

A method for optimizing communication paths in distributed systems

Martin Dam, Jens Myrup Pedersen

Aalborg University, Department of Electronic Systems, DK-9220, Aalborg East, Denmark

{martind, jens}@es.aau.dk

Abstract—One of the challenges in designing large distributed systems is to optimize the communication and dependencies between modules, where a trade-off between performance and resource usage often must be made. This paper suggest a simple taxonomy for comparing potential designs, and provides also an algorithm for automatically suggesting good candidates. The algorithm is compared to previous work, where it is shown to be significantly faster, while providing comparable results. The results are also shown to be near-optimal in comparison with a brute-force algorithm which is only tested for small problem instances.

Keywords—Networks, internet, routing, load balanced routing, network testing, topology, traffic control (communication), feedback communication, probabilistic routing.

I. INTRODUCTION

One of the challenges when designing a large software/hardware system is to optimize communication and dependencies between modules. This is usually done manually by system designers who are supposed have a good overview of the system, and a knowledge of what communication is needed between the different parts of the system. However, for large distributed systems this task is not trivial: Given a set of modules or components and a corresponding set of communication/interface demands, there can be many possible solutions. Especially for communication, there is an important trade-off between performance, such as delay requirements, and resource usage (e.g. cost and power consumption), which make the problem quite complex to solve manually. This motivates the work on more automated methods. It is important to note though, that the suggested approaches are meant as support for the manual work: Many aspects which are not easily put on formula, play important roles in system design.

In this paper we first shortly review existing work on automated methods for optimizing such systems, and come up with two proposals. First, we propose a taxonomy, based on what we call k-measure, which can be used to compare different designs using just a single value. This measure reflects a trade-off between the number of links in the system (modelled as a graph) and the length of the communication paths. Thus, it allows for comparing any proposed design with what can be obtained optimally (if it can be calculated) or by heuristic algorithms.

In the next part we propose a heuristic algorithm for finding a graph with close to optimal k-value. Performance

of the algorithm - both in terms of resulting graphs and time complexity - is evaluated and compared to previous work in the field.

An additional aspect is also dealt with, namely graph centralization. In some application fields, such as p2p and wireless sensor networks, it can be advantageous to have a few (or just a single) nodes handling many connections: This can be used for spectrum optimization, and these central nodes can be more permanent nodes, equipped with permanent power supply, more processing power etc. The heuristic algorithm we proposed is design to favor such centralization.

The rest of the paper is organized as follows. In section II we explain the terminology used in the paper, and give a brief introduction to Moyle's and Thompson's previous work on finding Minimum Equivalent Graphs (MEGs) from a digraph [1], and to later work by Carothers and Cragon [2], who were the first to study the MEG problem with additional constraints on delay and bandwidth. In section III we propose the taxonomy mentioned above, and we also introduce the heuristic algorithm for finding good communication paths, given a set of communication requirements. In section IV and V this method is compared to the method proposed by Carothers and Cragon, and to a brute-force implementation which can find optimal solutions in smaller graphs. The conclusions and suggestions for further work are given in section VI.

II. BACKGROUND

We will simply use the standard terminology for directed graphs, such as in e.g. [3]. A directed graph G consists of a set of nodes $V = \{v_1, v_2, \dots, v_n\}$ and a set of directed edges $E = \{e_1, e_2, \dots, e_k\}$. An edge describes a relation from one vertex to another, for example when information has to be passed from one module v_i to another module v_j . For simplicity such an edge is simply denoted e_{ij} . A path P_{ij} is an ordered set of edges and vertices following each other, such as $v_u, e_{uv}, v_v, e_{vw}, v_w$. The length of a path, $|P_{ij}|$, corresponds to the number of edges it contains, so this path is of length 2. There can be more than one path between a pair of nodes v_i, v_j . In that case, we will simply let P_{ij} denote an arbitrary shortest path from v_i to v_j . The length of such a shortest path, which is also the distance from v_i to

v_j , is written d_{ij} . Moreover, let $M = |E|$ denote the number of edges and $N = |V|$ denote the number of nodes.

Based on this general notation, we are now able to make a more precise formulation of the problem we intend to solve.

