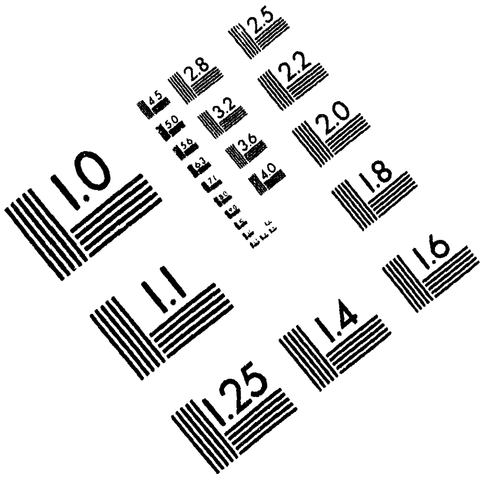
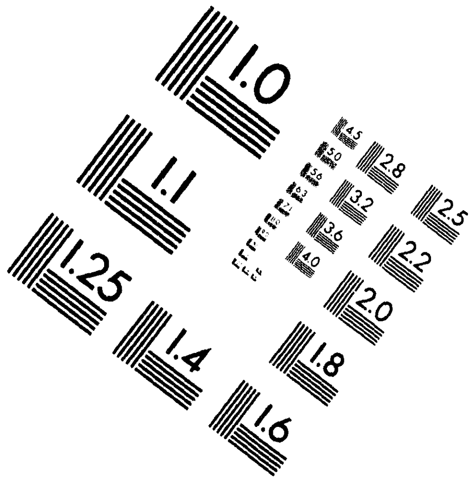
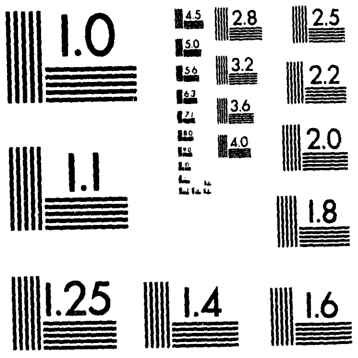



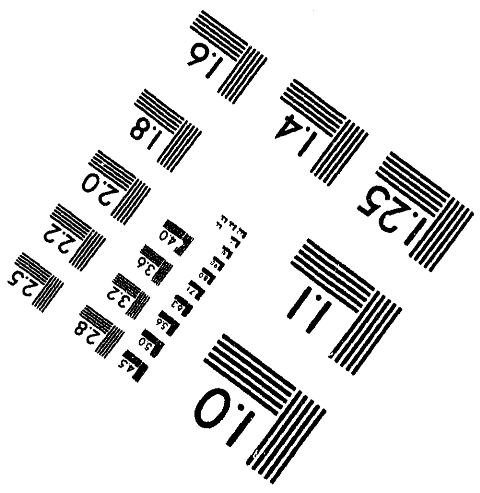
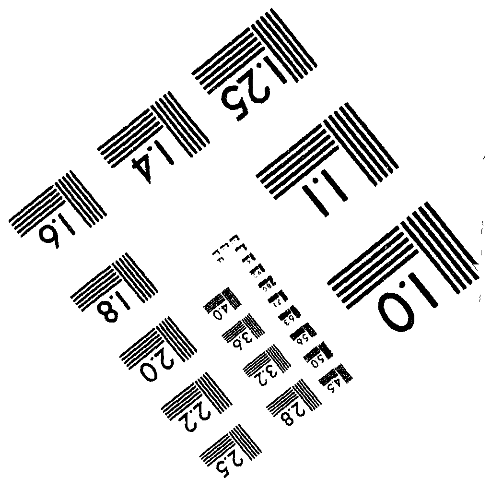
AIIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910

301/587-8202



1 of 1

12

CONF-9410167--1

Proceedings of the 1994 International Conference on Computer Integrated Manufacturing and Automation Technology.

