Automatic Planning of Nanoparticle Assembly Tasks

J. H. Makaliwe and A. A. G. Requicha
Laboratory for Molecular Robotics
University of Southern California

Abstract

This paper presents a practical planning system for 2-D assembly tasks at the nanoscale. The planner covers a whole range of problems in planning the assembly of nanoparticle patterns, including object assignment, obstacle detection and avoidance, path finding, and path sequencing. We describe algorithms based on optimization theory and visibility graph construction, and show how to integrate them into a comprehensive planner that has a low-order polynomial complexity and produces good results. We provide theoretical analysis and experimental results. This is a first step towards automating assembly tasks in nanorobotics.

1. Introduction

Robotic manipulation and assembly of objects at the nanoscale [Requicha, 1999] is a branch of nanorobotics that has generated considerable interest and promises to produce revolutionary advances in miniaturization. The construction and subsequent linking of patterns of nanoparticles by using the tip of a Scanning Probe Microscope (SPM) as a robotic end effector has been demonstrated [Requicha *et al.* 1998, 1999]. However, building patterns with large numbers of particles interactively is tedious and tends to be inaccurate, because humans have difficulties in performing precise positioning tasks. Automating the process of moving a large number of objects (potentially hundreds or thousands) in near real time is necessary to make such nanorobotic tasks possible.

It is well known that the general problem of motion planning in the presence of movable objects is PSPACE-hard [Wilfong 1988], and therefore optimal solutions are generally impossible to obtain. Thus, we must find solutions that are *satisfactory* by developing *practical algorithms*, i.e., heuristic algorithms that have good performance in terms of planning time and provide good solutions (albeit not optimal) in terms of execution cost.

Previous research in robotics has mostly avoided issues of manipulation or motion planning for a large number of objects amidst obstacles, mainly because of the complexity involved. Most of the work reported in the literature deals with motion planning for one or a few movable objects amidst obstacles. Closer to our problem is the research of [Ben-Shahar & Rivlin 1998], who considered several practical algorithms for rearranging many (about 30) objects by pushing. Their problem formulation is very similar to ours. However, our work presents different algorithms and covers several tasks not addressed in their project.

Several researchers have considered the use of heuristics to attack problems involving many objects. For instance, [Barraquand & Latombe 1990, 1991] used a Monte-Carlo algorithm on a potential field over a discretized version of a configuration space, and [Chen & Hwang 1991] considered a heuristic approach for solving problems with many movable obstacles. However, these projects deal mainly with the task of path planning in the midst of many objects, whereas our project is concerned with assembly, which includes several other issues in addition to path planning. Furthermore, we employ different algorithms for the tasks that they also cover.

The main contribution of this paper is the development of a practical planning system that covers the whole range of problems in 2-D nanoparticle assembly planning, including object assignment, obstacle detection and avoidance, path finding, and path sequencing. We introduce algorithms based on optimization theory and visibility graph construction and show how to integrate them into a comprehensive planner that has a low-order polynomial complexity. Insofar as we know, this project is the first attempt to do comprehensive high-level planning in the field of nanorobotics.

2. Problem Definition

We abstract the problem of constructing nanoparticle patterns with an SPM as follows. The system consists of a robot, a set of movable objects, and a set of obstacles.

- Let R be a robot, which can assume one of two states: active or non-active. While active, it will push any movable object whose center (or, more generally, reference point) lies in the path the robot is moving along. While not active, its movement does not have any effect on any object or obstacle. In SPM robotics, the robot is the tip, which can be activated by turning off the vertical feedback [Requicha *et al.* 1998]. We assume that robot paths are piecewise linear.
- Let $O = \{o_1, o_2, \ldots, o_n\}$ be a set of n identical objects, each of them movable (i.e., pushable) by R. For simplicity, each object is assumed to be circular. For each object, the current location is given.
- Let $P = \{p_1, p_2, \ldots, p_m\}$ be a set of m polygons that are not movable by R. They are obstacles that must

be avoided when the robot is moving in an active state. For each polygon p_j, the user can specify a minimum safe distance – the closest distance allowed for the robot to move in the neighborhood around the polygon. In the implementation, the polygons are represented as lists of edges. The polygons represent either objects that are actually immovable, or movable objects that have already been placed and are considered immovable to avoid interfering with previously-achieved results.

The task of pattern construction can be defined as follows:

Given a description of P, initial locations of each object in O, and a set of destination locations D, find a sequence of pushing actions by a single robot R that will rearrange the objects such that an object is put in each of the destination points in D. Each pushing action must be done without coming closer than the minimum distance to any polygon in P.

