Problem Solving Approaches for Load Balancing Problem

Mochammad Dikra Prasetya 13511030
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
dikraprasetya@yahoo.co.id; 13511030@std.stei.itb.ac.id

***Abstract*—Load balancing has been a problem in computer networking, which also happened on our daily life since long, for example: loading oil drums in N-trucks, packaging house inventories for relocation with N-boxes, etc. Most people already aware of it but remained with conventional solution, which is brute force (or complete search in algorithm term). As the problem exist for centuries, computer science, however, has been growing rapidly and expands its reach of problem solving. This paper will try to confront the problem with various applicable problem solving approaches and concluded with which approach is better based on analysis. Approaches which this paper going to discuss are greedy, dynamic programming, and heuristic searching.**

***Index Terms*—Load Balancing, Greedy, Dynamic Programming, Heuristic, IDA*.**

I. Introduction

Load balancing is a computer networking method for distributing workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units or disk drives. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any one of the resources. Load balancing for a parallel system is one of the most important problems which has to be solved in order to enable the efficient use of parallel computer systems. The whole work should be completed as fast as possible. As the resources are expensive, the work should be distributed fairly and kept the workers busy.

The simplified version of the problem could be found on our daily life, balancing loads for oil trucks for example. Drums might have different sizes or different weights, loading items should be arranged properly. Whether an item should be loaded in truck A or truck B became the main problem in balancing loads. A similar problem could be found in house relocations. Packaging house inventories depends on the number of boxes available and the size of items. Fitting item needs the right distribution.

The term balanced that will be covered in the paper will be measured in such function:

$$A = \left(\sum_{i=1}^{n} Xi\right)/n$$

$$f(n) = \sum_{i=1}^{n} |Xi - A|$$

Figure 1 – Illustration loading drums to truck

Let us define above function as *imbalance function*, the sum of the absolute differences between total mass of each truck and the average of all truck mass, whereas the more balanced distribution, the less value we get from the function. The value *n* describes the number of trucks, X_i represents total mass on each truck, and *A* is the average mass of all trucks.

As a generalization of the problem, the statement will be specified to a case whereas *m* items are need to be loaded and *n* containers are available. The objective is to seek the minimum value of imbalance function.

II. Theory

A. Greedy Algorithm

Greedy algorithms tend to find locally optimal choice. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time. Greedy algorithm consists of properties: candidate set; selection function; feasibility function; objective function; and solution function. A candidate set contains solution candidates and will be eliminated by a selection function to choose the best candidate to be added to the solution. Feasibility function used to determine if a candidate can be used to contribute to a solution. An objective function determines the value that will assigned to the solution. A solution function contains the accepted solution discovered. If a greedy algorithm is applicable for finding global solution, then on most cases it might be the best

solution for the problem.

An example of greedy would be taking a local solution on the Traveling Salesman Problem, which illustrated below:

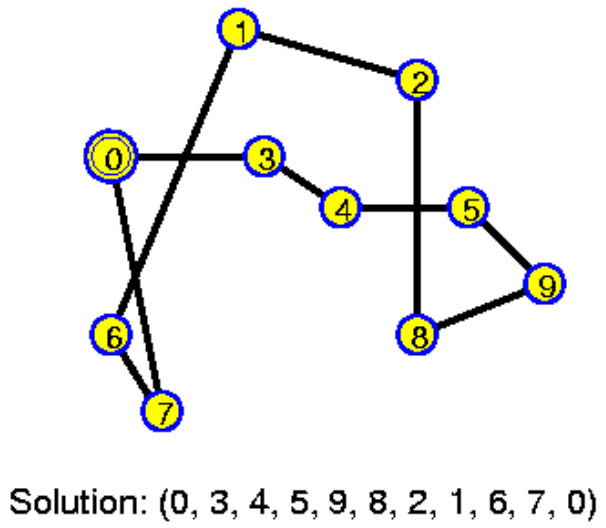

Figure 2 - TSP greedy algorithm illustration

B. Dynamic Programming

Dynamic programming is typically applied to optimization problems. Like divide-and-conquer, this approach solve complex problems by breaking them down into simpler subproblems. The illustration of subproblems can we see from the problem Fibonacci:

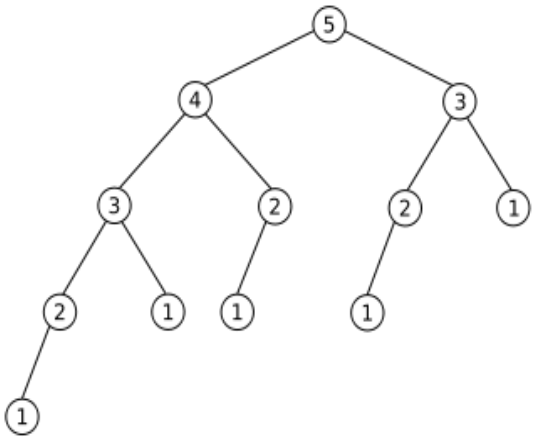

Figure 3- Fibonacci subproblem breakdown

It is applicable to problems exhibiting the properties of overlapping subproblems. The method takes far less time than brute force methods. This approach could be defined as a sequence of steps:

- Characterize the structure of an optimal solution;
- Recursively define the value of an optimal solution;
- Compute the value of an optimal solution in a bottom-up fashion;
- Construct an optimal solution from computed information.

