The Use of BFS to Solve Pocket Cube

Dyah Rahmawati 13511012
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13511012@std.stei.itb.ac.id

***Abstract*—Pocket Cube is a sample member of the big rubic cube's family. This cube is able to be solved by many ways. But now, the focus is BFS algorithm. This paper is about the Breadth-First Search (BFS) algorithm which is used for solving pocket cube. We will discuss about the ability of BFS to solve pocket cube using graph by enumerating the possibilitiy of movements till the solution is found.**

***Index Terms*—BFS, cube, solver.**

I. INTRODUCTION

Rubik's Cube is a 3-D combination puzzle invented in 1974 by Hungarian sculptor and professor of architecture Ernő Rubik. Originally called the "Magic Cube.

In a classic Rubik's Cube, each of the six faces is covered by nine stickers, each of one of six solid colours (traditionally white, red, blue, orange, green, and yellow, where white is opposite yellow, blue is opposite green, and orange is opposite red, and the red, white and blue are arranged in that order in a clockwise arrangement). An internal pivot mechanism enables each face to turn independently, thus mixing up the colours. For the puzzle to be solved, each face must be returned to consisting of one colour. Similar puzzles have now been produced with various numbers of sides, dimensions, and stickers.

Fig. 1. standard rubik and other rubik's family

One sample member of the large rubik's family is called pocket cube. It's size is 2 x 2 x 2. It is the two layered version of a Rubiks Cube. It was invented by Rubik Erno before the '80 and was patented on March 29, 1983. At first sight it may seem to be simple and small but even this puzzle has more than 3,6 million possible permutations.

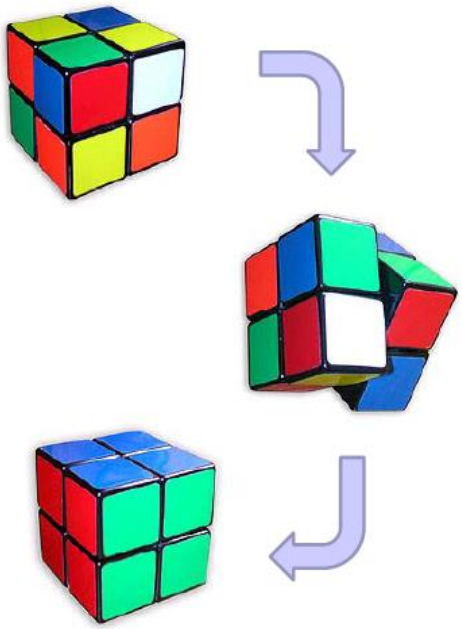


Fig. 2. Pocket Cube

Configuration Graph:

- vertex for each possible state
- edge for each basic move (e.g., 90 degree turn) from one state to another
- undirected: moves are reversible

II. SOME THEORIES

A. Graph & Graph Traversal & it's Algorithm

1. Vertex,vertices

A vertex of a graph is a connection point. A graph has a set of vertices, usually shown as V = {v1, v2, ..., vn}, V = {A, B, C} or V = {1, 2, ... N}. The number of vertices in a graph is |V|, but is sometimes written in equations as just V. A vertex may have no connections, one connection or many connections. A vertex may have any number of properties such as a name or a color.

2. Edge

An edge in a graph is a connection between vertices. Given vertices v1 and v2 in a graph, the edge between them may be written as (v1,v2) or sometimes [v1,v2]. A graph has a set of edges usually denoted E, E = {(v1,v2), (v2,v3)}. The number of edges in a graph is |E| but is sometimes written in equations as just E.

3. Graph

A graph is a set of vertices and a set of edges. G = (V, E).

In computer science, **graph traversal** is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way. Tree traversal is a special case of graph traversal.

Unlike tree traversal, graph traversal may require that some nodes be visited more than once, since it is not necessarily known before transitioning to a node that it has already been explored. As graphs become more dense, this redundancy becomes more prevalent, causing computation time to increase, as graphs become more sparse, the opposite holds true.

Thus, it is usually necessary to remember which nodes have already been explored by the algorithm, so that nodes are revisited as infrequently as possible (or in the worst case, to prevent the traversal from continuing indefinitely). This may be accomplished by associating each node of the graph with a "color" or "visitation" state during the traversal, which is then checked and updated as the algorithm visits each node. If the node has already been visited, it is ignored and the path is pursued no further; otherwise, the algorithm checks/updates the node and continues down its current path.

