Solving Serial-Link Manipulator Inverse Kinematics with Search

ALEX SHIE, Carnegie Mellon University

In this paper I redefine the problem of serial-link manipulator inverse kinematics into a search problem and discuss several algorithms and heuristics used to find the best possible solution in the search space as quickly as possible. Most of the results shown are based off of a divide and conquer algorithm that successfully finds solution with minimal error in the search space in logarithmic time.

Categories and Subject Descriptors: **Computers**

General Terms: Algorithms, Performance, Robotics

Additional Key Words and Phrases: Inverse Kinematics, Search, Serial-Link Manipulators

ACM Reference Format:

Gang Zhou, Yafeng Wu, Ting Yan, Tian He, Chengdu Huang, John A. Stankovic, and Tarek F. Abdelzaher, 2010. A multi-frequency MAC specially designed for wireless sensor network applications. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 6 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

In the world of serial-link manipulators (e.g. robotic arms) inverse kinematics refers to the process of finding joint variables such that the end-effector frame of the arm is in a desired position and orientation whereas forward kinematics refers to the process of finding the resulting end-effector frame position and orientation given a set of joint variables. One may expect that solving forward and inverse kinematics problems would be equally difficult, but it turns out that inverse kinematics problems are much more difficult to solve. This is especially unfortunate since inverse kinematics is necessary for arm modeling and control.

There are many methods to solve the inverse kinematics problem, but they each have their own disadvantages. For example, many methods generally require complicated usage of trigonometry and/or run into problems when the end-effector frame is at or near singularities. Common methods include solving the inverse kinematics purely algebraically and symbolically, decomposing the problem into Paden-Kahan sub-problems, and inverting the arm's Jacobian matrix, all of which have the aforementioned disadvantages.

In 2008 Samuel Cubero [Cubero 2008] suggested a new, iterative, general-purpose method of solving inverse kinematics that he refers to as "Blind Search" [1]. Cubero claims that this method does not share the unattractive disadvantages of the more traditional methods, that it can be applied to any serial-link manipulator, and that only the forward kinematics is needed. The basic idea is to consider all combinations of small, fixed changes (or no changes) in each joint in the arm from a given initial configuration, search for and pick the combination that results in the smallest error, set the combination as the new initial configuration if the error isn't small enough, and then repeat until a satisfactory solution is found.

Cubero's Blind Search method has a slight disadvantage, which is that it only works if the desired position and orientation is already very close to the initial configuration. This means that if one needs to find an IK solution to a goal point far away from the initial configuration, then one would need to create a path of discrete path within the manipulator's task space that gradually leads the manipulator from its initial configuration to the goal configuration.

Planning a feasible path for a serial-link manipulator is an entirely different problem in itself. As such, the goal of this project and paper is to develop and investigate a different method of IK search that maintains all of the advantages of Cubero's Blind Search and that does not require path planning or any other significant effort other than finding the forward kinematics for a given serial-link manipulator.

2. DEFINING INVERSE KINEMATICS AS A SEARCH PROBLEM

2.1 Search Space and Tree Structure

The search space of the inverse kinematics problem consists of a set of nodes corresponding to each joint in the serial-link manipulator. The values of the nodes corresponding to a particular joint are uniformly distributed within the joint's full range of possible values. There are edges between any given node and all nodes corresponding to the joints adjacent to the first node's joint, but there are no edges between nodes that correspond to the same joint. For example, consider Figure 1, which shows the general structure of the search tree. Any given node corresponding to joint 2 has edges between itself and all nodes corresponding to joints 1 and 3 and there are no edges between the joint 2 node and any other joint 2 nodes.

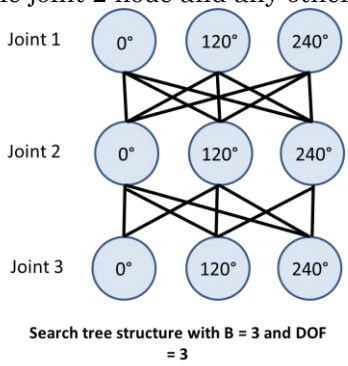

Fig. 1. General search tree structure of inverse kinematics with a branching factor (B) of for a 3 degree of freedom (DOF) manipulator

The size of the search space is dependent on two factors: the number of degrees of freedom (DOF) of the manipulator, and the desired precision of the inverse kinematics solution. The manipulator's DOF, which is assumed to be equal to the number of joints in this paper, is equal to the maximum depth of the search tree. Given that the forward kinematics is used to evaluate and compare possible solutions, every solution must be found at the maximum depth of the search tree. Solutions at shallower depths do not make sense because all joint variables are needed to evaluate the forward kinematics functions. This also implies a breadth first search approach is not possible given the structure of the problem.

