16.32 Final Report: Dynamic Programming for Car Kinematics

Keenan Albee
Massachusetts Intsitute of Technology
77 Massachusetts Ave
Cambridge, MA 02139
917-531-4411
albee@mit.edu

***Abstract*—A dynamic programming scheme was implemented in Python to solve the optimal car motion problem. The Reeds-Shepp Car model was used over a state space $X = X_s \times Y_s \times \theta$, with inputs $U = V \times \phi$. A simple collision checking algorithm was incorporated into the dynamic programming solver to allow for the placement of arbitrary rectangular obstacles in the state space. The solver was run over multiple demonstrative scenarios, including parallel parking, navigation around an obstacle, and a 180 degree turn. The algorithm runs in $O(e^n)$, where n is the number of state space and input variables, in this case 5. Even taking advantage of a few heuristic tricks, this borders on the edge of feasibility due to the curse of dimensionality. Future work might involve more computationally efficient approximate solutions to the optimization problem, including RRTs.**

TABLE OF CONTENTS

1. BACKGROUND

Dynamic Programming and Hamilton-Jacobi-Bellman

Dynamic programming provides a numerical tool for approximating optimization problems. Specifically, dynamic programming is the discrete-time approximation of the Hamilton-Jacobi-Bellman equation

$$-J_t^* = \min_{u \in U} \mathcal{H}$$

and in the limit provides the exact solution for the optimal cost-to-go, J^*. By discretizing input actions and state, one effectively creates a mesh of state variables over which the combination of discretized state variables may be applied. [1] [2] [3] In essence, the HJB equation states that the minimization of the Hamiltonian is performed by choosing control actions that follow an optimal decreasing measure of remaining cost, the cost-to-go.

Algorithm

The dynamic programming implementation is based off of that given by Kirk [4], and is included in detail in Appendix A. Figure 1 shows the generic dynamic programming algorithm. There are a few main ingredients: a mesh of cost-to-go values, J^*, states X and inputs U discretized to be solved computationally, a state update function $f(x_k, u_k)$ to propagate the state, and a cost function $cost()$ to assign an additive weighting to actions. In addition, a practical application requires some additional tools: an interpolation scheme for finding the cost-to-go at non-mesh values and for evaluating stored control actions, and a method for dealing with mesh edge cases; that is, a way to interpolate on the outer edges of a mesh when the desired point is not fully contained in an 8-point mesh cell.

1: **procedure** DYNAMIC PROGRAMMING(X, U)
2: $J^* \leftarrow h(x(t_f))$ ▷ For admissible final position
3: **for** k iterations **do** ▷ k mesh iterations
4: **for** X states **do** ▷ Discretized states
5: **for** U inputs **do** ▷ Discretized inputs
6: $x_{k1} \leftarrow f(x_k, u_k)$
7: $J_{k1} \leftarrow interpolate(x_k, x_{k1})$
8: $C_{k1} \leftarrow cost(u_k, x_k, x_{k1})$
9: $J_{k1,tot} = J_{k1} + C_{k1}$
10: **end for**
11: $U^*_{x_k} \leftarrow \min_{u \in U} J_{k1,tot}$
12: **end for**
13: **end for**
14: return J^* ▷ Optimal cost-to-go for entire mesh
15: **end procedure**

Figure 1. Generic dynamic programming.

Dynamic programming can be applied to a wide array of problems, storing known solutions in the cost-to-go while exploring and updating the state space until convergence is reached. One interesting area is using dynamic programming in an exhaustive search of dynamics or kinematics states/inputs to determine an optimal trajectory. One recent example, for instance, is Schildbach's look at a nonholonomic car system navigating tight environments, though in his approach pains are taken to ensure that the maneuvering vehicle only takes control actions that end up back on points of the cost-to-go mesh. [5] This report will take a closer look at optimal nonholonomic car kinematics planning in the presence of obstacles.