Several special cases of graphs imply the visitation of other nodes in their structure, and thus do not require that visitation be explicitly recorded during the traversal. An important example of this is a tree, during a traversal of which it may be assumed that all "ancestor" nodes of the current node (and others depending on the algorithm) have already been visited. Both the depth-first and breadth-first graph searches are adaptations of tree-based algorithms, distinguished primarily by the lack of a structurally determined "root" node and the addition of a data structure to record the traversal's visitation state.

Depth First Search or DFS is a method of traversing every element in a tree (or graph) by recursively visiting the first child, then second, etc, from the root node. The idea is we'll start at the root, then keep track of both children and go to the firrst child, treating it as a new root, and repeating the process until we reach a leaf. Then, we'll go back one step and go to the second child of the parent of the leaf and continue traversal. This process is recursive and continues until all nodes have been traversed.

DFS traverses down the tree, then across. Alternately, we can traverse the tree by layer using Breadth First Search (BFS). To do this, create a list of nodes to traverse and for each node in the list, add its children onto the end of the list. It should be clear why this will traverse the tree by layers.

In this case, we will learn more about BFS, and we will use it for solving pocket cube.

B. BFS

Breadth-First search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.\

BFS is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it. It does not use a heuristic algorithm.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

If the element that is sought is found in this node, quit the search and return a result. Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.

Below is the pseudocode:

```
procedure BFS(Graph,source):
      create a queue Q
      enqueue source onto Q
      mark source
      while Q is not empty:
      dequeue an item from Q into v
            for each edge e incident
            on v in Graph:
                  let w be the other
                  end of e
                  if w is not marked:
```

```
mark w
enqueue w onto
Q
```

About the space, since all of the nodes of a level must be saved until their child nodes in the next level have been generated, the space complexity is proportional to the number of nodes at the deepest level. Given a branching factor b and graph depth d the asymptotic space complexity is the number of nodes at the deepest level, O(bd). When the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O(| E | + | V |) where | E | is the cardinality of the set of edges (the number of edges), and | V | is the cardinality of the set of vertices. In the worst case the graph has a depth of 1 and all vertices must be stored. Since it is exponential in the depth of the graph, breadth-first search is often impractical for large problems on systems with bounded space.

And then about the time complexity, since in the worst case breadth-first search has to consider all paths to all possible nodes the time complexity of breadth-first search is $1+b+b^2+b^3+\ldots+b^d$ which is $O(b^d)$. The time complexity can also be expressed as O(| E | + | V |) since every vertex and every edge will be explored in the worst case.

Breadth-first search is complete. This means that if there is a solution, breadth-first search will find it regardless of the kind of graph. However, if the graph is infinite and there is no solution breadth-first search will diverge.

If the shallowest goal node is at some finite depth say d, breadth-first search will eventually find it after expanding all shallower nodes (provided that the branching factor b is finite).

About tht optimality, for unit-step cost, breadth-first search is optimal. In general breadth-first search is not optimal since it always returns the result with the fewest edges between the start node and the goal node. If the graph is a weighted graph, and therefore has costs associated with each step, a goal next to the start does not have to be the cheapest goal available. This problem is solved by improving breadth-first search to uniform-cost search which considers the path costs. Nevertheless, if the graph is not weighted, and therefore all step costs are equal, breadth-first search will find the nearest and the best solution.

It has been empirically observed (and analytically shown for random graphs) that incomplete breadth-first search is biased towards nodes of high degree. This makes a breadth-first search sample of a graph very difficult to interpret.

