah setiap gerakan, periksa apakah *Rubik's Cube* sudah selesai.
  4. Jika belum selesai, lanjutkan dengan mencoba gerakan berikutnya.
  5. Jika gerakan tersebut tidak mengarah ke solusi, mundur dan coba gerakan lain.

  Pseudocode algoritma *backtracking*:

Algorithm 1: Solve Rubik's Cube Using Backtracking

```
Input: cube: the initial state of the Rubik's Cube
Output: sequence of moves to solve the cube, or None if no solution is found
Function solve_rubiks_backtracking(cube):
    if is_solved(cube) then
        return []
    for move in all_possible_moves() do
        new_cube ← apply_move(cube, move);
        solution ← solve_rubiks_backtracking(new_cube);
        if solution is not None then
            return [move] + solution;
    return None;
```

- **Algoritma *dynamic programming***

Dynamic programming (DP) adalah teknik pemrograman yang menyelesaikan masalah kompleks dengan membaginya menjadi sub-masalah yang lebih kecil dan menyimpan hasil dari sub-masalah tersebut untuk menghindari perhitungan berulang. Untuk *Rubik's Cube*, DP dapat digunakan untuk menyimpan konfigurasi yang sudah ditemukan dan langkah-langkah yang diperlukan untuk mencapainya, sehingga mengurangi perhitungan berulang. Berikut merupakan langkah-langkah penerapan dari algoritma *dynamic programming*:

1. Definisikan tabel DP untuk menyimpan solusi sub-masalah.
2. Mulai dengan konfigurasi awal *Rubik's Cube* dan tambahkan ke tabel DP.
3. Untuk setiap konfigurasi, hitung semua kemungkinan gerakan berikutnya.
4. Simpan hasil gerakan dalam tabel DP.
5. Lanjutkan hingga menemukan solusi untuk konfigurasi yang diinginkan.

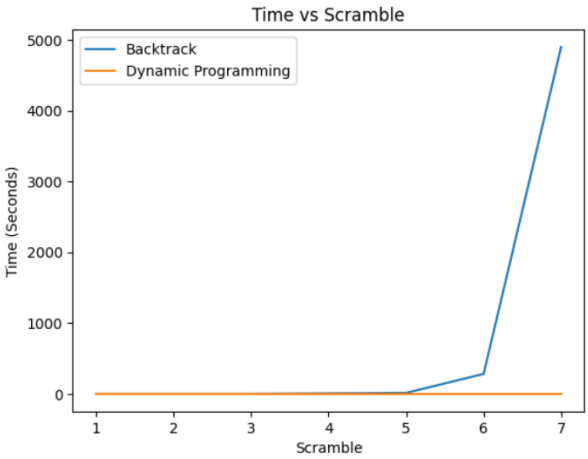
Pseudocode algoritma *dynamic programming*:

Algorithm 1: Solve Rubik's Cube Using Dynamic Programming

```
Input: cube: the initial state of the Rubik's Cube
Output: shortest sequence of moves to solve the cube, or None if no solution is found
Function solve_rubiks_dynamic(cube):
    if is_solved(cube) then
        return []
    if cube in dp then
        return dp [cube]
    shortest_solution ← None;
    for move in all_possible_moves() do
        new_cube ← apply_move(cube, move);
        solution ← solve_rubiks_dynamic(new_cube);
        if solution is not None then
            solution ← [move] + solution;
            if shortest_solution is None or len(solution) ¡ len(shortest_solution) then
                shortest_solution ← solution;
    dp [cube] ← shortest_solution;
    return shortest_solution;
Function solve_rubiks(cube):
    dp ← {};
    return solve_rubiks_dynamic(cube);
```

3. Analisis Perbandingan Algoritma

Untuk menganalisis perbandingan antara kedua algoritma, kami menguji keduanya dengan lima input *Rubik's Cube* 2x2x2 yang berbeda dan terurut secara ascending berdasarkan kompleksitasnya. Kompleksitas diukur berdasarkan jumlah gerakan yang diperlukan untuk menyelesaikan *Rubik's Cube* dari keadaan awal. Grafik dan tabel perbandingan *running time* dari kedua algoritma:



Input (scramble)	Backtracking (seconds)	Dynamic Programming (seconds)
1 gerakan	0,00020	0,00048
3 gerakan	0,29137	0,00050
5 gerakan	14,78876	0,00047
6 gerakan	282,13754	0,00051
7 gerakan	4899,74933	0,00050

Analisis Running Time:

- Backtracking:
  Waktu eksekusi untuk menyelesaikan *Rubik's Cube* dengan 1 gerakan termasuk sangat cepat, yaitu 0,00020 detik. Namun seiring dengan meningkatnya gerakan *scramble*, waktu eksekusi pun meningkat secara signifikan hingga sekitar 81 menit untuk 7 gerakan. Hal ini menunjukkan bahwa waktu yang dibutuhkan meningkat eksponensial seiring dengan peningkatan kompleksitas *scramble*.