2. METHOD

The Reeds-Shepp Car

The Reeds-Shepp car provides a simple model for car motion. It should be noted that this model effectively *ignores* the dynamics of car motion, instead making the assumption that the motion is slow enough that slippage cannot occur. This is a boon for dynamic programming, since the problem is no longer one of kinodynamic planning, but simply a kinematic

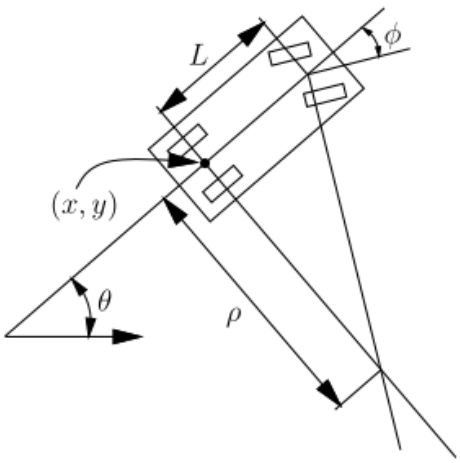

Figure 2. **The Reeds-Shepp simplified car model. [6]**

problem with motion obeying restrictions of geometry. The kinematic equations of motion are given by:

$$\dot{x} = vcos(\theta)$$

$$\dot{y} = vsin(\theta)$$

$$\dot{\theta} = \frac{v}{L}tan(\phi)$$

LaValle provides an explanation for the Reeds-Shepp car in [6]. In figure 2, ρ is the turning radius, and L is the distance from the front to rear axle. For simplicity, this is often assumed to be 1, as it is in this implementation. The Reeds-Shepp model also assumes that $v \in \{-v_{disc}, v_{disc}\}$, and that ϕ is assigned some maximum value ϕ_{max} which factors into the turning radius: a larger ϕ_{max} decreases ρ.

Collision Checking

Collision checking can easily be one of the most computationally expensive portions of a trajectory search algorithm. [6]. Fortunately for this problem, it was realized that since each mesh point does not modify its discretization of U, the resulting control actions at each mesh point will always be the same. This means that actions causing collisions can be stored in a precomputation, and these actions will always be skipped during the dynamic programming iterations.

Collision checking is practically a field of its own with regard to the potential complexity. For this problem, simple rectangular boundaries on x and y were chosen to represent obstacles. The algorithm for detecting a collision at any given point in X is shown in figure 3. This would typically be considered the 'broad phase' of collision checking, whereas the 'narrow phase' more carefully considers collision for nearby mesh points. In this problem, the precomputed collisions are stored in a supplemental matrix to note skips during dynamic programming.

Dynamic Programming

The simple car kinematic planning problem is 5 dimensional: $X = X_s \times Y_s \times \theta$, with inputs $U = V \times \phi$. The mesh must be formed from discretization of these variables. In this case, the mesh is the combination of a linear discretization of X, with J^*, v, and ϕ stored at each mesh point.

```
1:  procedure COLLISION CHECKING(X, U, obstacle, car)
2:      for X states do                          ▷ Discretized states
3:          for U inputs do                      ▷ Discretized inputs
4:              points ← car,from X_k
5:              for each in points do
6:                  if point < edge then
7:                      return False
8:                  end if
9:              end for
10:             return True                      ▷ If True, note this
11:         end for
12:     end for
13: end procedure
```

Figure 3. Simple Collision Checking

The $cost()$ function must also be determined. Two cost functions are used during implementation: one, a simple distance additive cost, weights the distance traveled over an action. The second adds an additional term that is activated if an action leads to a flip in the sign of v. This additional weighting terms helps to discourage the back and forth behavior that might be distance-efficient but in reality is a very poor method of moving a car (i.e. lateral to the tires).

$$cost() = dist(X_k, X_{k1})$$

$$cost() = dist(X_k, X_{k1}) + c * signflip(sign(X_k), sign(X_{k1}))$$

Interpolation

Figure 5 details the problem solved by interpolation. If a desired point does not lie on the mesh, then some scheme must be used to determine the approximate value of that point. This is used for finding the cost-to-go during the dynamic programming computation when an action drives the state off the mesh.