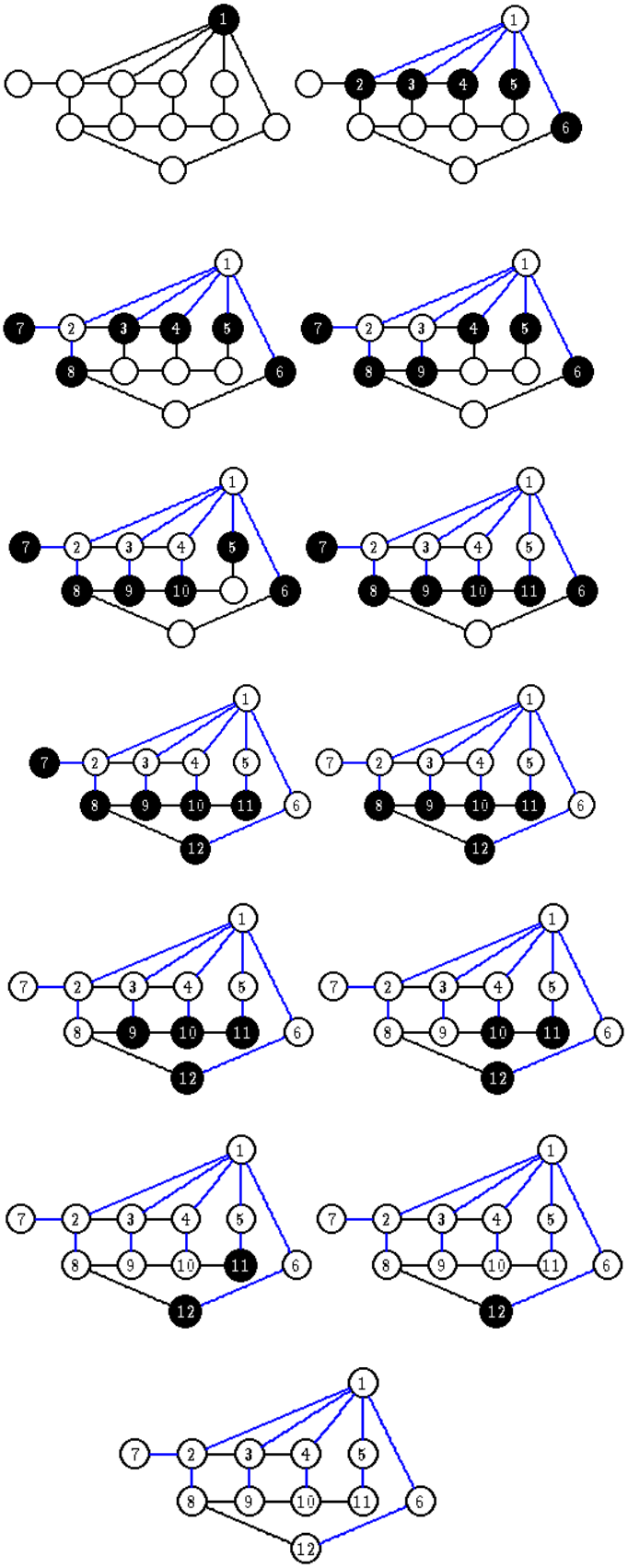


Fig. 3. Sample Ilustration for BFS

C. Permutation

Informally, a permutation describes an ordering of a set's elements. Formally, an N-element permutation is an

one-to-one mapping of the set 1, 2 … N to itself. A permutation f can be represented by the list of N numbers f(1), f(2)…f(N). Here is an example of a 5-element permutation:

$$\Pi = (3, 4, 1, 5, 2)$$

As a suggestion, that in order to apply Π to a 5-element list, we should first output the third element, then the fourth element, then the first element, and so on. Given the list [a, b, c, d, e], we can apply Π to it (permute its elements by Π) and obtain [c, d, a, e, b]. Figure 4 illustrates the process of applying a permutation.

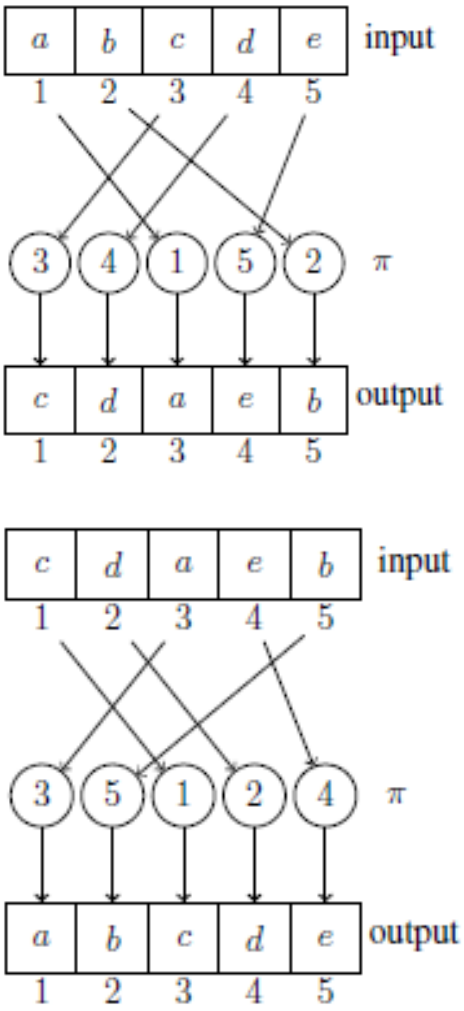


Fig. 4. The result of applying Π to [a, b, c, d, e]. Applying Π^{-1} to this result produces the original list.