The starting point of our problem is an initial directed graph, containing the modules of a system, together with a specification of communication requirements. The components (or modules) are described as nodes, and the connection between components as directed edges. Thus, we have the graph $G = \langle V, E \rangle$, where V and E are the nodes and edges as described above. In this initial graph there is a one-to-one correspondance between the edges and the communication requirements.

What we want to obtain is a new graph $G^0 = \langle V, E^0 \rangle$, where E^0 is a subset of E . For every edge e_{ij} in the original graph G , G^0 must contain a path from v_i to v_j . We will try to benefit from the fact that communication can take place using intermediate nodes, e.g. having paths longer than one, and that this can be used to reduce the number of edges in the graph.

A crucial element in the comparison, and a part of the k -measure introduced shortly, is the sum of distances from all ordered pairs of nodes with a communication requirement. For G , define this as d_* :

$$d_* = \sum_{e_{ij} \in E} d_{ij}$$

For the new graph G^0 we correspondingly define:

$$d_*^0 = \sum_{e_{ij} \in E} d_{ij}^0$$

where d_{ij}^0 is the distance from v_i to v_j in G^0 .

Note that G has a path of length 1 for all communication requirements in the system, and therefore $d_* = |E|$. In G^0 the distances can be longer, so $d_*^0 \geq d_*$.

The above problem was previously formulated as the Minimum Equivalent Graph (MEG) problem, where the aim is to find the graph with the smallest number of edges, with the same reachability as the original graph. This problem was initially solved by Moyle's and Thompson [1], followed by a number of papers with adjustments and corrections to their algorithms [4] and [5].

However, the MEG problem is not always a smart way of solving our original problem, since it does not take into account performance requirements - the paths in the final graph can be very long, leading to communication delays, excessive bandwidth consumption, and use of signification ressources for relaying. To address these problems Carothers and Cragon [2] proposed an algorithm for finding a smallest possible graph, given a bandwidth and path length requirement for each connection. The algorithm gives an approximation to the optimal result, and performs efficiently with a time complexity of $O(en^3)$. In section V we will compare

our heuristic algorithm to the H_RASP2 as proposed by Carothers and Cragon.

III. METHODS

In this section we will define our taxonomy k -measure and then describe our heuristic algorithm. In the optimization process of systems connections, there are many possible parameters to account for (e.g. delay, total number of connections, bandwidth etc.). To simplify the optimization, we propose to join relevant parameters in one equation in what we call k -measure. With k -measure, it is possible to compare two or more solutions directly with each other, based on a single value. In this paper we will define k -measure to account for the number of connections and the total path length (total delay for all connections). It is defined as:

$$k^0 = C \cdot M^0 + |d_*^0| \quad (1)$$

Where C is the maximum path length minus 0.9.

In this paper we assume that the maximum acceptable path length is 3. We therefore set the constant C to 2.1 in equation 1. This choice implies that if there is a path of length greater than 3, it is always better to use a direct connection. *Proof:* if from a node there is a path of length 4 to a destination, this length will contribute with 4 points to the k -value. If an edge from the node to its destination is added, the extra edge adds 2.1 points to the k -measure, but since the distance is reduced from 4 to 1, this saves 3 points. Therefore the k -measure of the revised graph will be lower, and consequently the solution will be better.

A. Description of Dam's Graph Optimization (DGO)

Let the input be an $N \times N$ adjacency matrix

$$M = [m_{ij}]$$

where

N = number of nodes

$m_{ij} = 1$ if there is a directed edge in G from v_i to v_j

$m_{ij} = 0$ if no edge exists

- (1) Sort the nodes in list S descending after total edges to and from the node (indegree+outdegree).
- (2) Generate an adjacency matrix F from ordering the nodes according to S , so the first item in S has index 0 in F , and the last item in S has index $N-1$ in F .
- (3) Generate list L from F by the following algorithm.
 - (3.1) for $n = 0$ to $N-1$ (ascending)
 - (3.2) $i = 0$
 - (3.3) for $j = n$ to 0 (descending)
 - (3.4) if $f_{ij} = 1$ then add path P_{ij} to L
 - (3.5) $i = i + 1$
 - (3.6) for $n = 1$ to $N-1$ (ascending)
 - (3.7) for $i = n$ to $N-1$ (ascending)