A second, distinct phase arises in dynamic programming. After the J^* mesh has been computed, U^* can be traced back to the goal from some starting value. This is trivial for control actions that always land on mesh points, but for nonholonomic car kinematics, staying on the mesh points is no guarantee. The tracing phase, then, requires an interpolation scheme to determine which control inputs to apply after arriving off of a mesh point. Interpolation logic is given in figure 4. This implementation skips interpolating if the entire cell is uninitialized, and performs simple averaging if the point is beyond an edge (but not marked as out-of-bounds during the precomputation). Finally, all other points (the majority) are interpolated using trilinear interpolation, which weights nearest points the most.

```
1:  procedure INTERPOLATE(X_k, X_k1)
2:      index ← FindNeighbors(X_k1)      ▷ Compute cell
3:      J_k1 ← FilterEdges(index)        ▷ Adjust edge indices
4:      if ∀J_k1 uninitialized then no update
5:      end if
6:      if J_k1 on edge then return mean(J_k1)
7:      end if
8:      return TriInterp(J_k1)
9:  end procedure
```

Figure 4. Interpolation Logic

Interpolation is used any time a value between grid cells is desired. In this implementation, interpolation is used for

J^*, v, and ϕ. (During the dynamic programming phase for J^* and during tracing otherwise).

There is also some decision-making to do with regard to how to use the interpolated value. For cost-to-go and ϕ this value can be used directly. For v, which is binary for the Reeds-Shepp model, one logical choice is to take the sign of the interpolated v and choose the binary of the same sign. For tracing U^*, this ensures that v is always the same magnitude.

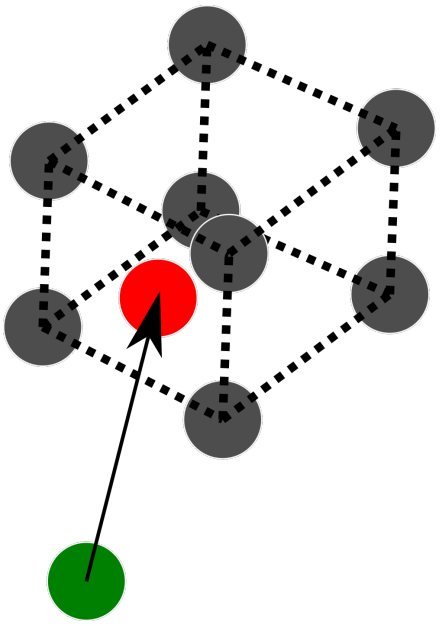

Figure 5. Propagating a point forward into a mesh cell. Red must be approximated by an interpolation scheme.

Implementation

Putting all the elements together, the overall dynamic programming approach can be summarized by figure 6.

procedure CK DYNAMIC PROGRAMMING(X, U)
  $J^* \leftarrow h(x(t_f))$ ▷ For admissible final position
  $cmesh \leftarrow CollisionChecking(X, U, obstacle, car)$
  $J^* \leftarrow \infty$
  for k iterations **do** ▷ k mesh iterations
    for X states **do** ▷ Discretized states
      for U inputs **do** ▷ Discretized inputs
        if collision U_k or X_k **then** skip
        end if
        $x_{k1} \leftarrow f(x_k, u_k)$
        $J_{k1} \leftarrow Interpolate(x_k, x_{k1})$
        $C_{k1} \leftarrow Cost(u_k, x_k, x_{k1})$
        $J_{k1,tot} = J_{k1} + C_{k1}$
      end for
      $U^*_{x_k} \leftarrow \min_{u \in U} J_{k1,tot}$
    end for
  end for
  return J^* ▷ Optimal cost-to-go for entire mesh
end procedure

Figure 6. Car Kinematics Dynamic Programming

A sample problem is posed in figure 7. A goal is specified (center green) on a mesh (black points), with obstacles (red) and a specified start position to trace (right green). Note that the mesh visible here is simply a cross section of θ; the state space is actually 3 dimensional.