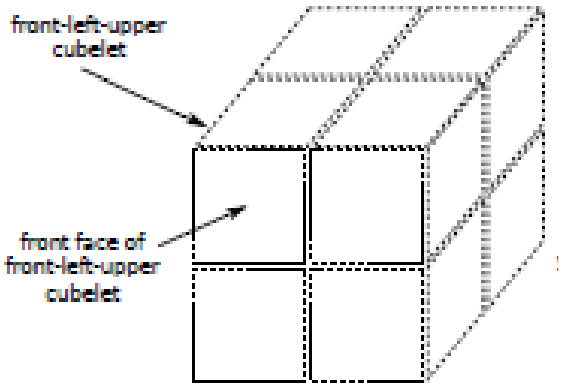
The inverse of a permutation "undoes" the effects of applying the permutation to a list of elements. For example, by applying Π^{-1} (the inverse of Π) to [c, d, a, e, b], we should obtain the original list [a, b, c, d, e]. Remember that we obtained [c, d, a, e, b] by applying Π^{-1} to [a, b, c, d, e]. A permutation's inverse can be computed by observing that $\Pi^{-1}(\Pi(i)) = i$ for $1 \leq i \leq N$. Intuitively, if Π moves the third element in the input to the first position in the output, Π^{-1} must take the first element in that output (which becomes its input) and move it to the third position in its own output, because Π^{-1}'s output must match Π's input

III. Analysis

We will try to solve Rubik's Cube problem by enumerat all possible configurations of the Rubik's cube as vertices in a graph, and uses edges to represent valid twists of the cube. Given the node of the starting state, we will use Breadth-First Seach (BFS) to find a sequence of edges (cube twists) that will "solve" the cube, by getting into a desirable configuration. Both configurations and moves are represented using permutations.

A. Cubic State

A plastic 2x2 Rubik's cube is made out of 8 plastic "cubelets". Each plastic cubelet has 3 visible faces that are colored, and 3 faces that are always face the center of the big cube, so we never see them, and we ignore them from now on. Therefore, a plastic 2x2 Rubik's cube has 24 (8x3) colored plastic faces belonging to the 8 cubelets. The code represents plastic faces using constants named as follows: yob is the yellow face of the cubelet whose visible faces are yellow, orange, and blue. The code also numbers the 24 plastic faces from 0 to 23, and these numbers are the values of the constants named according to the convention above. One way of representing the Rubik's cube configurations is to reflect the process of building a physical cube by pasting the 24 colored plastic faces on a wireframe of a cube. The left side of Figure 6 shows a wireframe 2x2 Rubik's cube. The cube's wireframe has 8 cubeletes wireframes, each of which has 3 visible hollow faces where we can paste a plastic face. We refer to the wireframe faces as follows: flu is the front face of the front-left-upper cubelet in the wireframe. Wireframe faces are also associated numbers from 0 to 23. A cube configuration describes how plastic faces are pasted onto the wireframe cube, so it maps the 24 plastic faces 0-23 to the 24 wireframe faces 0-23. This means that a configuration is a permutation, which can be stored in a 24-element array.



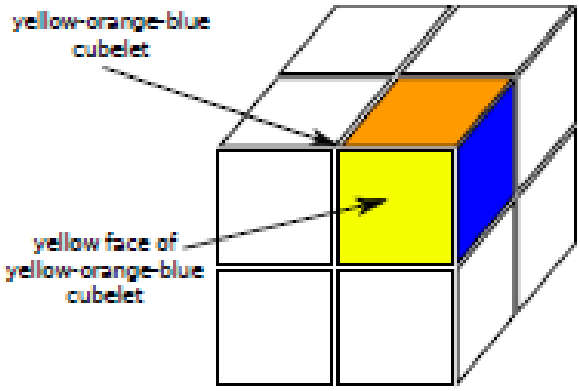


Fig. 5. The wireframe Rubik's 2x2 cube is the first image, and the plastic cube is at it's below. The plastic cube is made out of plastic faces, and the wireframe cube has positions where the plastic faces can be pasted.