Since we are using a single robot, the pushing movements should be done sequentially. In addition, we assume that the sequence of pushing actions to move a particular object from its initial to its final location is not interrupted by pushing actions involving other objects.

We are interested in finding a sequence of motions that accomplishes the construction task with minimum effort (where the effort is characterized by the shortest distance or some other measure). We know that optimality is not practically achievable, and therefore we have to turn to practical, heuristic-based techniques. An instance of the task is given in figure 1a.

In the actual environment, the robot is the tip of a Scanning Probe Microscope (SPM). For technical details and references on nanomanipulation, see [Requicha 1999, Requicha *et al*. 1998, 1999]. Physically, the device looks like an arm manipulator. However, since the distances traveled by the tip are large relative to the environment, it can be considered as a mobile robot with capabilities to push objects in the environment.

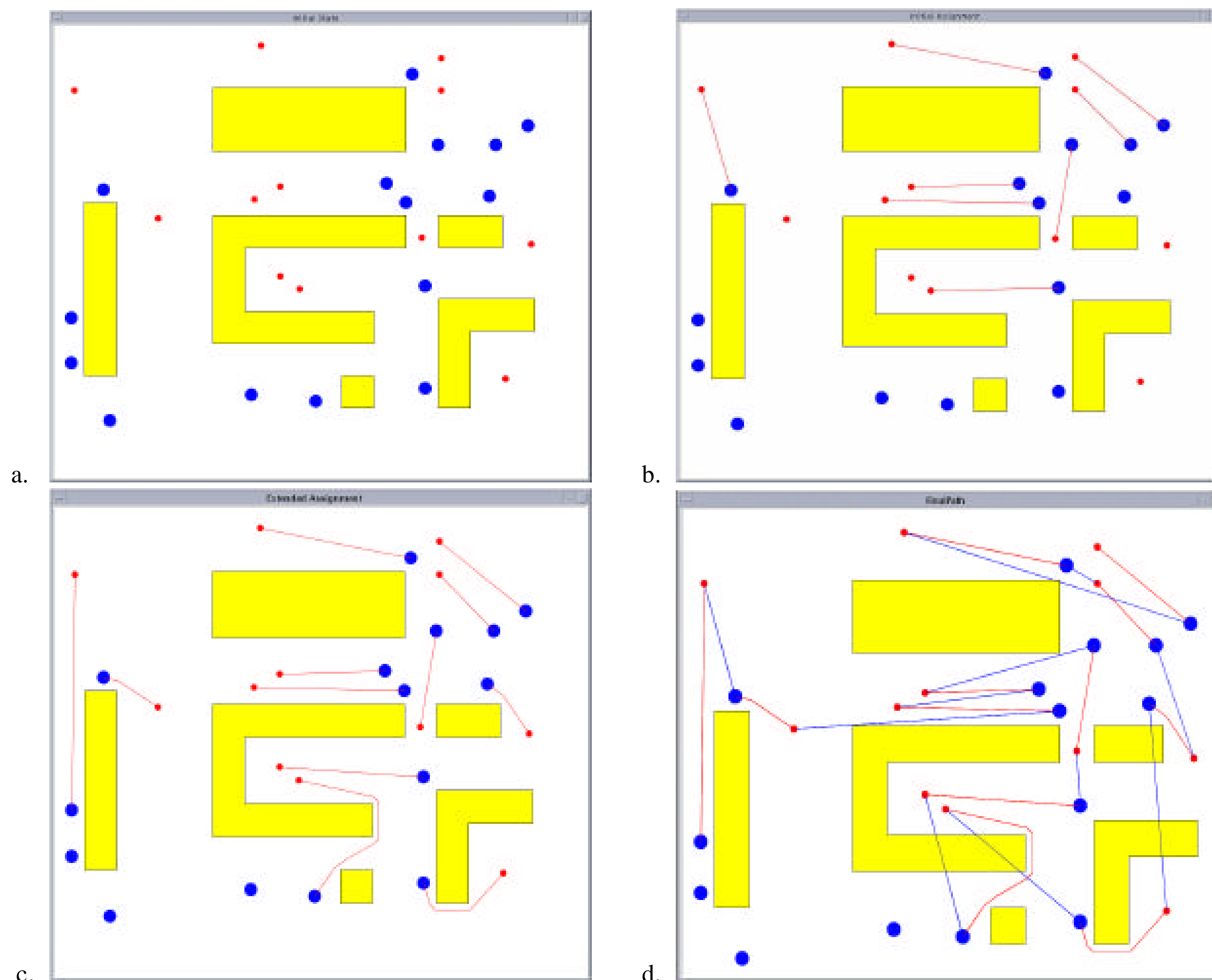

Figure 1 – (a) Initial problem, (b) result after the assignment of direct paths, (c) after the extended assignment of paths around obstacles, and (d) final overall sequence. Bigger circles are objects, small dots are destinations.