The concept of dynamic programming is exchanging memory for faster time response. If applicable, time complexity should be proportional to the size of memorization table.

As an example, here it is a Fibonacci problem with naïve approach compared with dynamic programming better solution.

Naïve approach:

```
function fib(n)
        if n = 0 return 0
        if n = 1 return 1
        return fib(n - 1) + fib(n - 2)
```

Dynamic programming approach:

```
var m := map(0 → 0, 1 → 1)
    function fib(n)
        if key n is not in map m
            m[n] := fib(n - 1) + fib(n
- 2)
        return m[n]
```

C. Heuristic Searching

While the exhaustive search is impractical, heuristic methods are used to speed up the process of finding a satisfactory solution via mental shortcuts to ease the cognitive load of making a decision. A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). As A* traverses the graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way.

However A* could be optimized combined with iterative deepening depth-first-search to keep the process use less memory, which these days is known as IDA* (iterative deepening A*). While the standard iterative deepening depth-first search uses search depth as the cutoff for each iteration the IDA* uses the more informative f(n) = g(n) + h(n) where g(n) is the cost to travel from the root to node n and h(n) is the heuristic estimate of the cost to travel from n to the solution. IDA* follows this pattern of pseudocode:

```
node                    current node
```

```
 g                  the cost to reach current node
 f                  estimated cost of the cheapest
path (root..node..goal)
 h(node)            estimated cost of the cheapest
path (node..goal)
 cost(node, succ)   path cost function
 is_goal(node)      goal test
 successors(node)   node expanding function


 procedure ida_star(root, cost(), is_goal(), h())
   bound := h(root)
   loop
     t := search(root, 0, bound)
     if t = FOUND then return FOUND
     if t = ∞ then return NOT_FOUND
     bound := t
   end loop
 end procedure


 function search(node, g, bound)
   f := g + h(node)
   if f > bound then return f
   if is_goal(node) then return FOUND
   min := ∞
   for succ in successors(node) do
     t := search(succ, g + cost(node, succ), bound)
     if t = FOUND then return FOUND
     if t < min then min := t
   end for
   return min
 end function
```

IV. Implementation & Analysis

A. Naïve Approach

Straight-forward solution is always an option for configurationally problems. As it is straight-forward, we will have a function that searching (up to) the entire search space in bid to obtain the required solution. Every steps should determine where should we put the *i*-th item into. Each item must be put only once on one of the containers. Then after the whole item have been put, we should count the value of imbalance function and keep the minimum value. Below is the pseudocode of the solution:

```
function completeSearch(n)
       if all item have been put
           ans = min(ans, imbalanaceFunct())
       else
           for all containern
              try put at Container-i
              completeSearch(n+1)
```

For most cases complete search would give us the optimal solution, but takes too much time. The complexity is at least O(n!) whereas the method will look in all of the possible configurations.

B. Greedy Algorithm

The simplest greedy that one may thought would be Greedy by Weight, whereas after sorting all of the weight, each of them will be distributed consecutively with the number of item on the container at most the $m/n + 1$ (for balancing the distribution). Below is the scratch pseudocode:

```
function greedyByWeight(n)
       sort all item in ascending order
       for each item put to container by order
```

But after that the problem arise is that approach would only give us local solution, which is not always the best one. To search the global solution, there are some observations:

Observation 1: If there exists an empty container, at least one container with more than one item must be moved to this empty container. Otherwise the empty container would give us greater value of imbalance function.

Observation 2: If $m > n$, then $m-n$ items must be paired with one other item already in some containers, which is known as Pigeonhole principle.

Observation 3: The greatest value of weight should be put alone. Pairing it with other item would only make the weight gap bigger.

By these observations, the optimal solution would be found in a condition where the item could be put as a pair ($m <= 2n$). The pairing will come in greedy, we match the biggest weight item with the lightest weight item available. If $m < 2n$, we should put dummy items with zero weight as the greatest item should be put alone.