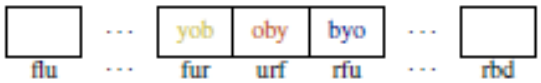


Fig. 6. A configuration of the Rubik's cube is represented as a 24-element array, mapping the 24 plastic faces to the 24 wireframe faces. The configuration above has the yob plastic cubelet mapped to the fru wireframe cubelet, like in Figure 5.

B. Graph Representation

Given the representation above, there are 24! Possible configurations. Some configurations are outright impossible. For example, mapping two faces of the same plastic cubelet to faces of different wireframe cubelets will clearly result in an impossible configurations, because we're not allowed to break apart the cube in order to solve it. A graph with 24! configurations won't fit into a normal machine's RAM, so we can't use the straight-forward approach of generating the graph first, and then running Breadth-First Search (BFS) on it. Instead, we will code up an implicit representation of the graph, which will allow us to generate the vertices and edges that the BFS visits, on-the-fly, as we run BFS. In order to run BFS, we need the vertices corresponding to our stating state and to the winning state, and an implementation of neighbors(V), which returns all the neighbors of a given vertex v.

C. Cube Twists

In order to implement neighbors(V), we need to analyze the configuration changes caused by cube twists, and code them up inside the neighbors(V) method. Twisting a cube changes the cube's configuration, by changing the position of the plastic faces onto the wireframe. A twist moves the cube's wireframe so, for example, rotating the front face clockwise will always move the plastic face that was pasted onto flu (the front face of the frontleft-upper cubelet) to rfu (the right face of the front-right-upper cubelet). Figure below illustrates the effects of rotating the front face clockwise.

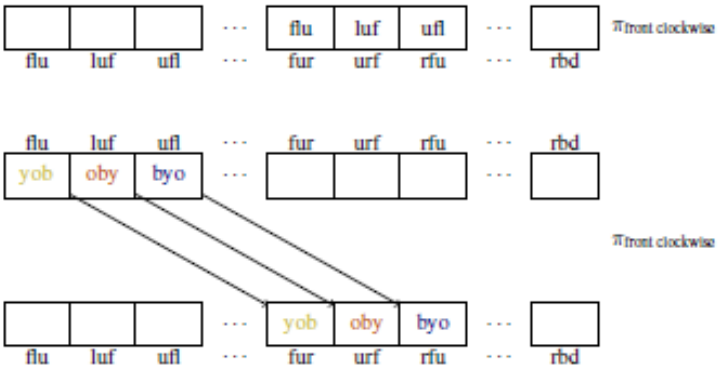


Fig. 7. The configuration changes and permutation representing a clockwise twist of the cube's front face.

Due to the property above, we can represent cube twists as permutations. Applying a twist's permutation to the list describing a configuration permutation produces the list describing the new configuration permutation. So we can implement neighbors(V) by hard-coding the permutations corresponding to cube twists, and applying them to the configuration corresponding to v. The inverse of a twist's permutation represents the move that would undo the twist.

V. Conclusion

BFS algorithm is widely used to solve many kind of problems and one of them is to solve pocket cube. The BFS solution for solving pocket cube gives optimum solution. But it is difficult to use it in daily life, it is of course easier to use formula that's been given. But, by using the formula, we won't know where the formula comes from, while by using BFS, we could understand the process. In this case, BFS gives a "not so small"solution because all posible states should be saved and can cause out of memory.

References

[1] http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Breadth-first_search.html
accessed on 19-12-2013 at 9:18 pm
[2] http://www.cse.ohiostate.edu/~gurari/course/cis680/cis680Ch14.html#QQ1-46-92
accessed on 19-12-2013 at 9:25 pm
[3] http://courses.csail.mit.edu/6.006/spring11/lectures/lec11.pdf
accessed on 20-12-2013 at 10:44 am
[4] http://s395.photobucket.com/user/tkoerting/media/rubikscubes.jpg.html
accessed on 20-12-2013 at 10:48 am
[5] http://ruwix.com/twisty-puzzles/2x2x2-rubiks-cube-pocket/
accessed on 20-12-2013 at 11:11 am

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Desember 2013

Dyah Rahmawati
13511012