The desired precision of the inverse kinematics solution is directly related to the branching factor of the search tree – a larger branching factor results in a solution with greater precision. For example, consider a revolute joint that can rotate in complete revolutions. To produce a solution with 1-degree precision one would need to divide the possible joint values into 360 points, which would result in a branching factor of 360.

Note that although the search space is discretized, the input, i.e. the desired end effector position and orientation, is still drawn from the continuous task space of the manipulator. This means that in many cases the inverse kinematics solution that exactly matches the desired position/orientation does not exist in the discretized search space. Therefore, the goal of the search is to find the set of nodes that

produces the solution closest to the desired position and orientation within the discretized search space. This also implies that desired positions and orientations that are not within the task space of the manipulator are still acceptable inputs and any search algorithm would be capable of simply finding the joint configuration that would result in the manipulator "pointing" in the direction of the desired position and orientation.

2.2 Computed Error and Solution Optimality

Given that the search space only includes values within the manipulator joints' range of motion, every solution in the search space is feasible. However, despite being feasible, the vast majority of these solutions are impractical and useless since they result in an end-effector position and orientation that are far from what we wanted. Therefore, the problem is not to find any solution to the manipulator's inverse kinematics, but rather to find the best solution present in the search space.

To find the best solution present in any search space, one must first define what it means for one solution to be better than another. In the case of inverse kinematics, metric for how good a solution is simply a combination of the distance between the current position and the goal position and the angle between the x and y basis vectors/axes of the current end-effector frame and the goal frame.

The position error E_P is the distance from the goal position and the current position and is fairly straightforward. The orientation errors E_{AX} and E_{AY} for the x and y axes are represented by 1- cos(φ), where φ is the angle between the goal basis vector and the current basis vector of the end-effector frame. This can be further simplified using the definition of the dot product to $1 - V_{XC} \cdot V_{XG}$, where V_{XC} and V_{XG} are the basis vectors for the x axis in the current end-effector frame and the goal frame, respectively. This same can be applied to the y axis. The total orientation error is thus $E_A = E_{AX} + E_{AY}$. The combined, total error is $E_T = K_P*E_P + K_A*E_A$, where K_P and K_A are weighting constants [Cubero 2008]. These weighting constants describe how important accuracy with respect to position and orientation are relative to each other. For example, if finding a solution very close to the goal in terms of distance is much more important than conforming to a particular end-effector frame, then one would need to choose weighting constants such that the position error dominates the orientation error.

3. SEARCH ALGORITHMS AND HEURISTICS

3.1 Brute Force and Divide and Conquer (DaC) Algorithms

For the sake of benchmarking other algorithms and heuristics, a naïve brute force method was implemented first. This solver used a depth-first approach that went through every possible solution in the search space and reported a list of joint assignments that resulted in the smallest error. The speed of this solver will be discussed in later sections, but in summary the brute force approach is unsurprisingly very slow and finding high precision solutions to manipulators with larger DOFs becomes very impractical. The slowness of the brute force approach is attributed to the number of nodes it needs to expand, which is denoted in Equation 1. One can easily see that if a large branching factor were to be chosen then the number of nodes that need to be expanded will increase exponentially for manipulators with greater DOFs.

$$\sum_{i=1}^{DOF} B^i$$

Eq. 1. Total number of node expansions necessary for brute force, where B is the branching factor

The brute force solver starts at the zero configuration for each joint and iterates through the joints' possible values incrementally. The brute force solver could have been improved significantly if the search terminates whenever if finds a solution below a certain error threshold, and if a random joint value ordering were to be used. However, these methods were not explored, partially due to time constraints but mostly due to the presence of other algorithms and methods that are many orders of magnitude faster than the brute force approach.

The main algorithm used in this project was a divide and conquer (DaC) algorithm that iteratively conducted coarse searches that gradually become more and more refined until a precise solution is found as explained in Algorithm 1 and Algorithm 2. First, let us define and review some abbreviations: DOF stands for degrees of freedom; let B stand for branching factor; let D be the number of joint space divisions per iteration, which is in other words the number of coarse samples taken in each joint per iteration.

The algorithm begins by creating the search tree as a global 2D array of nodes. The nodes are initialized with information such as the node's joint value, which joint the node corresponds to, and whether or not said joint is revolute.

Next, the algorithm initiates the search with the first joint in the chain. Each joint in the chain has its own queue of nodes to expand, i.e. values to try. For every iteration the queue is initially populated with only D number of nodes as opposed to B number of nodes as is the case in brute force. Each of the D nodes has values that are uniformly distributed through the joint's range of values. The algorithm initializes the sub-queues of each joint until it reaches the last joint in the chain, which naturally has no children. When the search is at the last joint (i.e. maximum depth has been achieved) the algorithm calculates the error between the end-effector position and orientation resulting from the current assignment of joint variables. The joint variable assignment that results in the least error is kept track of in a global variable.

