CMPE252
Artificial Intelligence and Data Engineering

BUG algorithms

A Simple Robot Model

- Point robot in a 2D environment
- Can move in any direction with unit speed
- Perfect localization
- **Perfect “bump” sensor**
  - ***Can follow any obstacle boundary***

Environment

- Bounded 2D environment
- Finite number of obstacles
- Obstacles are well behaved
  - *A line intersects an obstacle finitely many times*

Path Planning

- *How to go from point A to point B?*
- Ideas?

Bug 0 Algorithm

- Repeat till you reach goal
  - Move towards the goal in a straight line (*m-line*)
  - If you hit an obstacle, follow its boundary in a clockwise fashion till you reach the *m-line* again

- *Does Bug 0 always reach the goal?*

Completeness

- An algorithm is called **complete** if:
  - it returns a feasible solution, if one exists;
  - returns FAILURE in finite time, otherwise

- **Bug 0** is not complete

Exercise

- Assume all obstacles are convex polygons.

- *Prove/Disprove: If the goal is reachable, then **Bug 0** guarantees that the robot reaches the goal.*

Fixing Bug 0

- Bug 0 has no measure of progress
  - gets stuck in a loop
  - can repeated leave the boundary following mode from the same point it started
  - common problem for purely greedy/local approaches
- Add some global measure of progress

Bug 2

- Stay on the line connecting start to goal (*m-line*)
- Suppose you hit an obstacle at a point (H_i)
- Follow the boundary until you reach either:
  - goal --> TERMINATE
  - *m-line* again --> check if we are closer. If yes, then move along *m-line*. Else continue.
  - H_i --> declare FAILURE

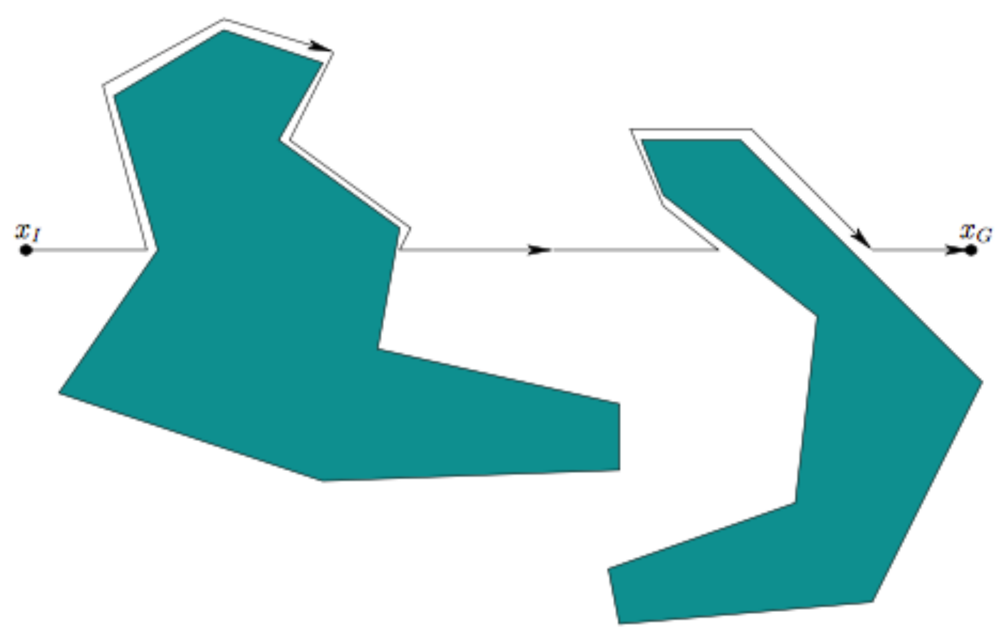


Figure 12.22: An illustration of the Bug2 strategy.