All the objects and obstacles reside in a sample prepared by chemical means. The input for the planner comes from imaging the sample. Certain well-known image processing techniques such as the Hough Transform provide practical means to transform the input into the representation described in the problem definition. The output is used to control the SPM tip with the Probe Control Software developed by our group. In this paper, we focus on *planning* and assume that other modules take care of input and output processing.

3. Overview of the Planner

The input consists of the set of edges in P (including the minimum safe distances), information about objects in O (including initial locations, object types, etc), and the set of destination locations D. The planner divides the task into several sub-tasks.

Since the objects are identical, the first sub-task is the *assignment* of objects to destination locations. In the first step, we assign objects to destination locations following *a direct path*, i.e., a single linear path from the initial position of the object to its final destination. The assignment is done such that the total cost for moving all objects is minimal. (In case there are several types of objects, each with its respective set of destination locations, we apply the assignment procedure separately for each type.) The result of this assignment step is a set of paths connecting initial and destination locations. This step, and the related *bipartite graph construction* step, are further discussed in section 4.

The first step might fail to assign objects for some destinations because some objects might be hidden behind obstacles, making it impossible to find direct paths from them to the destination locations. Hence, the next sub-task is to find paths through which the robot can navigate around obstacles while taking objects to unassigned destination locations. This is the *path finding* step, which may produce several alternative paths. To select the best path, this result is added to the previously-found direct paths by the *extend bipartite graph* function, and fed back into the *assignment* function. These functions constitute the second step and are elaborated in section 5.

The combined result of the *assignment* and *path finding* steps is either an assignment of a different object to each destination or the report of failure, in which case this planner cannot solve the problem. Upon successful completion we have a list of paths, each connecting one object and one destination location. The next sub-task is to impart a total ordering to these paths and produce an overall path that can be followed by the robot. The main considerations in this ordering are possible interference among the individual paths and optimality of the overall path. This task is handled in the *sequencing* step, which is described in section 6.

The final output is a linear sequence of pushing actions and intermediate robot movements. The overall algorithm can be described as follows.

Planning (O, P, D)
// assume D ¹ Æ

    $BG \leftarrow$ **BipartiteGraph**(O, P, D)
    $MatchSet \leftarrow$ **Assignment**(BG)
    $DNC \leftarrow$ members of D that are not covered in $MatchSet$

    If DNC ¹ Æ **then**
        $NewPaths \leftarrow$ **FindPaths**(O, P, DNC)
        $BG \leftarrow$ **ExtendBipartiteGraph** (BG, $NewPaths$)
        $MatchSet \leftarrow$ **Assignment** (BG)
        $DNC \leftarrow$ members of D that are not covered in $MatchSet$
        if DNC ¹ Æ **return** FAILURE
    end if

    Overall Path = **Sequencing** (*MatchSet*)
    If *OverallPath* = Æ **then return** FAILURE
    else return *Overall Path*

end Planning

4. Assignment of Objects to Destinations

We model the assignment of objects to destinations as a weighted bipartite graph matching problem. First, we create a bipartite graph $G = (V, E)$ with node partition $V = L \cup R$, where:

- Each node in L represents an object.
- Each node in R represents a destination.
- Each edge between a node in L and a node in R represents a safe connection between a represented object and a destination, where a connection is "safe" if there is a linear path between the object and the destination that is outside the minimum safe distance from each polygon.

  Define the weight for such an edge according to the cost criterion, e.g., as the Euclidean length of the path.

Checking whether a path is safe is done by testing the distances between the path and each edge of the polygons. This can be done efficiently with standard geometric algorithms.

The *BipartiteGraph* function runs in $O(nde)$ where n is the number of objects, d the number of destinations, and e the total number of edges in P. In practice, we can greatly reduce the number of tests by only considering

polygon edges that are in the neighborhood of the path being considered. This can by done by employing appropriate data structures to organize the edges of the polygons.