- (3.8) for $j = N$ to n (descending)
- (3.9) if $f_{ij} = 1$ then add path P_{ij} to L
- (3.10) $i = i + 1$
- (4) Generate an $N \times N$ matrix A with all $[a_{ij}] = 0$
- (5) Now generate the output matrix A by the following algorithm by using list L
 - (5.1) if L is empty goto 6)
 - (5.2) $b = \text{takeTopItem}(L)$
 - (5.3) If a path from b_i to b_j exists in A with $\text{pathlength} \leq 3$, goto 5.1)
 - (5.4) Insert edge e_{ij} into A and goto 5.1)
- (6) End algorithm

B. Time complexity for DGO

In order to determine the time complexity for DGO, we need to analyse each step in the algorithm.

- (1) Firstly all edges are counted for each node, and then sorted. Using Quicksort[3], this gives a complexity of $O(N^3)$.
- (2) This step is running through the list S , containing all the N nodes. Thus, the complexity is $O(N)$
- (3) This step selects each edge in F , and therefore has complexity of $O(|E|)$
- (4) For generating the empty matrix, a complexity of $O(N^2)$ is found when setting all entries to zero.
- (5) This step is the main algorithm. It runs through each edge in L , resulting in $|E|$ iterations. We are using a modified Breath-first search to determine if a path exists. This is modified in the way, that when it reaches a depth of C , or in our case 3, the algorithm stops, and no path is found. Also no node already visited, can be visited again in the search. The worst-case complexity of this is $O(|V|+|E|)$ [3]. This gives a complexity of $O(|E| \cdot (|V|+|E|)) \rightarrow O(|E|^2)$. In worst case, every edge except loopback exists, so a worst case given by nodes becomes $O(N^4)$.

Step 5 in the algorithm is the most complex in time, and the complexity of DGO is therefore $O(N^4)$.

H_RASP2 complexity is given by $O(|E|N^3)$ [2], which with the maximum number of edges in respect to N , comes to $O(N^5)$, which shows that DGO has an asymptotically better time complexity than H_RASP2. The practical running times are tested in the following section.

IV. EXPERIMENTAL TESTS

A. Description of the experiments

The goal of this section is to design, perform and analyse a set of tests, for proving four hypothesis:

- 1) DGO matches H_RASP2 in optimization
- 2) DGO is better at centralizing compared to H_RASP2
- 3) DGO is faster than H_RASP2 in execution time

- 4) DGO approximates a minimal solution within the given constraints

These four hypothesis is tested as follows.

- 1) A set of random graphs of different size and fill (to be defined) are generated and the graphs are optimized using both DGO and H_RASP2. The results in terms of k -measure are then compared.
- 2) In addition to k -measure, centralization is measured for the same graphs. This is done by comparing node degree distributions in the resulting graph.
- 3) Since execution time depends on hardware and software, each algorithm is implemented in C++ according to the same design rules, and they run on the same hardware. They run on a large amount of random graphs of different size and fill.
- 4) This is done by comparing the k -measures obtained in (1) to the optimal solutions. The optimal solutions are found using a brute-force method, which is feasible only with smaller graphs - in particular, this is done for $N = 6, 7, 8$. We test if DGO approximates a minimal solution, as H_RASP2 does, shown by Carothers and Cragon [2].

In the next part, we will discuss how test data is generated.

B. Test data generation

To ensure that the results are valid, the experimental tests are performed on a number of random graphs of different size and fill. Fill was previously defined [2]:

$$fill = \frac{M - N}{N^2 - N}, \quad 0 \leq fill \leq 1 \quad (2)$$

For testing a wide range of graphs, graphs of fill between 0.1 and 0.74 is used.

In order to test hypothesis 1, 2 and 3 the sizes of the graphs tested were selected to be $N = 10, 20, 30, 40, 50$. Due to the complexity of the brute-force algorithm hypothesis 4 was only test for $N=6,7,8$. In order to make a correct comparison between H_RASP2 and DGO, we define the maximum delay on the H_RASP2 algorithm to be 3. This corresponds to a maximum path length of 3, which is also the case for DGO.