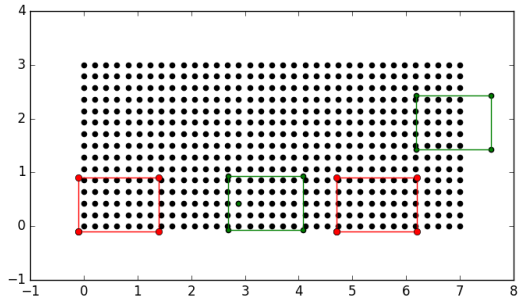

Figure 7. A sample car kinematics optimization problem.

Obstacles are found during the precomputation phase. In figure 8, red markers indicate inadmissible states after collision checking. The inadmissible area is larger than the obstacle boundary because of the car's geometry (this is similar to the concept of configuration space in robotics).

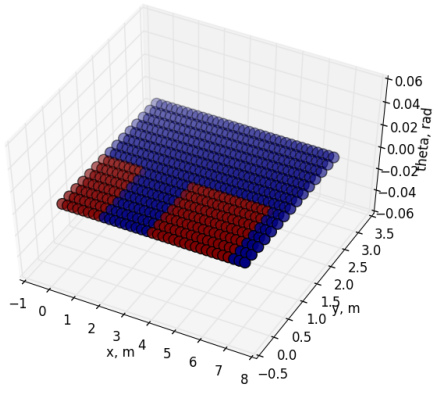

Figure 8. The $\theta = 0$ cross section, with admissible positions in blue.

Finally, the dynamic programming algorithm runs in full and after k iterations a mesh of J^* is prepared, shown in figure 9.

3. RESULTS

Sample Scenarios

Many scenarios were attempted during debugging and testing, but three in particular are mentioned here: parallel parking, the center box, and the 180 degree turn. These are intended to test the interpolation mechanism, collision checking, and cost function, as well as overall robustness of the implementation. The parallel parking scenario, shown in figure 10, successfully avoids obstacles and avoids lateral travel, only changing v twice, as expected. One notable feature was that the discretization of θ and ϕ leads to a slight 'wiggle' during straight line motion: a better discretization is always desirable if computation time allows.

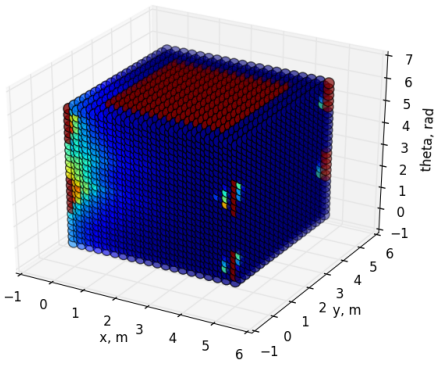

Figure 9. A state space mesh, with some outer edges not yet updated by iteration.

The parallel parking mesh reveals the narrow permissible area that the car must work with (blue). Further, the edge coloring shows some likely quirks in the edge interpolation, which requires extrapolating information from inner grid points to obtain an estimate.

The center box and 180 degree scenarios were also tested. Video results are provided in the accompanying presentation, along with video information from multiple other tests. The primary conclusions are that the collision checking algorithm functioned as expected, as evidenced by the parallel parking mesh. The cost function was adequate in minimizing distance traveled while avoiding excessive changes in v; tests with the primary cost function alone also had expected results. The center box tests showed successful navigation around an obstacle directly in the path of travel, while the 180 degree scenario showed the reach of the J^* propagation to the edge of the mesh to quickly reorient a backwards-facing car to the goal direction.

Computational Complexity

Dynamic programming is a classic example of an algorithm that suffers from the curse of dimensionality; its computational complexity is $O(e^n)$, where n is the sum of the number of states and control inputs. Within the same value of n, computational runtime varies linearly with the total number of mesh points, as shown from a few test runs.

Issues remaining include computational complexity and runtime performance in Python, requiring sparse meshes that produce less optimal results. For example, having a sparse discretization of U might result in a narrow control path being missed, resulting in a higher cost. Table 1 provides some mesh sizing and runtime information for the tests performed. As expected, the tests with more obstacles have more excluded points. Further, runtime scales roughly linearly with with the total number of mesh points, which is also expected based on the linear time complexity of the interpolation scheme and action update.