Having modeled the problem as a bipartite graph, the assignment of objects to destinations now becomes an instance of the classical problem of weighted bipartite matching: to choose a minimum-weight set of edges in the bipartite graph. The *Assignment* function does this by implementing the *Hungarian Algorithm* [Knuth 1993]. It returns a set of direct paths between the objects and the destinations and runs in $O(n^2d)$. The Hungarian Algorithm guarantees optimal graph matching (hence, optimal overall assignments for direct paths). Note that the weight between edges can represent any appropriate cost criterion.

Figure 1b shows the result of *BipartiteGraph* construction and running the *Assignment* function on the situation in figure 1. The algorithm assigns as many objects as possible with safe straight-line paths.

5. Finding Paths Around Obstacles

Note that in figure 1b some destinations do not have an object assigned to them because some objects are hidden behind obstacles and cannot be found by the *Assignment* function, which considers direct pahs only. Hence, we need another step to find paths that navigate around obstacles. This is done by the *FindPaths* function, which tries to find paths using visibility graph construction and a shortest path algorithm.

First, we consider the destinations that do not have objects assigned to them. This does not guarantee optimality, but it makes sense heuristically to focus the effort in the destinations that fail.

Next, we use the *visibility graph* method [Latombe, 1991] to find possible paths between objects and unassigned destinations. Since the robot can only push following piecewise linear paths, a slight modification is implemented in constructing the boundary of the configuration space around convex corners. Ideally, for moving circular objects, the boundary of the configuration space around such corners is a circular arc. We approximate this with piecewise linear segments. This is illustrated in figure 2.

Using the visibility graph, we find shortest paths connecting each unassigned destination to the initial positions of each object. To find the paths, we run Dijkstra's *single-source shortest path* algorithm [Cormen *et al*. 1990] for each unassigned destination.

The time complexity of the visibility graph construction is $O(n^2)$. The Dijkstra algorithm is run for each unassigned destination, with time complexity $O(dn^2)$. The entire running time of *FindPaths* is thus $O(dn^2)$.

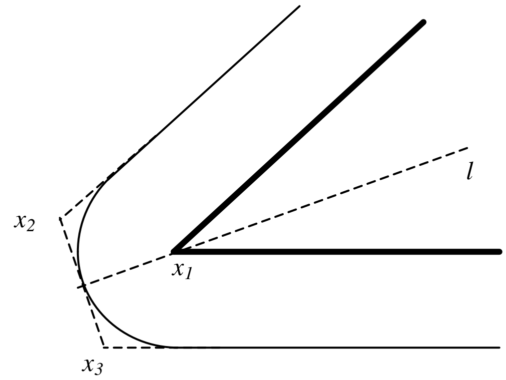

Figure 2. Approximating the boundary of the configuration space around convex corners. The bold line is the real boundary of the obstacle. The circular arc around the corner x_1 is the ideal configuration space boundary. We approximate it with the segment $x_2 - x_3$, which is tangent to the circular arc at the point where it meets the angle bisector *l*. Points x_2 and x_3 becomes vertices in the visibility graph.

Corresponding to each new path found by *FindPaths*, a new edge is added in the bipartite graph constructed in the *BipartiteGraph* construction step. This extension is performed by the *ExtendBipartiteGraph* function. The extended bipartite graph is then fed back into the *Assignment* function to find assignments (matchings) that incorporate both direct paths and paths that navigate around obstacles for destinations that cannot be reached with direct paths. With the extended bipartite graph, *Assignment* still runs in $O(n^2d)$. Continuing from figure 1b, figure 1c shows the result of applying the new assignments after incorporating the results of *FindPaths.* Note that some previous matches might get reassigned.

The combination of *FindPaths* and bipartite matching on the extended bipartite graph tries to approximate the optimal solution by:

- using the visibility graph method, which guarantees shortest paths around obstacles for unassigned destinations, and
- applying the Hungarian Algorithm, which guarantees an optimal assignment that takes into account direct paths and the additional paths found by the visibility graph method.

The only possibility of missing an optimal assignment for a destination stems from the priority given to direct paths. For destinations for which a direct path to an object has already been found, we don't try to consider alternative paths (to other objects) around obstacles, although they might actually be shorter than the direct path. Hence, by giving priority to direct paths, the algorithm might miss a shorter (indirect) path.

However, since pushing actions following paths around obstacles involve more complicated maneuvering than direct paths, the priority given to direct paths is justified.