When all sub-queues are empty, then the iteration is complete. The algorithm proceeds to the next iteration by first reducing the index spacing between nodes. For the first iteration D nodes were uniformly distributed through the joint's range of values, so the index spacing (for the array) was B/D. For each subsequent iteration, this spacing variable is further divided by B. In these iterations the search begins with the manipulator in the configuration specified by the best joint value assignment found thus far. The result is a search that that iteratively narrows in on the general location of the optimal solution. This is illustrated in Figure 2.

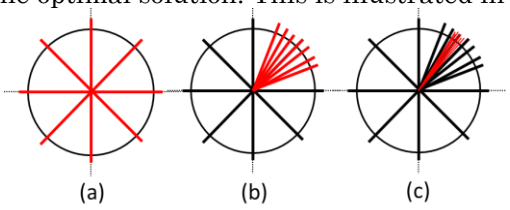

Fig. 2. The figure above illustrates the divide and conquer (DaC) algorithm. The black unit circle represents the possible values for a given revolute joint while the red lines represent nodes added to the queue for a given iteration. The first, second, and third iteration of the algorithm are shown in (a), (b), and (c), respectively. Here D = 8.

ALGORITHM 1. Recursive Depth Search (recursiveDepthSearch)

Input: Goal position and orientation in terms of x/y axes basis vectors, spacing between node values for a given joint, the current node being expanded, the current assignment/path of nodes
Output: Technically none – results are stored in the bestAssignment and bestError global variables

```
if current joint is not the end joint then
      subqueue = initializeQueue(curNode, spacing)
end

repeat
      curNode = subqueue.pop
      assignment.append(curNode)
      recursiveDepthSearch(curNode, assignment, goals, spacing)
      assignment.pop
until subqueue is empty;

if current joint is end joint then
     error = computeForwardKinematicsError(assignment, goals)
     if error < bestError then
          bestError = error
          bestAssignment = assignment
end

return
```

ALGORITHM 2. Divide and Conquer for Inverse Kinematics

Input: Position/orientation goals, search tree, D = the number of samples/divisions per joint
Output: The call to the recursive depth search results in the assignment of the bestAssignment and bestError global variables

```
bestAssignment = {0,0,0…for each joint}
spacing  = branchingFactor/D
repeat
      recursiveDepthSearch(None, assignment, goals, spacing)
      spacing /= D
until spacing < 1;
```

3.1.1. Divide and Conquer. In divide and conquer, the index spacing between nodes to be sampled in the recursive depth search is first determined and then the recursive depth search is called. The recursive depth search sets up sub-queues for each joint except for the last joint, which evaluates the forward kinematics using the assignment/path used to get there. The best solution is kept in a global variable. The divide and conquer approach calls the recursive depth search over and over again, reducing the node index spacing each time until a search with a spacing of 1 has been completed. The best joint variable assignments are used as a starting point for each successive call to the recursive depth search from the divide and conquer function.

3.2 Heuristics and Other Methods

There are three heuristics/methods that were imagined and tested to improve the performance of the DaC solver: constant breadth (CB), dynamically-created tree (DT), and breadth tuning (BT).

As a divide and conquer method, the DaC algorithm described earlier boasts high speeds and great scalability. However, for any divide and conquer algorithm, such as binary search, it is assumed that the list of values being searched is already sorted or the function being sampled is monotonic. In the case of inverse kinematics, the algorithm samples the forward kinematics for a given manipulator. The forward kinematics for just about every manipulator is comprised of a jumble of trigonometric functions, and as such is not monotonic or sorted at all. However, the DaC algorithm ignores this and proceeds as if the forward kinematics were monotonic. This results in, predictably, sub-optimal solutions due to the algorithm "diving" into a local minimum rather than the global. The constant breadth (CB) heuristic serves to counter this problem.

Before the CB heuristic can be described, it is first necessary to define what is meant by "breadth" as it is not the "breadth" in "breadth first search." In Figure 2 one can see that past the first iteration only a fraction of the joint's possible values are explored. The width of this region is determined by number of nodes added to the sub-queues with a given fixed spacing. I noted earlier that the number of nodes added to the sub-queues is D, but that was actually a simplification for explanatory purposes. The actual number of nodes added is 2*(D/2 + Br) – 1, where Br is what shall be known as breadth. In regular DaC, Br is 1. This expression was used to ensure that the number of nodes added to the sub-queue is odd so that there will be an even number of nodes on either side of the node corresponding to the best assignment. Note that the simplified expression D + 2*Br – 1 does not work in the case that D is odd (D/2 is actually Floor[D/2] in many programming languages). Increasing Br by 1 means that one more node on both "sides" of the node corresponding to the best assignment will be added to the queue. Given that there is constant spacing between nodes in terms of array indices and values, this results in a slightly broadened search (Figure 3). The constant breadth (CB) heuristic is to simply choose the value of Br that results in the best balance of speed and accuracy.