SAND94-2005C

Adaptive Path Planning for Flexible Manufacturing*

Pang C. Chen
Sandia National Laboratories
Albuquerque, NM 87185-0951

Abstract

Path planning needs to be fast to facilitate real-time robot programming. Unfortunately, current planning techniques are still too slow to be effective, as they often require several minutes, if not hours of computation. To overcome this difficulty, we present an adaptive algorithm that uses past experience to speed up future performance. It is a learning algorithm suitable for automating flexible manufacturing in incrementally-changing environments. The algorithm allows the robot to adapt to its environment by having two experience manipulation schemes: For minor environmental change, we use an object-attached experience abstraction scheme to increase the flexibility of the learned experience; for major environmental change, we use an on-demand experience repair scheme to retain those experiences that remain valid and useful. Using this algorithm, we can effectively reduce the overall robot planning time by re-using the computation result for one task to plan a path for another.

1 Introduction

One of the most important enabling technologies of flexible manufacturing is path planning, which refers to finding a short, collision-free path from an initial robot configuration to a goal configuration. It has to be fast (ideally within seconds) to support real-time task-level robot programming. Accordingly, a large amount of research has been done on path planning [9, 11, 1, 2, 12], mostly for stationary environments. There is also some work on planning for mobile robots in time-varying environments that contain constantly moving obstacles [6, 10, 15]. All of these planners, however, typically require minutes of computation for mobile robots, and tens of minutes for 6 degrees of freedom manipulators. Further, little work has been done for changing environments in which movable obstacles remain relatively stationary during sequences of tasks, as opposed to time-varying environments with constantly moving obstacles.

To address the need of flexible manufacturing, we present a path planner for incrementally changing environments. Robots often perform multiple tasks in the same or a slowly changing environment, and in such cases planning time can be greatly reduced by reusing the computation results for one task to plan for another. One target application is manufacturing of evolving products in which the design changes made to a product are relatively small. In this case, the assembly motion for the product before change can often be reused with little modification for the new product.

We assume that for each robot task, the obstacles are stationary, but may slowly change their configuration or shape over the course of the robot performing many tasks. We present a learning algorithm that 'adapts' to the environment change. There are a few path planners that incorporate learning [3, 8, 13]; however, none deals specifically with changing environments. Our algorithm extends the work of [3], which as stated, can only be applied to stationary environments.

In the following section, we first briefly describe the work in [3] on stationary environments, and then present the new algorithm for changing environments. The algorithm is composed of two experience-manipulating schemes designed to cope with minor and major environmental change. In addition to presenting the algorithm, we also identify three other variant strategies for using old experiences in new environments. We illustrate the algorithm and its variants with an example in Section 3, and demonstrate in Section 4 the effectiveness of the proposed algorithm.

2 Algorithm

Let task (u, w) be defined as finding a collision-free path to move the robot from configuration point u to w. We assume that there are initially two path

*Supported by US DOE contract DE-AC04-94AL85000.

MASTER

gtB

✓

planners available: Reach and Solve. The Reach planner is required to be fast, symmetric, and only locally effective, i.e., it should have a good chance of success if u and w are close to each other. Any greedy hill-climbing method using a potential field [1] or sliding [7] approach should be sufficient to implement Reach. The Solve planner, on the other hand, is required to be much more globally effective than Reach, and hence is very slow. The planner may even be the human operator himself. It is the performance of this planner that we wish to improve.

In our learning scheme, we retain the global effectiveness of Solve by calling it whenever necessary, while reducing the overall time cost by calling Reach whenever possible. To utilize Reach, we maintain a digested history of robot movements in the form of a connected graph, called the *experience graph* $G = (V, E)$ with vertices V and edges E. Set V is a sparse collection of subgoals that the robot can attain and use. Set E indicates the subgoal connections that the robot can follow through the application of Reach. Ideally, G is to be used by Reach to achieve most tasks without the help of Solve. If Reach is incapable of achieving a task through G, Solve is called. If Solve is also incapable of finding a solution, then we simply skip to the next task. Otherwise, we learn from the solution of Solve by abstracting (or compressing) it into a chain consisting of a short sequence of subgoals that Reach can use later to achieve the same or similar tasks.

2.1 Environmental Assumptions

To allow fruitful learning, we assume that the environmental change is incremental, i.e., occasional and localized. By occasional, we mean that the interval between workcell changes is large compared to the amount of time spent on each task. By localized, we mean that the workcell change involves only a few objects in a relative small area of the workspace. Both conditions are prevalent in applications and have their intuitive implications: Occasional implies that old experience may be useful for significant amount of time, and localized implies that old experience may have salvage value.