We have also explored an interesting alternative for implementing *Findpaths* by adapting the idea of the *probabilistic roadmap planner* (PRM) [Kavraki *et al*. 1996]. It works by incrementally adding random points that serve as intermediate points between destinations and objects. The random points are thrown in the neighborhood of the destination, and the neighborhood is incrementally increased until we find a path that connects the destination and an object via the intermediate points.

The PRM approach is better than the visibility graph method in cases where the objects are actually close to the destinations (*e.g*., if they are just around the corner). A path connecting such points is quickly found by working incrementally, without the burden of computing the visibility graph for the entire system. However, our experiments indicate that the visibility graph method is more efficient in the general case. In addition, it is *complete* while the PRM is only *probabilistically complete*, and it guarantees finding shortest paths, whereas the PRM does not.

6. Sequencing the Paths

At this point, we have a set of paths, each representing a pushing movement for moving one object to its destination. To integrate them into one path, we have to find a total ordering among them. This is the task of the *Sequencing* step. (Note that if there are several types of objects, the assignments will be done separately for each type, but the sequencing step will combine all of them.)

There are two considerations in sequencing the paths. First, we have to put precedence constraints between paths that might interfere with each other. Second, we try to minimize the movement of the robot between the pushing actions, to obtain results closer to optimal.

We create a new graph, a *directed graph* called the *PrecedenceGraph,* which is defined as follows*:*

- Each *node* of the *PrecedenceGraph* represents one pushing action, i.e. one *path* found by the *Assignment* function.
- Each *directed edge* of the *PrecedenceGraph* represents a precedence constraint between the pushing actions.

An example of a situation that requires a precedence constraint is shown in figure 3.

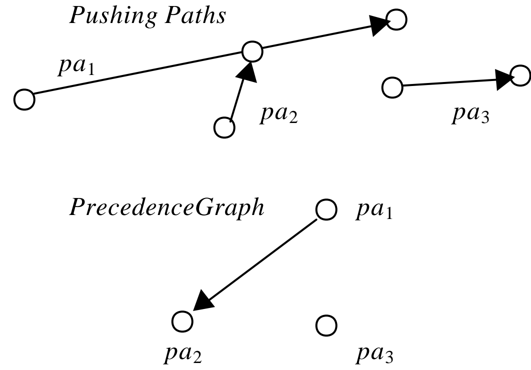

Figure 3. An Example of a Precedence Constraint.

The top of figure 3 shows three pushing actions. Pushing action pa_2 might interfere with the motion of pa_1 if pa_2 is done first. This possible interference can be avoided by requiring that pa_1 precede pa_2. The *PrecedenceGraph* for this problem is shown at the bottom of figure 3. Edges show required precedence relations between actions.

Having constructed the *PrecedenceGraph*, a total (linear) ordering of the actions can be found by a *Topological Sort* algorithm [Cormen *et al*. 1990]. The idea is to repeatedly find a node of in-degree 0 and select it as the next one in the total order.

To reduce the intermediate movements between pushing actions, we can apply heuristics in the topological sort procedure. When there are alternatives in choosing nodes with in-degree 0, we choose that which minimizes the cost of the intermediate robot movement from the last chosen pushing action.

Checking precedence constraints takes $O(d^2t^2)$ where d is the number of destinations and t is the number of intermediate points created in *FindPaths.* The d^2 reflects the fact that in the worst case we have to check each pair of pushing actions, and the number of pushing actions is equal to the number of destinations. The factor t^2 comes in because each path might consist of several intermediate points. However, the bound is not tight. In practice we found that d^2 strongly dominates t^2 since most of the paths are direct paths or paths with only a few intermediate points.

Topological sorting takes $O(d + r)$ where r is the number of precedence constraints. If we use heuristics, at each step we need to compare nodes with in-degree 0, in which case the time complexity becomes $O(d^2)$.

The optimality of the *Sequencing* function is not guaranteed since it is an instance of the Traveling Salesperson Problem, which is NP-complete. Furthermore, due to the precedence constraints, the graph is not complete, and the cost function fails to satisfy the triangle inequality, a condition that is usually assumed in developing good algorithms. In general, if

the triangle inequality assumption is dropped, good approximate tours cannot be guaranteed [Cormen *et al*. 1990]. However, our algorithm tries to reduce the execution cost by using the heuristic described above, which in effect performs all the motions in a neighborhood before moving to the next one.