Completeness

- An algorithm is called **complete** if:
  - it returns a feasible solution, if one exists;
  - returns FAILURE in finite time, otherwise

- **Bug 2** is complete
  - If the goal is reachable, it guarantees that the robot reaches the goal
  - If the goal is not reachable, it reports failure in finite time

Bug 1

- Move towards the goal
- If an obstacle O_i is encountered at location H_i
  - Follow the boundary of O_i until H_i is revisited
  - Move to the point on O_i that is closest to the goal (call this point L_i)
- Continue

all figures from Ch 12. http://planning.cs.uiuc.edu/

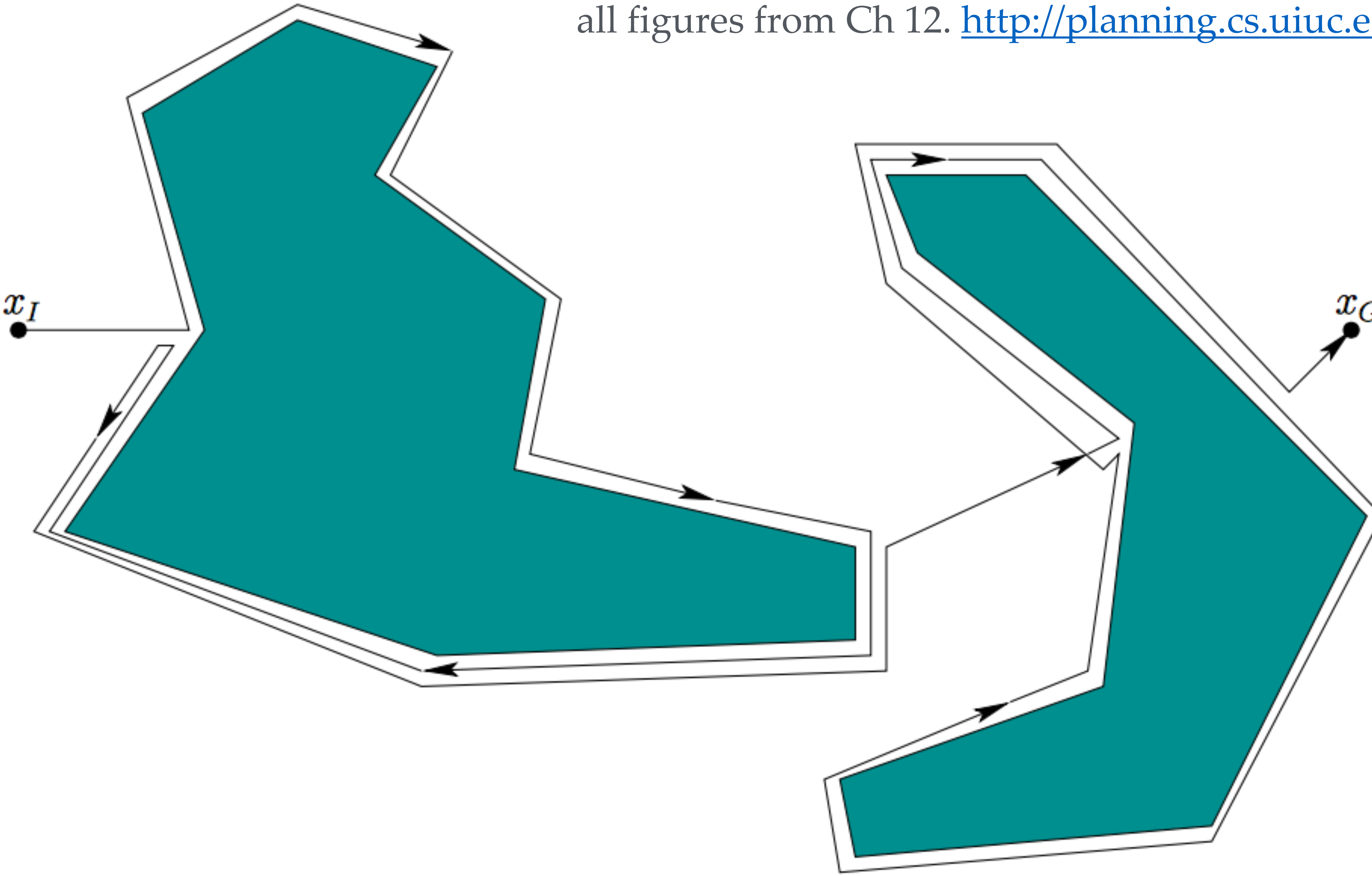


Figure 12.20: An illustration of the Bug1 strategy.