2.2 Formal Specification

Formally, the speedup learning algorithm Adapt is shown in Figure 1. It is the same as the one for stationary environments [3] except for the extra boxed fragments. The second boxed fragment introduces $\mathcal{T}$, the

```
Algorithm Adapt($\mathcal{R}, \mathcal{S}; \mathcal{T}$)
  $u \leftarrow$ current position; $v \leftarrow u$; $G \leftarrow (\{v\}, \emptyset)$;
  do forever
    $\boxed{\text{Repair(G);}}$
    $w \leftarrow$ goal();
    if (not $\mathcal{R}(u, w; G, h)$) then
      if (not $\mathcal{S}(u, w; G, h)$) then continue;
      $\rho \leftarrow$ Abstract($\mathcal{S}[u, w; G, h]$);
      $G \leftarrow$ Augment($G, \rho$);
      $\hat{w} \leftarrow$ last vertex of $\rho$;
    endif
    $\boxed{\text{if } \mathcal{T}(u, w; G, h) \text{ then}}$
      Execute($\mathcal{R}[u, w; G, h]$); $u \leftarrow w$;
  enddo
end.
```

Figure 1: A learning algorithm for path planning in incrementally-changing environment

trace procedure that verifies and repairs old experience on demand. The first fragment, which introduces Repair, is not part of the algorithm, but is included for later discussion (Section 2.5) of other variants of the algorithm that use it.

In the algorithm, u is the current robot configuration, and w is the next goal configuration. To access G, we maintain two pointers: $\hat{u}$ and $\hat{w}$, each of which points to a vertex of G that is known to be reachable with one call of Reach from u and w, respectively. The algorithm is based on two planners: $\mathcal{R}$ and $\mathcal{S}$, which are in turn based on Reach and Solve, respectively. Both $\mathcal{R}$ and $\mathcal{S}$ have task (u, w) as arguments, and graph G and a heuristic vertex ordering function h as parameters. For planner $\mathcal{R}$, we use $\mathcal{R}(\cdot)$ to denote the predicate that $\mathcal{R}$ is successful, and $\mathcal{R}[\cdot]$ to denote the path planned when $\mathcal{R}$ succeeds, and similarly for $\mathcal{S}$.

Planner $\mathcal{R}$ searches for ways to achieve task (u, w) using only Reach and G as guideline. The algorithm for $\mathcal{R}(\cdot)$ is the same as for the stationary case: Search the vertices of G in order according to heuristic h, and find a vertex v satisfying Reach(v, w). If v exists, then set $\hat{w} \leftarrow v$, and return success; else return failure.

However, to generate $\mathcal{R}[\cdot]$ for changing environments, we require the success of $\mathcal{T}(\cdot)$, which guarantees that there is a connected sequence of vertices Γ in G from $\Gamma_1 = \hat{u}$ to $\Gamma_k = \hat{w}$ for some $k \geq 1$. Once $\mathcal{T}(\cdot)$ succeeds, a simple solution for $\mathcal{R}[\cdot]$ would be the concatenation of Reach$[\Gamma_j, \Gamma_{j+1}]$ for j going from 0

✓

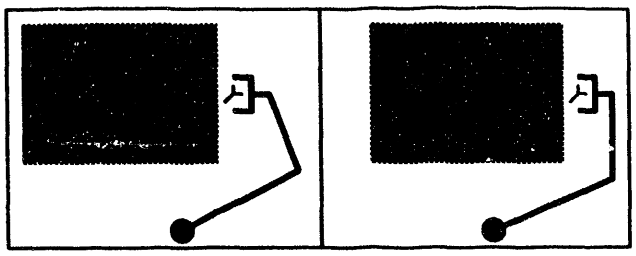


Figure 2: Object-attached experience using critical tag-points.

to k. Incidentally, the quality of this solution can be improved locally as we shall see in Section 2.6.

Planner $\mathcal{S}$ sets $\hat{w}$ for the future augmentation of G, and is the same as for the stationary case: To evaluate $\mathcal{S}(\cdot)$, set $\hat{w}$ to be the best vertex in G according to h, and return(Solve($\hat{w}, w$)). To generate $\mathcal{S}[\cdot]$ once $\mathcal{S}(\cdot)$ returns success, simply output Solve[$\hat{w}, w$].