Apart from optimality, there is an important question of whether the sequencing guarantees completeness. This is not the case primarily due to the assumption that in the total ordering of the paths each pushing movement taking an object to its destination is not interrupted by pushing movements for other objects. It is possible to construct examples where the movement of an object should be broken into more than one separate segments. First, we should move the object to a temporary location to give way for a second object, followed by pushing the second object, and finally concluded with pushing the first object from its temporary location to its destination. We found out that situations that need this kind of maneuvering rarely arise in the kind of nanoparticle assembly tasks we are considering. An algorithm that handles this situation is discussed in [Ben-Shahar & Rivlin, 1998].

Figure 1d shows the final result of sequencing on the situation in figure 1c.

7. Results and Discussion

The planner (apart from sensing and motor control) is written in Java and executes on a Sun Ultra-10 workstation. We have tested it on several problems with 50 objects. Running times for 50 objects and 10 obstacles are in the range of 5 – 10 seconds.

Each subtask is accomplished in low-order polynomial time. Furthermore, experimental results show that many of the complexity bounds are not tight. In practice, the influence of the e factor could be greatly reduced, and in the precedence detection the t^2 factor is usually a small number.

The optimality of the pushing paths (i.e. paths where the robot is pushing an object) is guaranteed, save for the fact that we give priority to direct paths over paths that involve turns around obstacles.

Optimality is not guaranteed for the motion of the robot between pushing actions. However, this subtask is known to be intractable. All optimal solutions for that problem are exponential and, to the best of our knowledge, no practical algorithm has produced results within a constant ratio of the optimal ones.

The completeness is not guaranteed in cases where the pushing movement for one object has to be broken into more than one separate segments, interrupted by the pushing movement for another object. However, this situation rarely arises in the tasks we are working on.

In summary, we have demonstrated a practical comprehensive planner for constructing 2-D nanoparticle patterns. It is efficient and produces good plans. This is a first step towards automating nanoparticle assembly tasks in nanorobotics.

References

[Barraquand & Latombe 1990] J. Barraquand and J.-C. Latombe, “A Monte-Carlo algorithm for path planning with many degrees of freedom”, *IEEE Int. Conf. Robot. Automat.*, 1990, pp. 1712 – 1717.

[Barraquand & Latombe 1991] J. Barraquand and J.-C. Latombe, “Robot motion planning: a distributed representation approach”, *Int. J. Robot. Res.*, Vol. 6, No. 10, pp. 628 – 649, December 1991.

[Ben-Shahar & Rivlin 1998] O. Ben-Shahar and E. Rivlin, “Practical pushing planning for rearrangement tasks”, *IEEE Trans. Robotics and Automation*, Vol. 14, No. 4, pp. 549 – 565, August 1998.

[Chen & Hwang 1991] P. C. Chen and Y. K. Hwang, “Practical path planning among movable obstacles”, *IEEE Int. Conf. Robot. Automat.*, 1991, pp. 444 – 449.

[Cormen *et al*. 1990] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: The MIT Press, 1990.

[Kavraki *et al*. 1996] L. Kavraki, P. Švesta, J.-C. Latombe, and M. H. Overmars, “Probabilistic roadmaps for path planning in high dimensional configuration spaces”, *IEEE Trans. Robotics and Automation*, December 1996, pp. 566 – 580.

[Knuth 1993] D. E. Knuth, *The Stanford GraphBase*. New York, NY: The ACM Press, 1993.

[Latombe 1991] J-C. Latombe, *Robot Motion Planning*. Norwell, MA: Kluwer, 1991.

[Requicha, 1999] A. A. G. Requicha, “Nanorobotics”, in S. Nof, Ed., *Handbook of Industrial Robotics*. New York, NY: Wiley, 2nd ed., pp. 199 – 210, 1999.

[Requicha *et al.* 1998] A. A. G. Requicha, C. Baur, A. Bugacov, B. C. Gazen, B. Koel, A. Madhukar, T. R. Ramachandran, R. Resch, and P. Will, “Nanorobotic assembly of two-dimensional patterns”, *Proc. IEEE Int. Conf. Robot. Automat.*, Leuven, Belgium, May 16 – 21, 1998.

[Requicha *et al.* 1999] A. A. G. Requicha, R. Resch, N. Montoya, B. E. Koel, A. Madhukar, and P. Will, “Towards hierarchical nanoassembly”, *Proc. Int’l Conf. on Intelligent Robots & Systems* (*IROS ‘99*), Kyongju, S. Korea, pp. 889-893, October 17-21, 1999.

[Wilfong, 1988] G. T. Wilfong, “Motion planning in the presence of movable obstacles”, *Proc. ACM Symp. Computational Geometry*, 1988, pp. 279-288.