Readings

- Original paper [1]
- Chapter 2 of [2]
- Chapter 12.3.3 of [3]

[1] Lumelsky, Vladimir J., and Alexander A. Stepanov. "Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape." *Algorithmica* 2.1-4 (1987): 403-430. (URL)

[2] H. Choset et. al., "Principles of Robot Motion: Theory, Algorithms, and Implementations", MIT Press, 2005. (Available online)

[3] S. M. LaValle. “Planning Algorithms.” http://planning.cs.uiuc.edu/

Relevant Papers

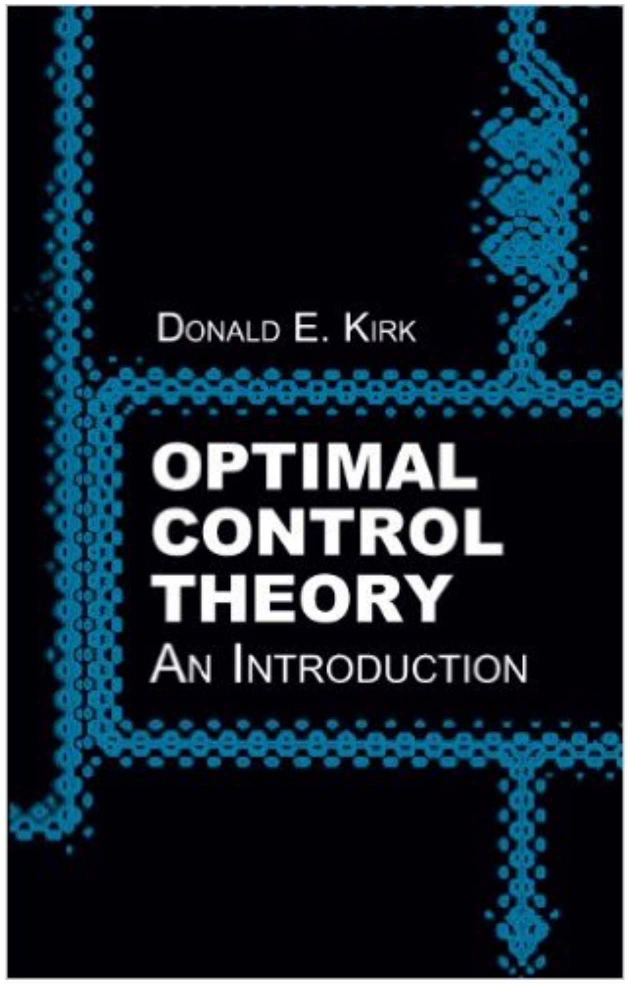
- Taylor, Kamilah, and Steven M. LaValle. "Intensity-based navigation with global guarantees." Autonomous Robots 36.4 (2014): 349-364. PDF
- Gabriely, Yoav, and Elon Rimon. "Cbug: A quadratically competitive mobile robot navigation algorithm." Robotics, IEEE Transactions on 24.6 (2008): 1451-1457.