A pseudorandom number generator was used in a random graph generator. The graphs are generated through the following steps:

- 1) N vertices are generated
- 2) An edge is randomly inserted into the graph
- 3) If the fill of the graph is equal to the wanted: Goto step 4).
Else: Goto step 2).
- 4) Check if the graph is connected. If so stop, else goto step 1).

C. Test execution

In this section, we will discuss how our tests are implemented and executed.

Hypothesis 1) and 2) are tested together, and use the same test set. The implementation is done in MATLAB. It runs for $N = (10, 20, 30, 40, 50)$ and for fill from 0.1 to 0.74 with 0.02 spacing. For each N and fill, 50 random graphs are generated. H_RASP2 and DGO run on each graph, and the average k -measure and centralization is compared. Only $N = 10, 30$ and 50 is included in this paper due to page limits.

Hypothesis 3) is tested by implementing H_RASP2 and DGO in C++ after the same design rules. The random graph generator implemented in MATLAB for hypothesis 1) and 2) is used to generate 10 random graphs of the same N and fill as in 1) and 2). For improving the time measurement accuracy the algorithms runs 1000 times per graph. The execution time is measured via UNIX time() [6].

Hypothesis 4) is tested using a brute force method. The time complexity of this method is high: With $N = 10$, a total of $2^{(N*(N-1))} \approx 10^{27}$ combinations are possible. Calculating all these graphs, would be a enormous task, and some extra constrains are applied:

- 1) Only the edges in the original graphs are valid. This gives a maximum combinations of 10^9 with fill = 0.5 and $N = 8$.
- 2) A decision tree is implemented, so if a graph is found to be invalid, every subgraphs where more edges are removed, is marked as invalid.
- 3) If a node has only one outgoing or one ingoing edge, this edge is locked.

With these constrains, it is possible to perform the calculations for graphs up to $N = 8$ and fill = 0.5 within a acceptable time. The test set is generated as in hypothesis 3). The graph with the best k -measure is found for each original graph, and compared to the optimized graph generated by H_RASP2 and DGO.

V. TEST RESULT

In this section we will discuss the results of the test described in the previous sections.

The result of hypothesis 1) is shown in figures 4, 5 and 6. It is easy to see, that DGO and H_RASP2 almost performs equally according to k -measure. Even with different fill and size, the results are the same. Hypothesis 1) is thereby shown to be correct.

Regarding hypothesis 2), figures 1, 2 and 3 show a small improvement in centralization. The graphs only display the first 10 nodes, and therefore for $N = 10$ (figure 1), the value goes to 1. This indicates, that 100% of the edges are included in the plot. As a result, it shows that for smaller graphs ($N < 50$), DGO delivers a better centralization than H_RASP2, but for larger graphs the result is not conclusive.

The time issue tested in hypothesis 3), is shown in figures 7, 8 and 9. These confirm that DGO is faster than H_RASP2. The hypothesis is therefore shown to be correct.

For hypothesis 4) the result is not yet conclusive since only $N = 6, 7$ and 8 was able to be executed. The preliminary results are shown in figures 10, 11 and 12. These show that DGO is close to the optimal solutions, as found by the brute force. The conclusion is that Dam's Graph Optimization is coming close to the optimal solution, and that it matches H_RASP2 in optimization.

VI. CONCLUSION

This paper suggested a simple taxonomy known as k -measure for evaluating a distributed systems design when it comes to communication and dependencies between modules. Depending on the specific requirements different trade-offs between performance and resource usage can be chosen. Moreover, the algorithm DGO was introduced to create design proposals based on an input graph specifying the communication/dependency requirements. DGO was shown to be faster than the previously proposed algorithm H_RASP2, while producing comparable and near-optimal results. The algorithms were also compared with respect to centralization, where DGO was shown to be just slightly better. This could indicate a limited potential, unless higher k -values are accepted.

ACKNOWLEDGMENT

The work was original a part of a studentproject at Internetteknologier and computersystems at Aalborg University, Denmark. The authors would like to thank the E-study board at Aalborg University and section of Networking and Security for their support.