- Dynamic Programming
  Untuk semua input (1, 3, 5, 6, dan 7 gerakan *scramble*), waktu eksekusi tetap konstan sekitar 0,00047 hingga 0,00051 detik. Hal ini menunjukkan bahwa algoritma *dynamic programming* mampu mengatasi peningkatan kompleksitas tanpa peningkatan signifikan dalam waktu eksekusi.

4. Kesimpulan

Dari analisis dan grafik perbandingan, dapat disimpulkan bahwa algoritma *dynamic programming* umumnya lebih efisien dibandingkan dengan *backtracking* dalam menyelesaikan permainan *Rubik's Cube* 2x2x2, terutama untuk input dengan kompleksitas yang lebih tinggi. Hal ini disebabkan oleh kemampuan *dynamic programming* dalam menghindari perhitungan berulang melalui penyimpanan hasil sub-masalah. *backtracking*, meskipun sederhana dan langsung, cenderung lebih lambat karena harus mencoba semua kemungkinan solusi.

Referensi

Korf, R. E. (1997, July). Finding optimal solutions to *Rubik's Cube* using pattern databases. In AAAI/IAAI (pp. 700-705).

Rokicki, T. (2010). Twenty-two moves suffice for *Rubik's Cube*®. The Mathematical Intelligencer, 32(1), 33-40.

Rokicki, T., Kociemba, H., Davidson, M., & Dethridge, J. (2014). The diameter of the *Rubik's Cube* group is twenty. siam REVIEW, 56(4), 645-670.

Felgenhauer, B., & Jarvis, F. (2005). God's number is 20. Available at https://www.cube20.org/.

Ferenc, D. (2024). Herbert Kociemba's optimal cube solver – Cube Explorer. Retrieved from https://ruwix.com/the-rubiks-cube/herbert-kociemba-optimal-cube-solver-cube-explorer/.

Kociemba, H. (2024). Cube Explorer. Retrieved from http://kociemba.org/cube.htm.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 27(1), 97–109.

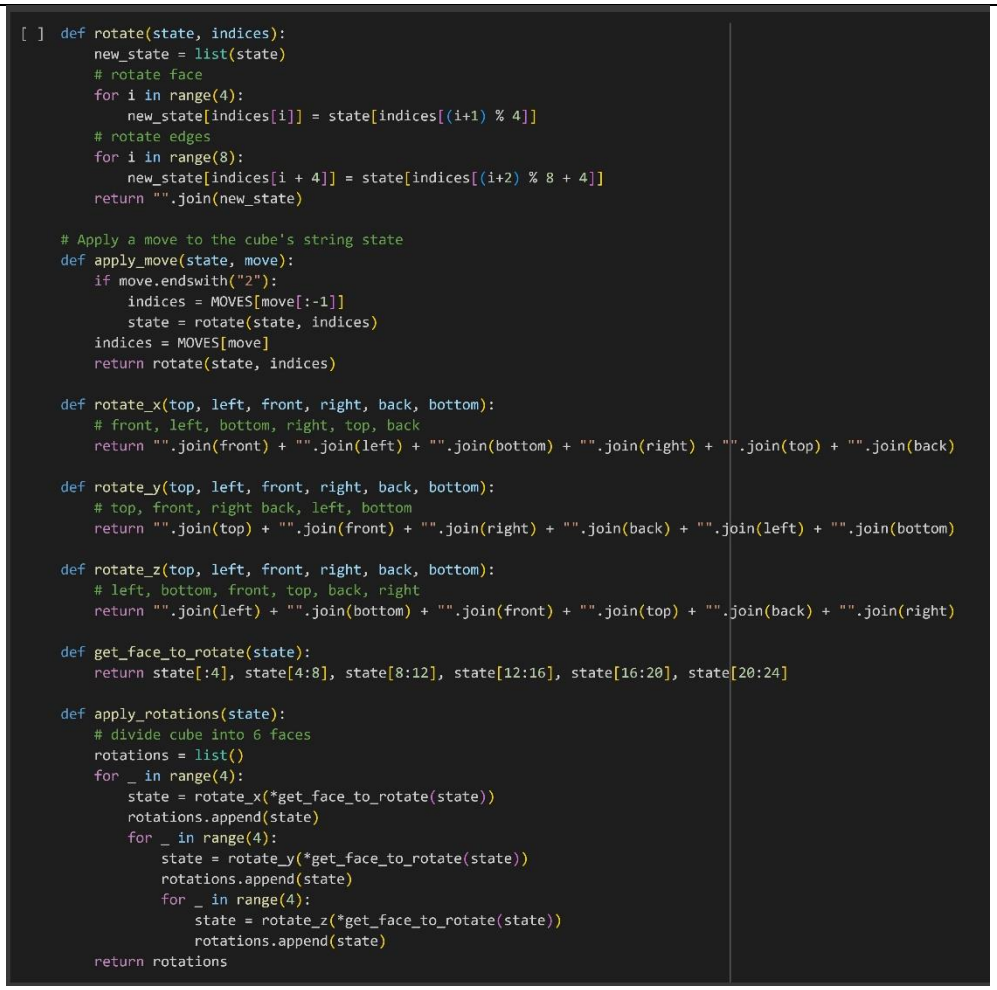
Korf, R. E. (1997). Finding optimal solutions to *Rubik's Cube* using pattern databases. In AAAI/IAAI (pp. 700–705).