2.3 Object-attached Experience Abstraction

To abstract a solution path from v to w with $v \in G$, we again assume as in [3] that there is an efficient Abstract($\cdot$) function available that returns a short chain of critical vertices from v to $v' = w$, with each segment traversable by Reach. We assume that the size of the chains abstracted from solutions of $\mathcal{S}$ are all boundable by a constant. In practice, this is a reasonable assumption, since a typical task consists of only 3 smooth motions: departure, traversal, and approach. The process of compressing a solution path into a few subgoals can be implemented in many ways: One simple method is by means of binary search on the appropriately discretized solution path.

Now, to increase the flexibility of the subgoals, we require the vertices returned by Abstract($\cdot$) to be relative robot positions associated with nearby objects, rather than the absolute positions in the stationary case. That is, instead of remembering the robot positions as some points in absolute space, we now remember each of them as an offset from some nearby object serving as a landmark.

One way to implement this strategy is to create a tag-point (pose for the robot tool point) for each critical robot position, and affix the tag-point to the local coordinate of a nearby object. Then, as this nearby object changes its location or orientation, the tag-point can be adjusted accordingly so that the robot tool point can maintain its distance to the object under change. Figure 2 shows such an example. In the left frame, the robot position is recorded via the tag-point of the robot tool, and is attached to the rectangular object. As the object moves toward the right, the tag-point moves along also, enabling the robot to comply with the change. If the tag-point had not been attached to the object, the corresponding robot position would have become invalid in the new environment.

Two potential drawbacks of this tag-point method are that solving the inverse-kinematics for the tag-point will be necessary to recompute the robot configuration for the subgoal, and that solutions may disappear for tag-points whose attached objects have moved too much. Nevertheless, under this object-attached experience abstraction scheme, we can adjust to any minor environmental change without expensive experience repair.

2.4 On-Demand Experience Repair

Of course, if the environment changes significantly, the validity of G will deteriorate. How much deterioration G will suffer depends on how drastically the environment changes. If the change is major and extensive, then it may be better to start over with no experience (G reinitialized), rather than to work with the old impaired experience. In the more interesting case where the change may be major (e.g., introducing a new object) but not extensive (e.g., the rest of the workcell is undisturbed), the right choice is not as clear. Therefore, we introduce an on-demand repair scheme (second boxed fragment in Figure 1) to retain those experiences that remain valid and useful.

In this scheme, we plan as if G is connected, until $\mathcal{R}(\cdot)$ succeeds and we actually need to produce a path. Then, to generate $\mathcal{R}[\cdot]$, we require the success of $\mathcal{T}(\cdot)$ to provide a connected sequence from $\hat{u}$ to $\hat{w}$. As $\mathcal{T}(\cdot)$ searches for and verifies such a sequence, it may come across invalid edges, which it simply deletes. If $\hat{u}$ is already connected to $\hat{w}$ in G, then no repair need take place. If, however, $\hat{u}$ and $\hat{w}$ do not belong to the same (connected) component due to the deterioration of G, then Solve is called to reestablish their connectivity. It is of course possible that connectivity cannot be reestablished due to the environmental change. In this case, the portion of G connected to $\hat{w}$ is deemed useless, and hence discarded. The procedure for $\mathcal{T}(\cdot)$ is as follows:

```
1. While there exists a sequence $\Gamma$ of vertices in $G$ connecting $\hat{u} = \Gamma_1$ to $\hat{w} = \Gamma_k$ for some $k \geq 1$ do
    (a) If Reach($\Gamma_i, \Gamma_{i+1}$) for all $1 \leq i < k$ then return success;
```

(b) Else remove edge (Γ_i, Γ_{i+1}) with smallest i such that $\neg$Reach(Γ_i, Γ_{i+1}).

2. If Solve$(\hat{u}, \hat{w})$ succeeds then augment G with Abstract(Solve$[\hat{u}, \hat{w}]$); return success;
3. Else remove the (connected) component of $\hat{w}$ from G, and return failure.

2.5 Other Repair Strategies

It is also possible to cope with major environmental change using other variants of the on-demand repairing strategy. One trivial strategy is simply to forget the old experience and start over (with G reinitialized) whenever there is a change in the environment. The corresponding algorithm, $\mathcal{A}_0$, can be obtained from Figure 1 by skipping the boxed condition, and defining Repair(G) to be the reinitialization procedure.