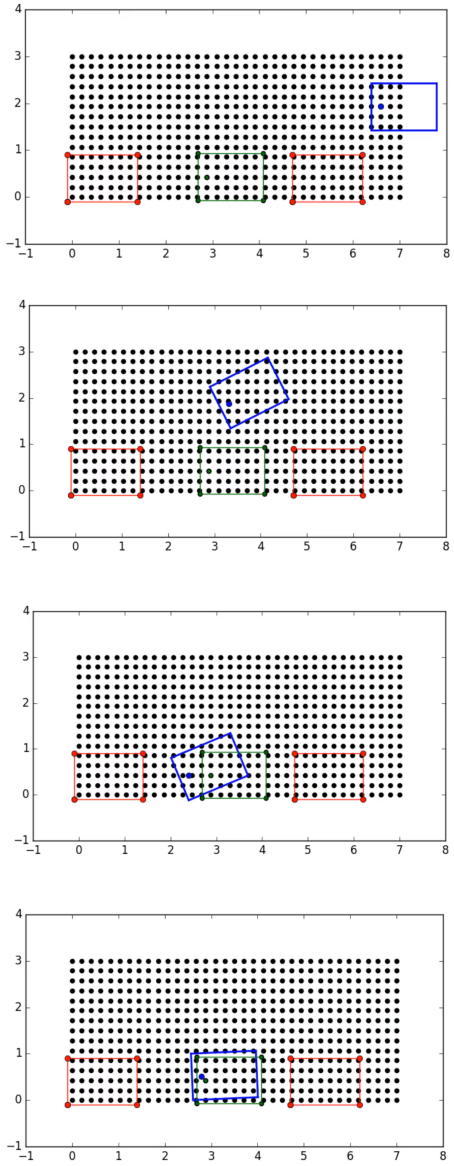

Figure 10. A successful parallel parking maneuver. Green is the goal state, and red are obstacles.

Test Type	Parallel Parking	Center Box	180 Degree
Grid Size	319200	810000	478800
Excluded Points	144749	278395	42218
Time to Solve (s)	1307.4	3097.1	3229.2

Table 1: Computation Time and Point Exclusion

4. CONCLUSIONS

Discussion

The results fulfilled one of the primary goals of this project: to demonstrate sensible parallel parking using dynamic programming. The meshing and U^* results of the project proved very satisfying. Beyond qualitative success, the runtimes of all tests to achieve reasonable performance (i.e. reasonable discretization) were rather high (on the order of an hour

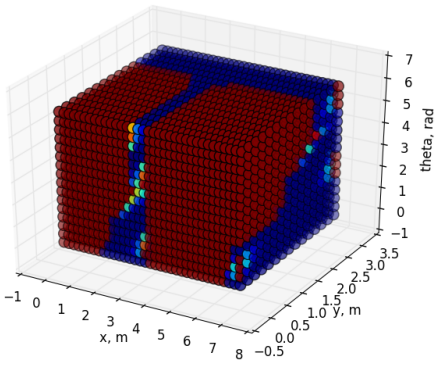

Figure 11. The mesh for parallel parking. Note blue areas in between obstacles.

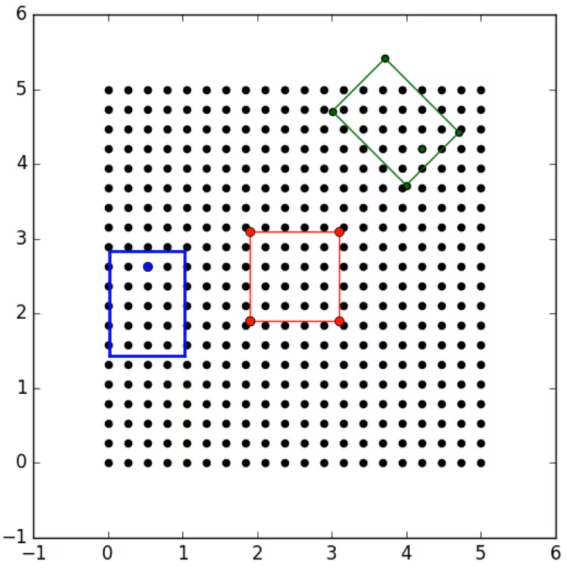

Figure 12. The center box scenario.