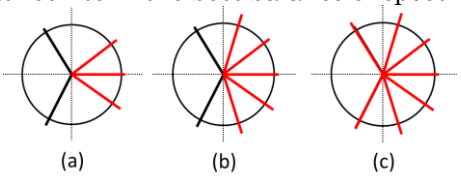

Fig. 3. The figure above illustrates the effects of increasing the breadth variable (Br) during the second iteration of the divide and conquer algorithm (D = 3). Red lines correspond to nodes added to the queue for a given revolute joint. Br = 1, 2, and 3 for (a), (b), and (c), respectively.

It turns out that varying Br results in large tradeoffs of speed and accuracy within a small range of values, so it would be nice if we could use fractions in Br rather than just whole numbers to find a better balance of speed and accuracy. Unfortunately, Br needs to be a whole number, but the next best thing also works: varying Br depending on the joint and DOF, which is what breadth tuning (BT) is all about.

In breadth tuning (BT), one needs to examine the manipulator in question and determine whether or not it would be okay to sacrifice accuracy for some joints for the sake of speed of computation. For example, consider the elbow manipulator in Figure 4. If one is much more concerned with having a solution that matches with the goal position rather than the goal orientation, then one can easily see that the first three joints (from the base) are more important than the latter three because the

position of the end effector is dependent on only those three joints. Similarly, the end effector orientation is dependent only on the latter three joints. Thus, one would be able to get away with having higher Br on the first three joints and lower Br on the last three joints. Note that for manipulators with very few DOF, e.g. the two link planar manipulator, one can get away with CB with a larger Br for all joints since it is fast enough as it is anyway.

So far all of the heuristics and methods have been concerned with the speed of the solver and the accuracy/optimality of the solution. The dynamically-created tree (DT) method addresses the potential issue of memory. Even with moderately high solution resolution, i.e. branching factor, most modern computers can handle creating every node in the search tree without any problems. However, for very high solution resolutions (e.g. when the branching factor is literally in the millions) or when working with an embedded device without as much memory, memory usage can be a concern. Finding a way to reduce memory usage is especially relevant when one realizes that in a typical DaC search the majority of nodes aren't expanded – they aren't even added to the queue since they aren't even needed.

In regular DaC, all the nodes of the search tree are created once during the preprocessing phase with global scope (or equivalent) and are used repeatedly for all subsequent searches. In DT, a data structure containing all of the nodes in the search tree does not exist. Instead, the algorithm creates nodes as necessary for each call to the recursive depth search function. This is possible and easy due to the fact that the values of all nodes are implicitly known and in order from least to greatest value. For example, consider the case when B = 360 and D = 3 for a given revolute joint. We know that if the nodes existed in an array that the node in index 0 has a value of 0, the node in index 1 has a value of 1 (degree), the node in index 2 has a value 2, etc. Therefore, if we want to add nodes to the sub-queue we need to add the nodes with indices 0, B/D, and 2*B/D, which are 0, 120, and 240. But given that these nodes don't exist yet, we have to create three nodes and give each the values the 0^{th}, 120^{th}, and 240^{th} nodes would have had. This value is just 360/B*index, and so we get 0 degrees, 120 degrees, and 240 degrees, which are the nodes we wanted. Since these nodes are created within the recursive function call, they are stored on the stack and have local scope. As such there is a loss of efficiency in that a particular node that is only created once in DaC can be created many times in DT. Despite this, DT still results in a dramatic reduction in memory usage, as will be seen later.

4. EXPERIMENTAL SETUP AND RESULTS

4.1 Experimental Setup

The all code was written in Python and executed on a laptop with a 2.0GHz dual core processor and 4 GB of RAM within an Ubuntu operating system. Tests were done on comparing average error and speed of the brute force method, DaC with CB with various breadth values, DaC with BT, and DaC with BT and DT across several DOF. Separate tests were conducted to compare various results of varying D, and varying B with DaC with and without DT. All tests consisted of picking random joint values, plugging them into the forward kinematics of a particular manipulator, and then feeding the resulting end-effector positions and orientations as input into the inverse kinematics solver. Most tests consisted of running the solvers for 100 random points, but there were some that were limited to 10 points due to lengthy run times.