Another less trivial strategy is to verify each edge of G first whenever there is a change. Then with the time investment, we can initialize G to the home component that contains the current robot position. The corresponding algorithm, $\mathcal{A}_1$, can again be obtained from Figure 1 by skipping the boxed condition, and defining Repair(G) to be the above home-component extraction procedure.

Notice that both strategies above only update G according to environmental change, and do not really repair old experience. In contrast, a third strategy that repairs actively is to first apply $\mathcal{T}$ to attempt reaching every vertex of G from home, before taking on any new task. The corresponding algorithm, $\mathcal{A}_2$, can be obtained from Figure 1 by skipping the boxed condition, and defining Repair(G) to be the above repair-all procedure.

All of the suggested algorithms (including the repair-on-demand algorithm $\mathcal{A}_3$) have their advantages and disadvantages. Intuitively, if the environment undergoes a major and extensive change, then starting over with $\mathcal{A}_0$ may be the best choice. On the other hand, if Solve costs much more than Reach, then using $\mathcal{A}_1$ to save some old experience may be better. Alternatively, if the change is only local, then repairing old experience with $\mathcal{A}_2$ or $\mathcal{A}_3$ may be more beneficial. Which algorithm to use thus depends on the particular application.

2.6 Solution Quality and Redundancy

So far we have focused on task solvability but not solution quality. If solution quality is not important, then in $\mathcal{R}[\cdot]$, we can simply produce the solution of going through Γ with Reach. In this situation, the experience graph will always be a tree. However, if solution quality is important, then it may be worthwhile to locally optimize Γ by seeking to "cut corners" whenever possible. The result of this compression is that G may be augmented with additional edges to enable shorter sequences in the future. Also, the redundancy introduced may be useful in combating against experience deterioration.

3 Example

We illustrate the learning algorithm with a simple example involving a point robot in a 2D workspace. (Similarly, the algorithm can plan for arbitrarily shaped and jointed robots by planning for a point robot in the configuration space.) Let Reach implement a go-straight procedure, with Reach(u, w) returning success iff w is visible from u, and Reach$[u, w]$ returning the line segment $\overline{uw}$. Let Solve implement a greedy 2-step go-straight procedure, with Solve(u, w) returning success iff the two points are connectable by at most 2 line segments, and Solve$[u, w]$ returning the shortest such connecting path. To complete the algorithmic specification, let the heuristic used in $\mathcal{R}$ and $\mathcal{S}$ be $h = h_1$, with h_1 ordering the vertices of G according to the distance to w, starting with the closest point first.

Figure 3 illustrates Adapt with a series of snapshots. Frame (1) shows the initial setting with the robot at home $u = w_0$ amongst two objects A and B. The robot's initial tasks are to inspect both A from w_1 and w_2, and B from w_3 and w_4. To begin, the experience graph G is initialized to the single vertex $v_0 = w_0$.

The first goal indicated by w_1 is shown in Frame (2). Since $\mathcal{R}$ is unable to plan using only Reach and G, Adapt then calls $\mathcal{S}$. Using h, $\mathcal{S}$ chooses to extend from v_0 to w_1, since v_0 is the only vertex in G. The path produced by Solve(v_0, w_1) consists of the line segments $\overline{v_0v_1}$ and $\overline{v_1v_2}$. This path is then abstracted into the chain connecting v_0 to v_1 and v_1 to v_2. The result of augmenting G is that G now becomes the 3-vertex chain. Using this augmented G, $\mathcal{R}$ is now able to produce a path from $u = w_0$ to w_1, which consists of the segments $\overline{uv_0}$, $\overline{v_0v_1}$, $\overline{v_1v_2}$, and $\overline{v_2w_1}$, with $\overline{uv_0}$ and $\overline{v_2w_1}$ being null segments.

With the first task accomplished, the next task is to go to w_2 shown in Frame (3). Since $\mathcal{R}$ is again unable to plan using only Reach and G, Adapt then calls $\mathcal{S}$. Using h, $\mathcal{S}$ chooses to extend from v_2 to w_2, and produces the line segments $\overline{v_2v_3}$ and $\overline{v_3v_4}$. The result of augmenting G is that G now becomes the 5-vertex chain with new vertices v_3 and v_4.