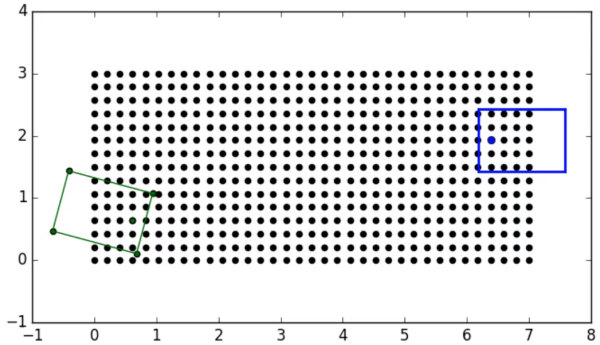

Figure 13. The 180 degree turn scenario.

for 500,000 size meshes), and assuredly not useful for online computation. It is possible that offline lookup tables for car maneuvers could be computed, but this is unlikely to be exceedingly helpful in cluttered environments. One promising area to explore in the future might be the use of randomized planning methods as an approximation of the optimization problem. Rapidly exploring random trees (RRTs), for instance, provide one possible route to achieve computation efficiency by randomly exploring the state space with a bias toward unexplored regions, as shown in figure 14. However, if full car dynamics were considered then RRTs would rely heavily on finding a proper heuristic to guide estimates toward admissible regions constrained by the dynamics. [7]

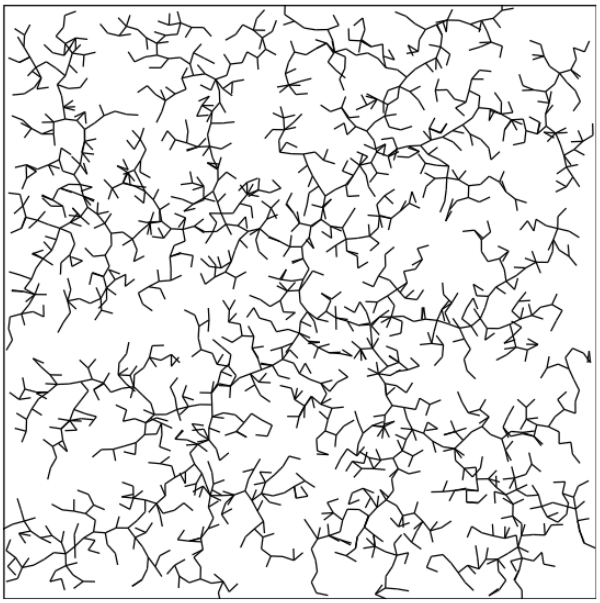

Figure 14. An RRT exploring a two-dimensional state space. [6]

APPENDICES

A. DYNAMIC PROGRAMMING ALGORITHM

Kirk's version of the computational dynamic programming algorithm is provided.

ACKNOWLEDGMENTS

I would like to thank Professor Hall for teaching what was a challenging, but very interesting course.

REFERENCES

[1] S. Hall, "Lecture 9: HJB Equation," pp. 1–32, 2018.

[2] ——, "Lecture 8 : Dynamic Programming," pp. 1–29, 2018.

[3] Y. Ma, *Dynamic Programming and the HJB Equation*. [Online]. Available: http://yima.csl.illinois.edu/psfile/ECE553/Lectures20-24.pdf

[4] Kirk, "Optimal control theory," pp. 1018–1018, 1971. [Online]. Available: http://doi.wiley.com/10.1002/aic.690170452

[5] G. Schildbach and F. Borrelli, "A dynamic

programming approach for nonholonomic vehicle maneuvering in tight environments," in *2016 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, jun 2016, pp. 151–156. [Online]. Available: http://ieeexplore.ieee.org/document/7535379/

[6] S. M. Lavalle, "Planning Algorithms," 2006.