Each of the solvers was tested on 4 distinct serial link manipulators (Figure 4): a 2 DOF two link planar manipulator, a 3 DOF cylindrical manipulator, a 4 DOF SCARA manipulator, and a 6 DOF elbow manipulator. The link lengths for these manipulators were chosen to be similar to one another to make their respective end-effector position errors comparable. For reference, the link lengths for the two link planar manipulator were .5m for the first link and .25m for the second link.

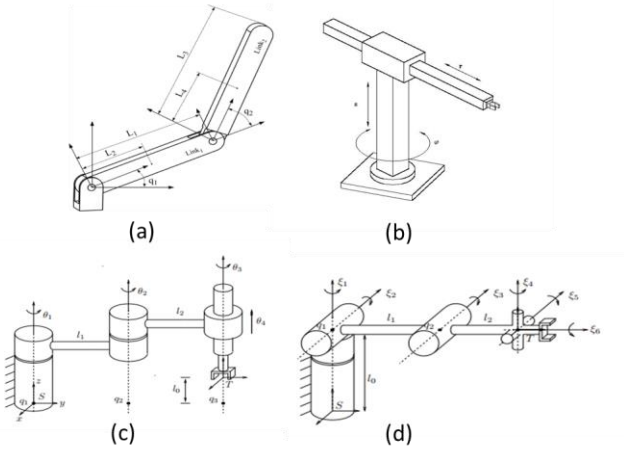

Fig. 4. The figure above shows the four kinds of manipulators whose forward kinematics were used for testing: (a) two link planar manipulator, (b) cylindrical manipulator, (c) SCARA manipulator, (d) elbow manipulator with 2, 3, 4, and 6 DOF respectively.

4.2 Results

The first few tests conducted were to see what kinds of branching factors (B) were feasible for the brute force approach. It turned out that the 2 DOF manipulator could handle a B of 720 in a few seconds, which results in a solution resolution of half of a degree. For larger DOF manipulators the largest practical B dropped exponentially, with some points taking several minutes to solve. Eventually a test was conducted with B = 60 across multiple manipulators to compare the number of nodes expanded, which is directly related to the speed of the solver, by the brute force solver and the DaC (D = 3) + DT + BT solver. The results can be seen in Figure 5. Note that the 6 DOF manipulator was not tested with the brute force algorithm given that the brute force algorithm took several minutes to expand over 10 million nodes to solve a single point for a 4 DOF manipulator. For all of the following tests involving errors, only position error is shown in graphs and discussed since orientation error turned out to vary in the same way position error varied. In other words, there were no cases in which a solution produced high position error but low orientation error, or vice versa.

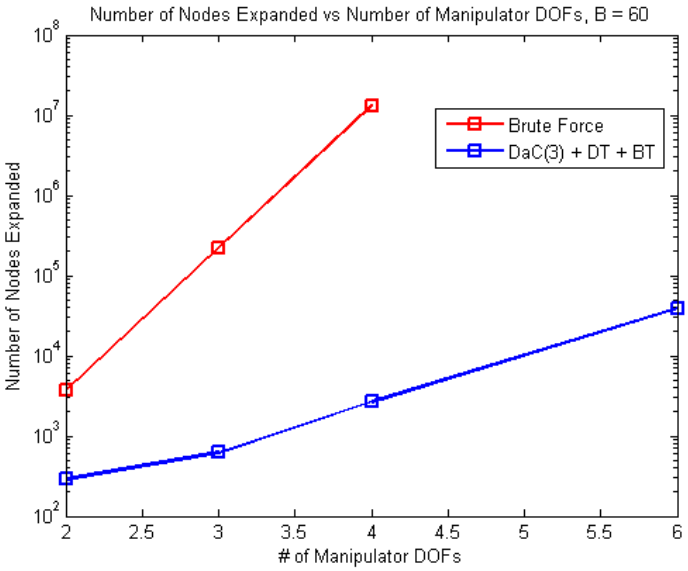

Fig. 5. Brute force algorithm vs. DaC (D = 3) + DT + BT, with the number of nodes expanded to solve one point vs. the DOF of the tested manipulator

The next few tests conducted were to see the effects of varying D for the manipulators with 2, 3, and 4 DOFs, with D ranging un-uniformly from 3 to 32 and with B set at 7200, which results in a solution resolution of 1/20th of a degree. As can be seen in Figure 6 (D is labeled as Number of Joint Space Divisions per Iteration), the time it takes to solve a single point increases quite rapidly as the DOF of the manipulator increases. The 3 and 4 DOF manipulators have fewer data points than the 2 DOF manipulator because the missing points would have been extremely impractical to evaluate due to run times. Now consider Figure 7, which arguably has the most surprising result of the project. As D increases within the range of 3 to 32, we see that the resulting average position error actually tends to increase, which is the opposite of what one may expect. One point that literally stands above the others is the point corresponding to D = 4 for the 2 DOF manipulator. This high error wasn't the result of bad luck – multiple tests were re-run to verify the consistency of this error. The reason for this is not well known and is further discussed later in this paper. One may conclude from these tests is that the optimal value of D is clearly 3, and as such D is 3 for the rest of the tests that did not vary D. It is also worth noting that other tests with the 2 DOF manipulator were run with D increasing past 32 to 180 and 360. By the time D reaches these larger values there is an overall decrease in the average position error, but by then there is a very significant increase in run time.