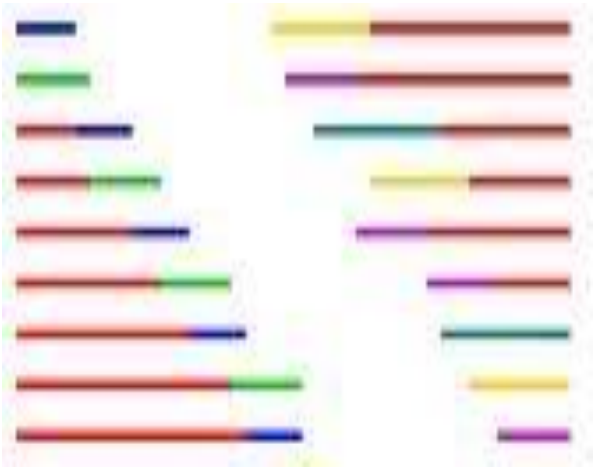

Figure 4- Optimal weight pairing illustration

Above illustration showed us that by pairing the greatest and lowest weight item we could get the most balanced configuration on the case. Below is the pseudocode of the optimal solution:

```
function optimalGreedy(n)
     put dummy items so we have 2n item
     sort all item in ascending order
     for i = 1 to n
         put i-th and n-i+1-th item to
           container i
```

Therefore, our Greedy by Weight approach with the greedy pairing strategy will always give us the optimal solution. This solution is also have a satisfying time complexity of O(*n log n*) which comes from the sorting time. If we omit the sorting time, we would get an O(*n*) time complexity. However, this solution are limited to the fulfillment of condition *m* <= *2n*, where the item could be put as a pair at most. As the problem grow, a more sophisticated observations should be conducted and maybe greedy approach would not be an applicable solution.

C. Dynamic Programming

Dynamic programming approach would be applicable if the problem can be broke-down to subproblems and eventually the base of recursive. There are some observations that provides us the structure of dynamic programming recursive:

Observation 1: A configuration where there is no items would make the imbalance function return 0.

Observation 2: When putting *i*-th item, its best imbalance function score would depends on *i-1*-th item configuration.

From above observations, we could translate them into the recursive formula of dynamic programming solution. Meanwhile if we look closely we could see that the problem is quite similar with Traveling Salesman Problem as the dynamic programming would take a set type parameter that contains last configurations. The formula would represented as the code below:

```
Map bestSol <= map of configuration set,
                The dp table  n x 2^n
function dpSolution(n; last_conf)
   if n = 0  return 0
   else
   for each container try to put item-n
        try to put
        bestSol = currentImbalanaceFunc()
              + dpSolution(n-1; lastlastconf)
   return bestSol;
```

The time complexity of the solution above would be O(*n* 2^n), as it corresponds the size of memorization table. It is quite fast dynamic programming approach for such problem, however, the table size would limit the range of constraint covered by the solution. Better solution is not at reach of dynamic programming whereas this problem is similar with Traveling Salesman Problem while that problem still stuck in the same state of dynamic programming.

D. Heuristic Searching

IDA* should be applicable in such problem. Heuristic searching needs heuristic function where it counts the closest forecast to reach the best solution. Pruning is the most important property of heuristic searching. Below is the observations that will support our heuristics:

Observation 1: Heuristic searching does better at limited space, so we should limit it at some value. We could apply binary search to find the answer, and use the pivot value as the limit. Therefore, heuristic searching will be taking the role of validation function of binary search.

Observation 2: If we set the limit value of answer, we could forecast the maximum number of space that we could waste on. Therefore we could reduce the searching space that we should visit on the search.

Observation 3: If we try to put *i*-th item which weights the same as *i-1*-th, we start to put it on the last container that *i*-1-th try to put.

With those observations, we could meet a great time complexity thanks to all of the pruning on above observations. The whole algorithm will be described in the pseudocode below:

```
Limitval= the limit from pivot
function heuristicCheck(n)
   if all items is put
       return true
   else
   for each container try to put item-n
        try to put
        check if waste is applicable
          for next try
        if heuristicCheck(n+1)
           return true

     return false


function binarySearch(n)
    le = 1
    ri = n
    while le < ri
      piv = (le+ri)/2
      set piv as limitval
     if heuristicSearch(1)
        ri = piv
     else
        le = piv+1

     return ri  //the minimum
               //imbalance function
```

The heuristic search runs very fast for the most cases. Pruning made the search space reduced vastly and could handle bigger case of *n* and *m*.

V. Conclusion

From all of the implementation above, the most applicable and best at time would be heuristic searching using binary search and IDA* validation check. But for some cases, greedy works far better on the case *m* <= *2n* as the optimal solution do exist at that case.

References

[1] Steven Halim. Competitive Programming 3rd Edition, 2013
[2] Rinaldi Munir, Diktat Strategi Algoritma, 2009.
[3] Thomas H Cormen, Introduction to Algorithm, 2003.
[4] http://mathfaculty.fullerton.edu/mathews//n2003/BisectionMod.html
[5] http://www.apl.jhu.edu/~hall/AI-Programming/IDA-Star.html
[6] www.cs.berkeley.edu/~vazirani/algorithms/chap5.pdf

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, 20 Desember 2013

Mochammad Dikra Prasetya 13511030