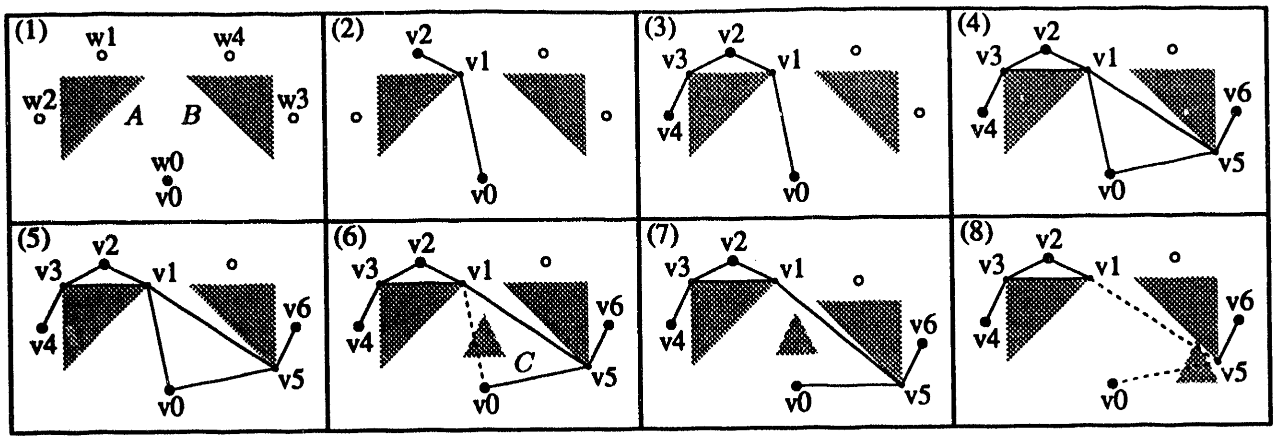


Figure 3: Snapshots of Adapt under environmental change

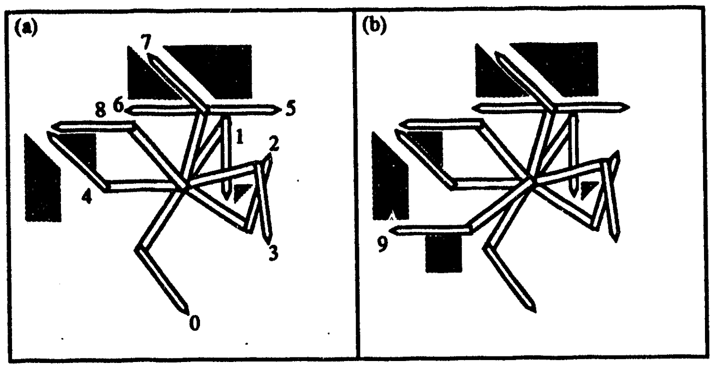


Figure 4: A planar 2-link robot environment with incremental change

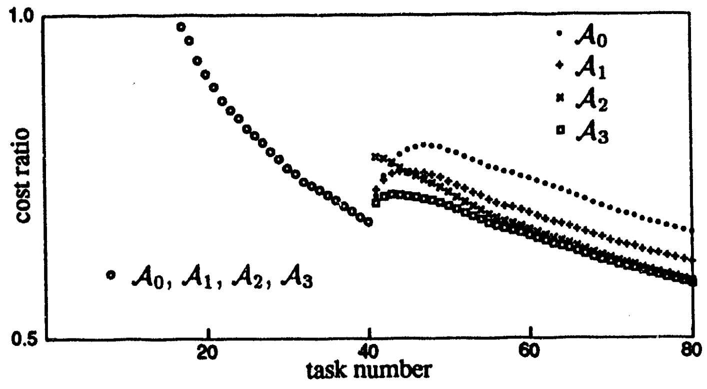


Figure 5: Time improvement of Adapt with all 4 repair strategies over Solve

With this G, $\mathcal{R}$ is still unable to succeed in reaching w_3 in Frame (4). Consequently, $\mathcal{S}$ chooses to extend from v_0 and produces 2 more segments $\overline{v_0v_5}$ and $\overline{v_5v_6}$. Thus, before calling $\mathcal{R}[\cdot]$, G is a 7-vertex chain with new vertices v_5 and v_6. After calling $\mathcal{R}[\cdot]$, however, G becomes cyclic due to the addition of edges (v_3, v_1) and (v_1, v_5) as a result of locally optimizing the solution path $(v_4, v_3, v_2, v_1, v_0, v_5, v_6)$.