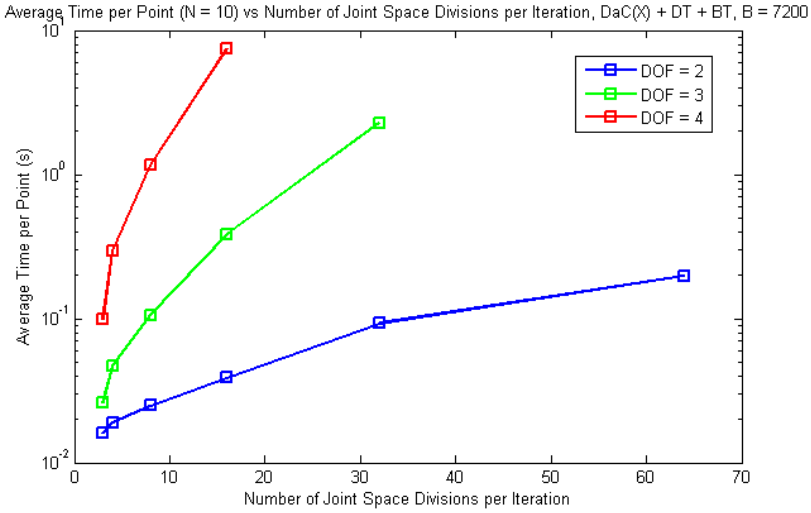

Fig. 6. Average runtime per point vs. D for various DOFs

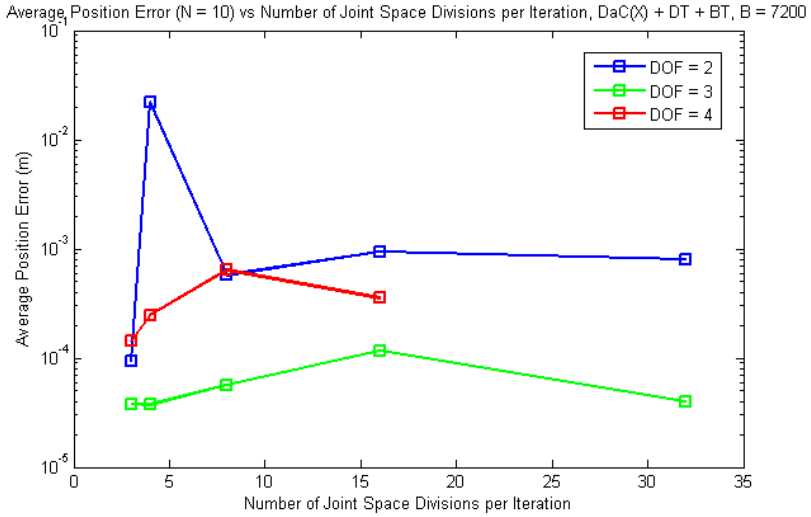

Fig. 7. Average position error per point vs. D for various DOFs

The next tests were to compare speed and accuracy of the various heuristics developed for this project (i.e. CB, DT, BT). For these tests D was held constant at 3 and B was set to be 7200. B is set to 7200 for many of the tests because it was thought that a B of 7200 results in a high enough solution resolution and it leads to a node count that is high enough to make the tests interesting and informative when it comes to speed. Various breadth values of 1, 2, and 3 were also tested for CB. Also recall that DaC + CB(1) is essentially DC without any heuristics other than the value of D. Figure 8 shows the results of the tests measuring position error. As one can see, the worst solver by far is DaC + CB(1) while the rest have comparable performance. Other than the CB(1) solver, all solvers were able to achieve position errors less than or close to less than a millimeter, which is pretty good. Note that the position errors were actually average position errors in a run of 100 random points, which were different for each solver tested. This explains the apparent difference in performance between DaC + BT and DaC + DT + BT, which result in the exact same errors and solutions given that DT does not actually affect the search process in any way except

in terms of speed and memory usage. Another thing to point out is the relatively extreme errors produced by DaC + CB(2) and DaC + CB(3) for the 2 DOF manipulator. These results will be discussed later in the paper.