Roux, G. (2024). Roux method. Retrieved from https://rouxmethod.wordpress.com/.

Rubik's Cube official site. (2024). Retrieved from https://www.rubiks.com/.

First versions of *Rubik's Cube*. (2022, November). Retrieved from https://www.firstversions.com/2015/08/rubiks-cube.html.

LAMPIRAN

	Fungsi-fungsi pada kode program

```
[ ]  def rotate(state, indices):
         new_state = list(state)
         # rotate face
         for i in range(4):
             new_state[indices[i]] = state[indices[(i+1) % 4]]
         # rotate edges
         for i in range(8):
             new_state[indices[i + 4]] = state[indices[(i+2) % 8 + 4]]
         return "".join(new_state)

     # Apply a move to the cube's string state
     def apply_move(state, move):
         if move.endswith("2"):
             indices = MOVES[move[:-1]]
             state = rotate(state, indices)
         indices = MOVES[move]
         return rotate(state, indices)

     def rotate_x(top, left, front, right, back, bottom):
         # front, left, bottom, right, top, back
         return "".join(front) + "".join(left) + "".join(bottom) + "".join(right) + "".join(top) + "".join(back)

     def rotate_y(top, left, front, right, back, bottom):
         # top, front, right back, left, bottom
         return "".join(top) + "".join(front) + "".join(right) + "".join(back) + "".join(left) + "".join(bottom)

     def rotate_z(top, left, front, right, back, bottom):
         # left, bottom, front, top, back, right
         return "".join(left) + "".join(bottom) + "".join(front) + "".join(top) + "".join(back) + "".join(right)

     def get_face_to_rotate(state):
         return state[:4], state[4:8], state[8:12], state[12:16], state[16:20], state[20:24]

     def apply_rotations(state):
         # divide cube into 6 faces
         rotations = list()
         for _ in range(4):
             state = rotate_x(*get_face_to_rotate(state))
             rotations.append(state)
             for _ in range(4):
                 state = rotate_y(*get_face_to_rotate(state))
                 rotations.append(state)
                 for _ in range(4):
                     state = rotate_z(*get_face_to_rotate(state))
                     rotations.append(state)
         return rotations
```

```
[ ]  # Check if the state is solved
     def is_solved(state):
         return state == SOLVED_STATE

     # Backtracking solver
     def solve_cube_backtrack(state, depth, max_depth, path):
         shortest_path = None
         if is_solved(state):
             return path

         if depth == max_depth:
             return None

         for move in MOVES:
             new_state = apply_move(state, move)
             result = solve_cube_backtrack(new_state, depth + 1, max_depth, path + [move])
             if result:
                 if not shortest_path or len(result) < len(shortest_path):
                     shortest_path = result

         return shortest_path
```

Kode program
backtracking

```
[ ]  def canonical_form(state):
         rotations = apply_rotations(state)
         return min(rotations)

     def precompute_states():
         dp = {}
         queue = deque([(SOLVED_STATE, [])])
         dp[canonical_form(SOLVED_STATE)] = []

         # Correct upper limit for progress tracking
         total_states = 3_674_160
         # total_states = 100
         visited = set()
         visited.add(canonical_form(SOLVED_STATE))

         with tqdm(total=total_states, desc="Precomputing states") as pbar:
             while queue and len(dp) < total_states:
                 current_state, path = queue.popleft()

                 for move in MOVES:
                     new_state = apply_move(current_state, move)
                     canonical_new = canonical_form(new_state)
                     if canonical_new not in visited:
                         dp[canonical_new] = path + [move]
                         visited.add(canonical_new)
                         queue.append((new_state, path + [move]))
                         pbar.update(1)

         return dp

     def solve_dp(state, dp):
         canonical_state = canonical_form(state)
         return dp.get(canonical_state, None)

     # Check if dp file exist
     if os.path.exists("/content/drive/MyDrive/Colab Notebooks/Tubes SA/dp.json"):
         with open("/content/drive/MyDrive/Colab Notebooks/Tubes SA/dp.json") as f:
             dp = json.load(f)
     else:
         start_dp = time.time()
         dp = precompute_states()
         end_dp = time.time()
         print("DP Time: ", (end_dp - start_dp))
         with open("/content/drive/MyDrive/Colab Notebooks/Tubes SA/dp.json", "w") as f:
             json.dump(dp, f)
```

Kode program *dynamic programming*

	Contoh implementasi kedua algoritma
	Tampilan output program

Link Kode Program: https://github.com/gidachmad/rubiks-2x2x2-solver.git