Frame (5) shows that $\mathcal{R}$ is now capable of reaching w_4, with $\hat{w}_4 = v_1$. Consequently, $\mathcal{S}$ is not called for the first time, and G is not modified.

So far, the workcell has been stationary. In Frame (6), we return the robot to its home and introduce a new object C. With $\mathcal{A}_0$ using the start-over strategy, we would lose the entire G and not retain anything from the 3 previous calls to $\mathcal{S}$. With $\mathcal{A}_1$, we would verify all 8 edges in G with Reach, remove the only broken edge (v_0, v_1), and retain the rest of G since it remains connected. If Reach costs much less than Solve, then the return on the initial time investment is certainly justifiable compared to that of $\mathcal{A}_0$. This case demonstrates that improving solution quality can also increase experience redundancy, which in turn decreases experience deterioration under change. With $\mathcal{A}_2$ using the active-repair scheme, we would also just remove edge (v_0, v_1) from G at the end of Repair(G). With $\mathcal{A}_3$ using the repair-on-demand strategy, we simply do nothing.

Frame (7) shows what happens if we introduce some minor change by moving object B and its object-attached goals w_3 and w_4. Because of the object-attached abstraction scheme, v_5 and v_6 also move along with B. Consequently, if the robot were to go back to w_3, it would again succeed by simply reaching toward v_5 and v_6.

Frame (8) shows what happens if we move object C to a corner and decide not to inspect object B anymore. In this case, $\mathcal{A}_1$ would be identical to $\mathcal{A}_0$ in reducing G back to the single vertex v_0, except that $\mathcal{A}_1$ would also have to spend time verifying all 7 edges of G before removing them. With $\mathcal{A}_2$, G would be actively repaired, which means that it would call Solve twice to reestablish the connectivity of the 2 components to v_0. With $\mathcal{A}_3$, we again do nothing until the need arises. If we choose not to inspect B anymore, then only one component needs to be reconnected to v_0, which means only one additional call to Solve would be required in the future. This case demonstrates the situation where using $\mathcal{A}_3$ is better than using $\mathcal{A}_2$.

4 Computational Experience

Using Adapt, we have improved the performance of the same path planner used in [3], this time operating under environmental change. Figure 4 shows a 2-link planar robot environment in which Adapt is applied. The environment exemplifies the planar component of a typical robot workcell in a SCARA configuration [5] with the z-component decoupled. In this experiment, the initial environment shown in Frame (a) has 5 polygonal obstacles in the workcell and a goal set consisting of 9 preselected goal positions. Starting at home 0, the robot is to go through a sequence of goals randomly selected from the goal set. During the exercise, we introduce an incremental environmental change, shown in Frame (b), by adding a new obstacle to the workcell and a new goal position to the goal set.

The result of this experiment, with Adapt using all 4 different repairing strategies, is shown in Figure 5. Here, the ratio of the cumulative planning cost required by Adapt to that required by Solve only is plotted against the task number. The planning costs are averaged over 100 runs and are measured by the number of robot-to-obstacle distance evaluations, which is the dominating factor in the computing cost of each planner. The environment change is introduced after task 40. To emphasize the important features of the result, the initial portion of the curve corresponding to ratios greater than 1 is not plotted. The unplotted portion actually decreases monotonically from 2.5 at task number 1 to 1.0 at task number 16. The experiment shows that before the environmental change, Adapt is able to learn and speed up its performance relative to Solve from 150% slower to 33% faster. It also shows that Adapt needs about 16 training tasks before becoming competitive with Solve, a fact attributable to both the task simplicity for Solve and the significant costs incurred by Adapt during solution abstraction and compression.