REFERENCES

- [1] D. M. Moules and G. L. Thompson, "An algorithm for finding a minimum equivalent graph of a digraph," -, vol. -, pp. -, November 1968.
- [2] J. D. Carothers and H. G. Cragon, "Graph-theoretic techniques for computer intraconnection minimization," *IEEE transactions on systems, man, and cybernetics*, vol. Vol. 23, No 3, pp. 876–888, May 1993.
- [3] R. L. R. C. S. Thomas H. Cormen, Charles E. Leiserson, *Introduction to Algorithms*, 3rd ed. The MIT press, 2009.
- [4] H. T. Hsu, "An algorithm for finding a minimal equivalent graph of a digraph," *Journal of the Association for Computing Machinery*, vol. Vol. 22, No 1, pp. 11–16, January 1975.
- [5] S. Martello and P. Toth, "Finding a minimum equivalent graph of a digraph," *NETWORKS*, vol. Vol 12, pp. 89–100, July 1982.
- [6] "Unix time()," 12-09-2010, <http://unixhelp.ed.ac.uk/CGI/man-cgi?time>.

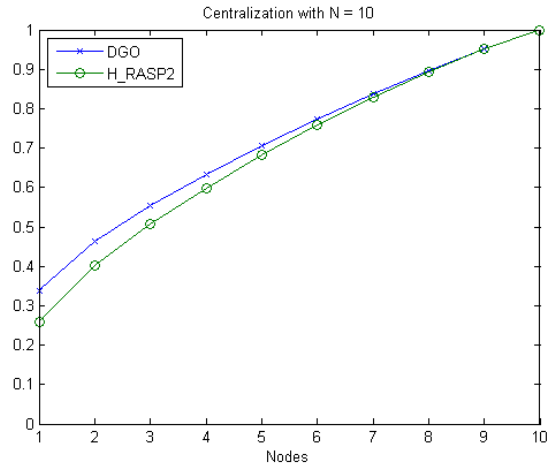


Figure 1. Centralization of H_RASP2 and DGO with $N = 10$. The higher at a low node number, the better