In terms of speed, we see clear trends due to mathematically consistent changes in node counts. As seen in Figure 9, DaC + CB(1) is, although very error-prone, by far the fastest of the solvers, beating the next best ones by almost an order of a magnitude. Also note that DaC + CB(3) is by far the slowest solver, being slower than the rest by at least an order of magnitude. It is interesting to see such wide ranges in performance between Br = 1 and Br = 3, and it is these results that prompted the creation of BT, which has been shown to maintain a balance of speed and accuracy.

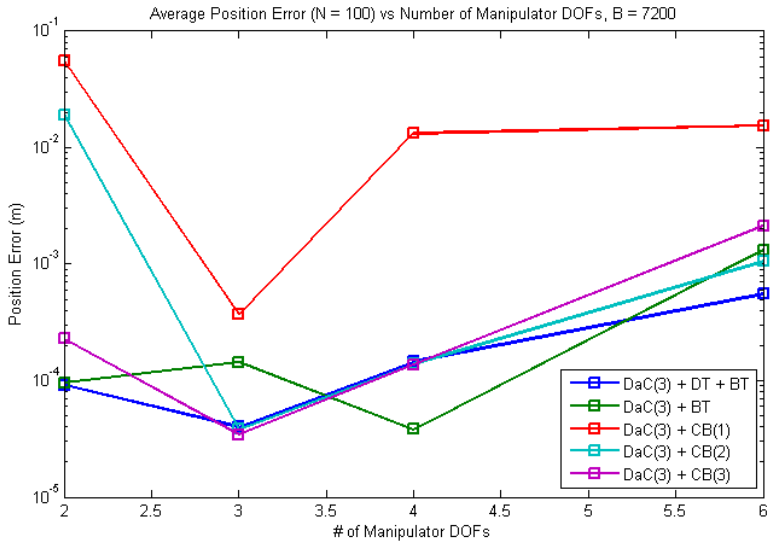

Fig. 8. Average position error per point vs. manipulator DOF for all heuristics

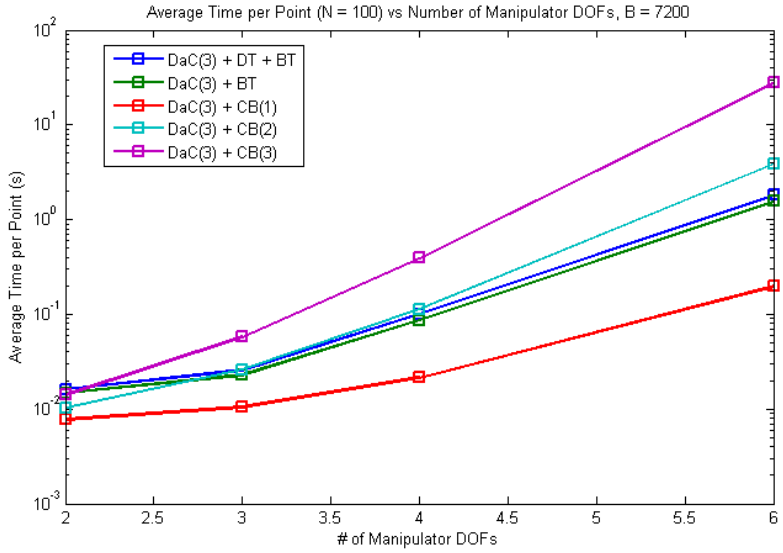

Fig. 9. Average run time per point vs. manipulator DOF for all heuristics

The next test was to compare memory usage (measured as the number of nodes created before and during the search for a single point) between DaC + BT + DT and DaC + BT. The results are shown in Figure 10. Given that the number of nodes created is exponential (see Equation 1) as the branching factor increases, the line for DaC + BT appears to be linear when both axes are logarithmic. On the other hand, we see that the solver using DT creates much fewer nodes when the branching factor exceeds 10000, but before that point it turns out that the solver without DT creates fewer nodes simply because any given node isn't created more than once. Note that the solver without DT has fewer data points because my computer began stalling when the solver attempted to create over 4.7 million nodes.

Speed was also compared for the tests between the solver with DT and without DT. As one can see in Figure 11 and Figure 9, the solver without DT performs slightly faster than the one without DT. This graph mostly serves to prove the speed scalability of DaC. For example, the solver with DT experienced a speed decrease by a factor of 7 while the branching factor increased by a factor of over 4.7 million. This shows that the time complexity of the DaC solver(s) is logarithmic in nature with a base of D.