After the environmental change, the performance curve for Adapt splits up into 4 curves, each corresponding to a different experience repairing strategy. The curves for $\mathcal{A}_0$, $\mathcal{A}_1$, and $\mathcal{A}_3$ exhibit similar behaviors in that they all gradually increase and then decrease at roughly the same rate, with $\mathcal{A}_3$ being clearly better than $\mathcal{A}_1$, which in turn being clearly better than $\mathcal{A}_0$. The curve for $\mathcal{A}_2$ is different in that it first jumps to a high point and then comes down rapidly to approach the curve for $\mathcal{A}_3$. The jump is due to the high initial cost of active repair, and the rapid decrease is due to the benefit of the repair. Overall, the relative performance of the repairing strategy is as expected, since the environmental change is incremental, involving only local and occasional change. In fact, one can formalize the concept of local and occasional change, and prove the optimality of on-demand repair $\mathcal{A}_3$ relative to the other variants $\mathcal{A}_0$, $\mathcal{A}_1$, and $\mathcal{A}_2$ under such change [4].

5 Conclusion

We have presented an adaptive path planning algorithm for flexible manufacturing in incrementally-changing environments. The algorithm extends a previous work for stationary environments with two augmenting experience-manipulating schemes: For minor environmental change, an object-attached experience abstraction scheme is introduced to increase the flexibility of the learned experience; for major environmental change, an on-demand experience repair scheme is introduced to retain those experiences that remain valid and useful.

We have discussed the tag-point approach to storing the object-attached experience. In justifying our on-demand experience repair scheme ($\mathcal{A}_3$), we have also identified three other variants with different repairing strategies: $\mathcal{A}_0$ simply forgets the old experience and starts over whenever there is a change; $\mathcal{A}_1$ first verifies the old experience and then retains only the home component; and $\mathcal{A}_2$ actively repairs the old experience before taking on new tasks. We have discussed the relative merits of each repair scheme and characterized their performance curves. Finally, we have demonstrated the practicality of our algorithm by improving the performance of an existing path planner under a changing environment.

References

[1] Barraquand, J., Latombe, J., "A Monte-Carlo algorithm for path planning with many degrees of freedom," *Proc. of IEEE Int. Conf. on Robotics and Automation*, pp. 1712-1717, 1990.

[2] Chen, P., Hwang, Y., "SANDROS: A Motion Planner with Performance Proportional to Task Difficulty," *Proc. of IEEE Int. Conf. on Robotics and Automation*, pp. 2346-2353, 1992.

[3] Chen, P., "Improving Path Planning with Learning," *Machine Learning: Proc. of the Ninth Int. Conf.*, pp. 55-61, 1992.

[4] Chen, P., "Adaptive Path Planning in Changing Environments," Sandia Report SAND92-2744, 1993.

[5] Craig, J., *Introduction to Robotics: Mechanics and Control*, pp. 268-269, 1989.

[6] Fujimura, K., Samet, H., "Time Minimal Paths among Moving Obstacles," *Proc. of IEEE Int. Conf. on Robotics and Automation*, pp. 1110-1115, 1989.

[7] Glavina, B., "Solving Findpath by Combination of Goal-Directed and Randomized Search," *Proc. of IEEE Int. Conf. on Robotics and Automation*, pp. 1718-1723, 1990.

[8] Goel, A., Callantine, T, Donnelian, M., Vazquez, N., "An intergrated experience-based approach to navigational path planning for autonomous mobile robot," *Proc. of IEEE Int. Conf. on Robotics and Automation*, pp. 818-825, 1993.

[9] Hwang, Y., Ahuja, N., "Gross Motion Planning — A Survey," *ACM Computing Surveys*, v. 24, (3), pp. 219-291, 1992.

[10] Kant, K., Zucker, S.W., "Planning Collision-free Trajectories in Time-Varying Environments: A Two-level Hierarchy," *Proc. of IEEE Int. Conf. on Robotics and Automation*, pp. 1644-1649, 1988.

[11] Latombe, J., *Robot Motion Planning*, Kluwer Academic Publishers, 1991.

[12] Lozano-Pérez, T., "A Simple Motion-Planning Algorithm for General Robot Manipulators," *IEEE J. of Robotics and Automation*, v. RA-3, 3, pp. 224-238, 1987.

[13] Pandya, S., Hutchinson, S., "A Case-based Approach to Robot Motion Planning," *Proc. of IEEE Int. Conf. on Systems Man and Cybernetics*, pp. 492-497, 1992.

[14] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[15] Reif, J., Sharir, M., "Motion Planning in the Presence of Moving Obstacles," *Proc. of 26th FOCS*, pp. 144-154, 1985.