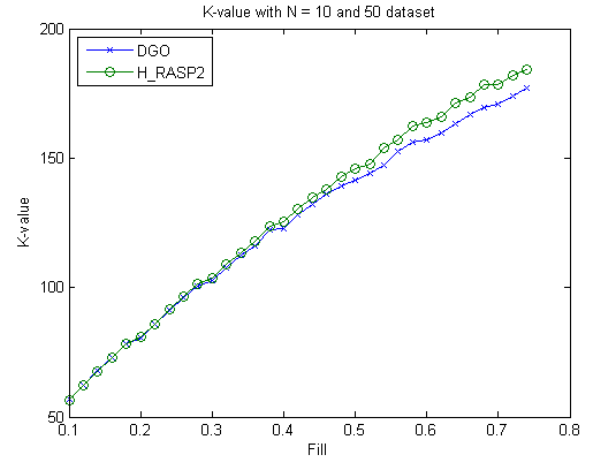


Figure 4. Performance of H_RASP2 and DGO with $N = 10$

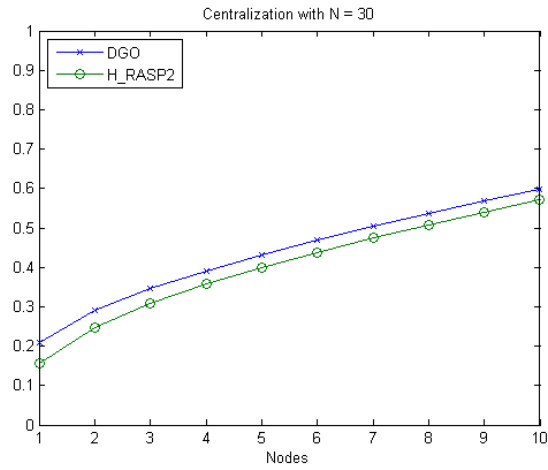


Figure 2. Centralization of H_RASP2 and DGO with $N = 30$

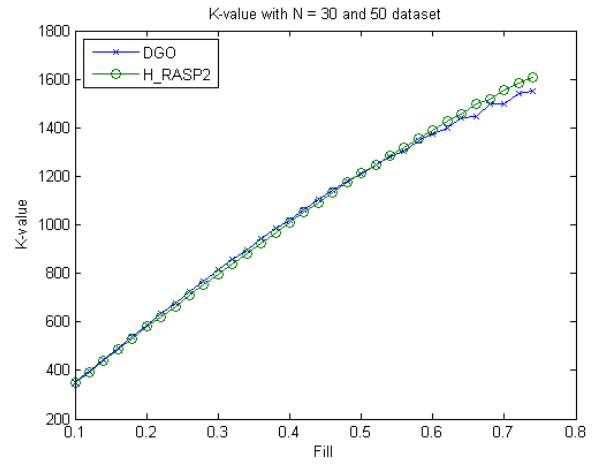


Figure 5. Performance of H_RASP2 and DGO with $N = 30$

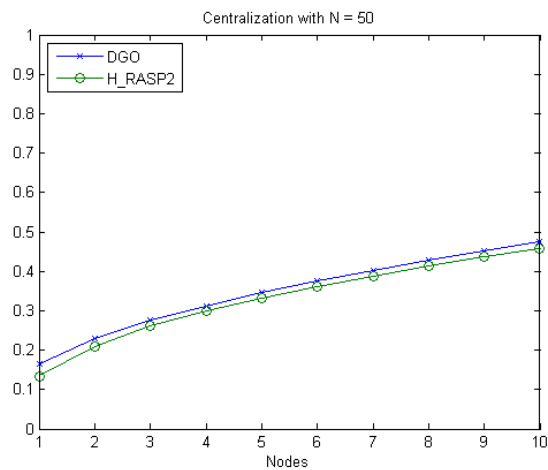


Figure 3. Centralization of H_RASP2 and DGO with $N = 50$

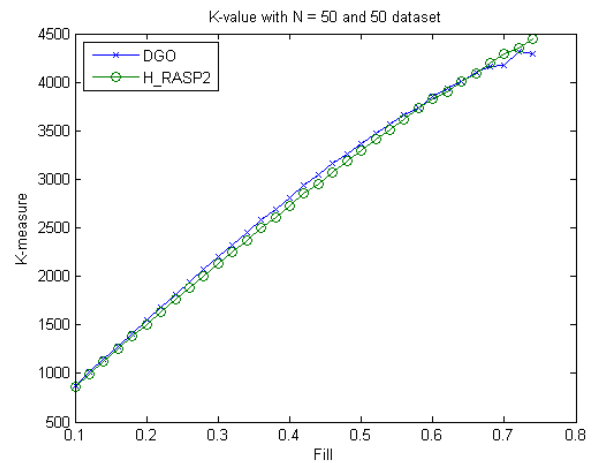


Figure 6. Performance of H_RASP2 and DGO with $N = 50$

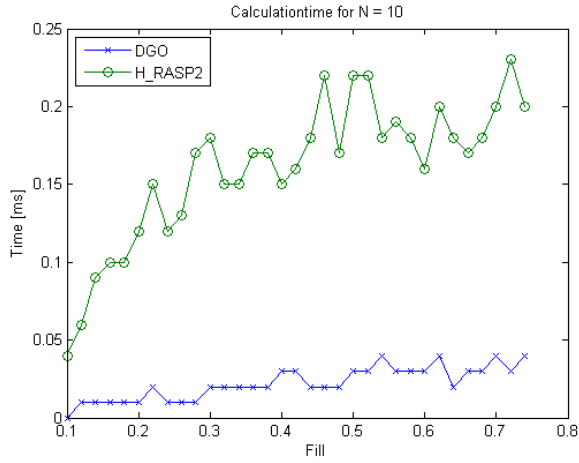