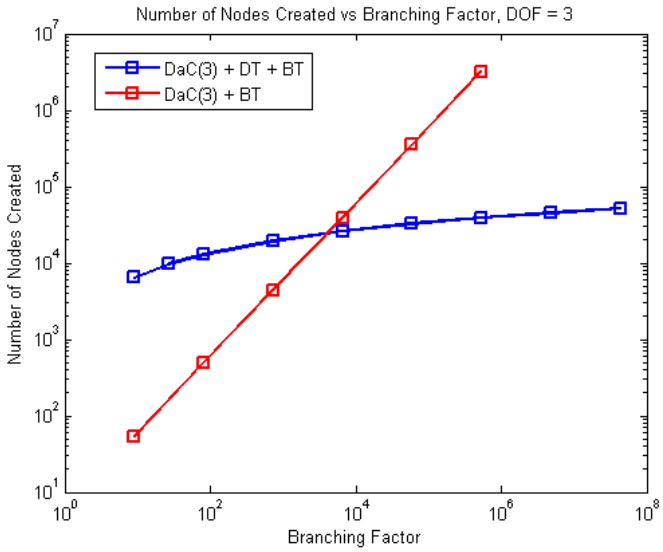

Fig. 10. Number of nodes created for a single run vs. the branching factor of the search tree

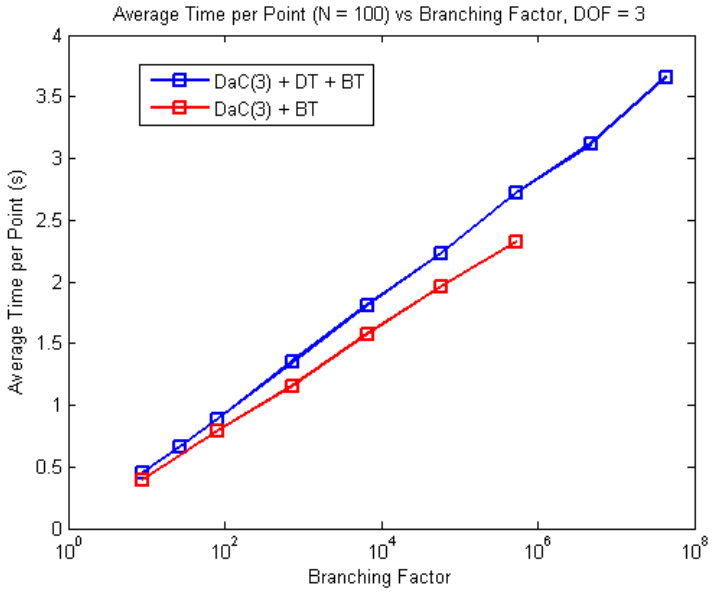

Fig. 11. Average run time per point vs. the branching factor of the search tree

5. CONCLUSIONS AND DISCUSSION

As seen from several of the tests, there were some situations in which the 2 DOF two link planar manipulator produced errors much larger than the older manipulators did with the same solvers. I do not have a full explanation of this phenomenon, but my guess is that it has something to do with the forward kinematics function that is unique to each manipulator. It may be the case that there are certain regions in the two link planar manipulator's forward kinematics that result in local minima that are close to the global minimum and that mislead the solver. This would explain the strange data presented in Figure 7, which shows a sharp increase in error when D = 4 for only the two link planar manipulator.

From the results shown and discussed, we can conclude that the DaC inverse kinematics solver with some heuristics applied is a viable solution to the problem of inverse kinematics for serial-link manipulators. The best of the solvers developed for this project can consistently produce solutions sub-millimeter accuracy on average. These solvers can also produce such solutions in less than 2 seconds per point for a 6 DOF manipulator, which is reasonably fast. However, it is still an order of magnitude or two slower than solving inverse kinematics directly with twists and matrix exponentials. For example, it takes MATLAB approximately 10 seconds to solve 500 points accurately for a 7 DOF manipulator.

The DaC algorithm can certainly be further improved. For example, one heuristic one may be able to look into is to break up the manipulator into separate chains. This would be possible only for certain cases in which the end-effector position and orientation only depend on a subset of the joints available. For example, one could try to run the solver on the first 3 joints on the SCARA robot and then run the solver for the last joint after a solution for the x-y position has been found. Another thing to note is that all of the DaC solvers used for this project store the possible joint values within nodes, which are distinct data structures. Having a large number of data

structures around surely has some overhead cost, so one may wish to investigate ways of dealing with numbers without the objects.

REFERENCES

Mark W. Spong, Seth Hutchinson, and M. Vidyasagar. 2006. *Robot Modeling and Control*. John Wiley and Sons Inc. Chapter 3.

Richard M. Murray, Zexiang Li, and S. Shankar Sastry. 1994. *A Mathematical Introduction to Robotic Manipulation*. CRC Press. Chapter 3.

Samuel N. Cubero. 2008. Blind Search inverse kinematics for controlling all types of serial-link robot arms. *Mechatronics and Machine Vision in Practice*, 229-244.