[7] S. LaValle, "Randomized Kinodynamic Planning," 2001.

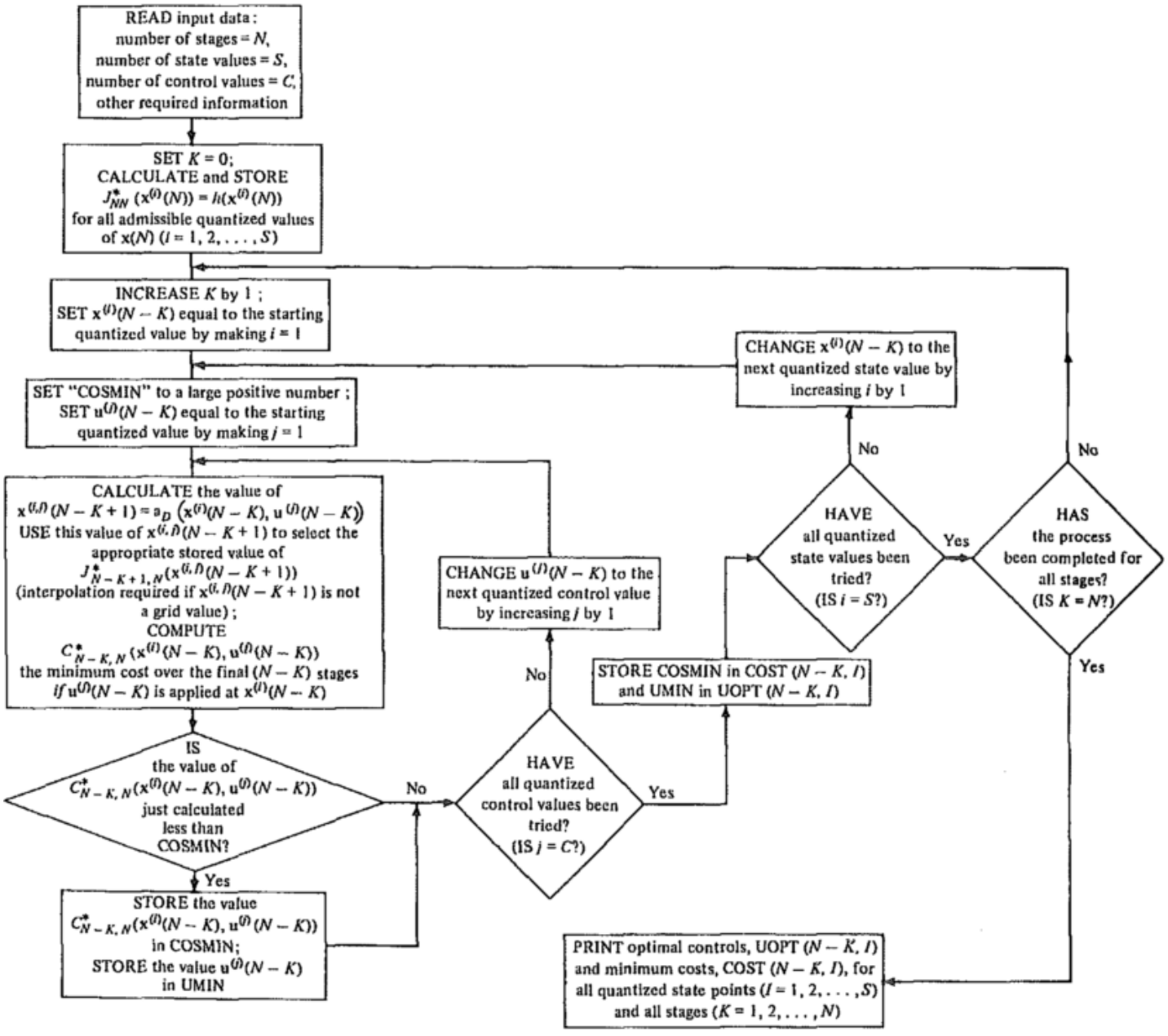

Figure 15. The general dynamic programming algorithm. [4]