Figure 7. Execution time for H_RASP2 and DGO with $N = 10$

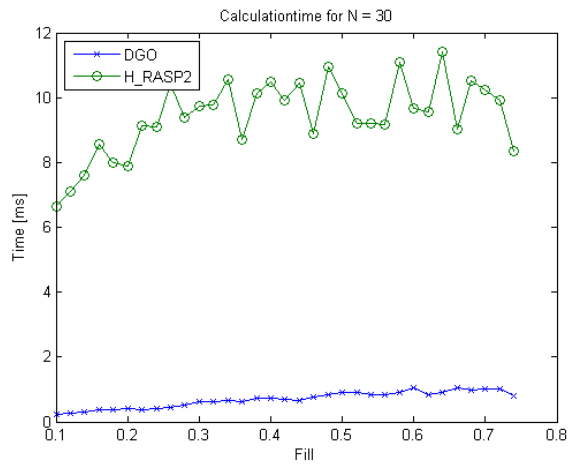


Figure 8. Execution time for H_RASP2 and DGO with $N = 30$

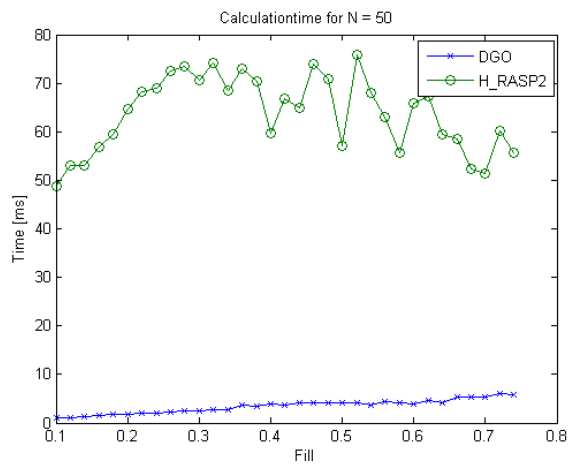


Figure 9. Execution time for H_RASP2 and DGO with $N = 50$

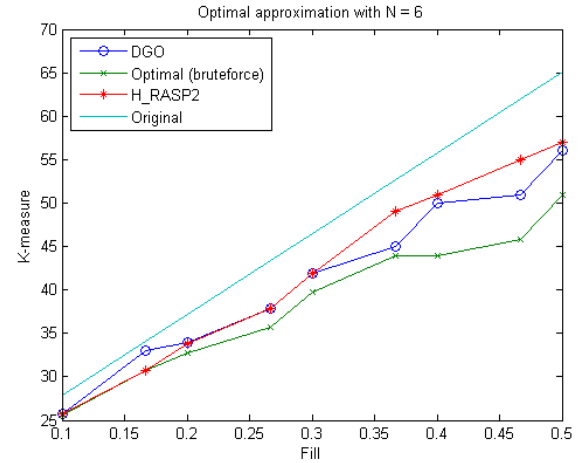


Figure 10. Algorithm approximation to optimal solution for $N = 6$

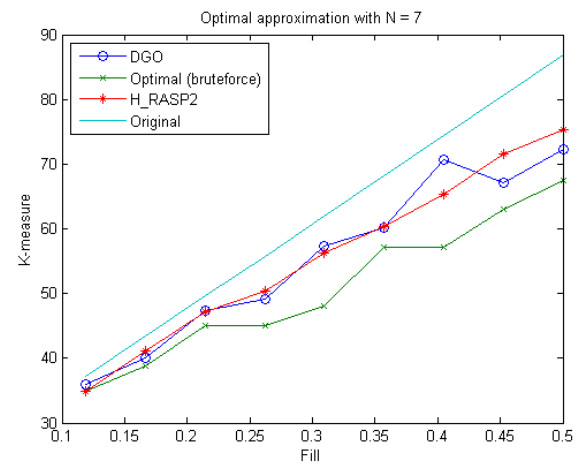


Figure 11. Algorithm approximation to optimal solution for $N = 7$

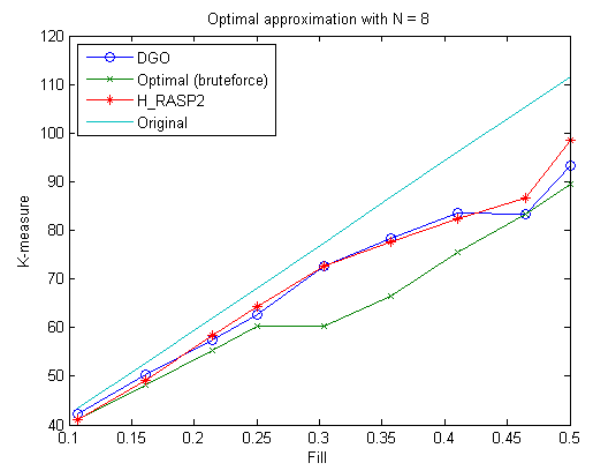


Figure 12. Algorithm approximation to optimal solution for $N = 8$