CMPE252
Artificial Intelligence and Data Engineering

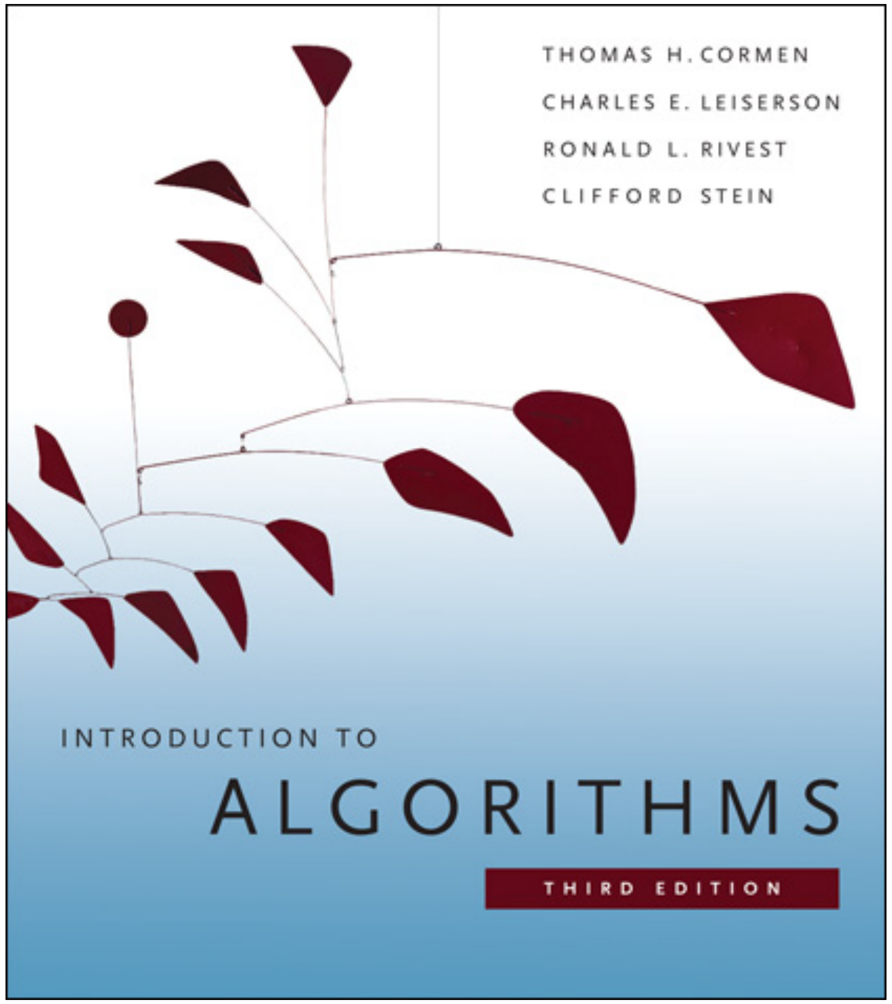
Dynamic Programming

Readings

Primary



Ch 3

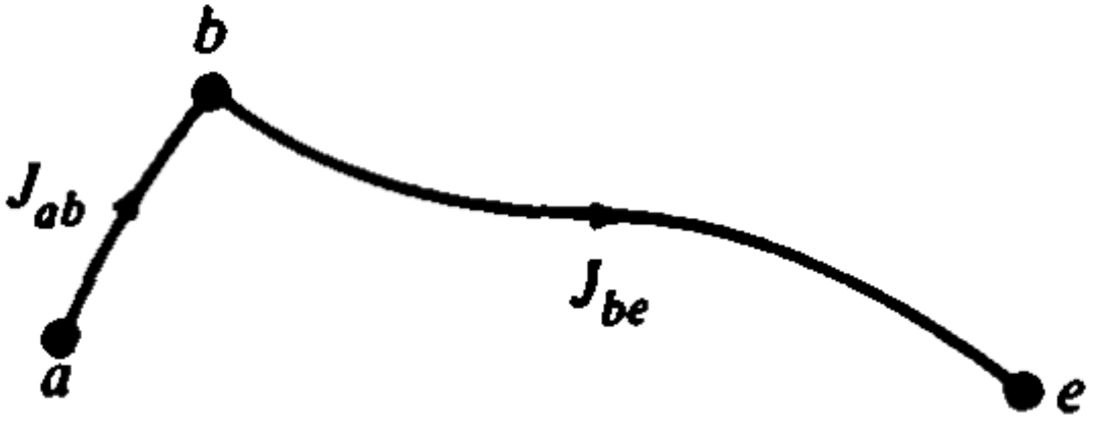


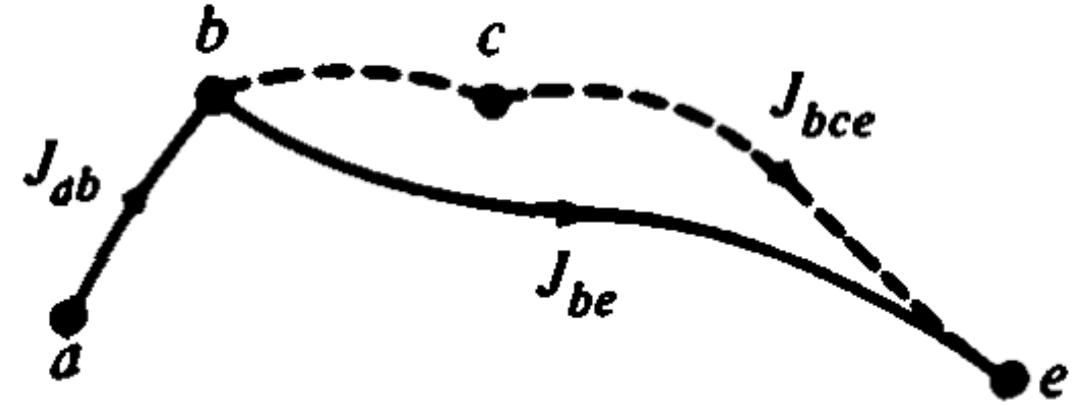
Ch 15

Etymology

- Dynamic Programming really means *multi-stage planning*
- Back in the days, *programming* referred to *planning* and not coding
- *Dynamic* is meant to reflect that this is multi-stage planning where you are choosing actions over multiple time steps instead of a single *static* choice

Optimal Sub-structure





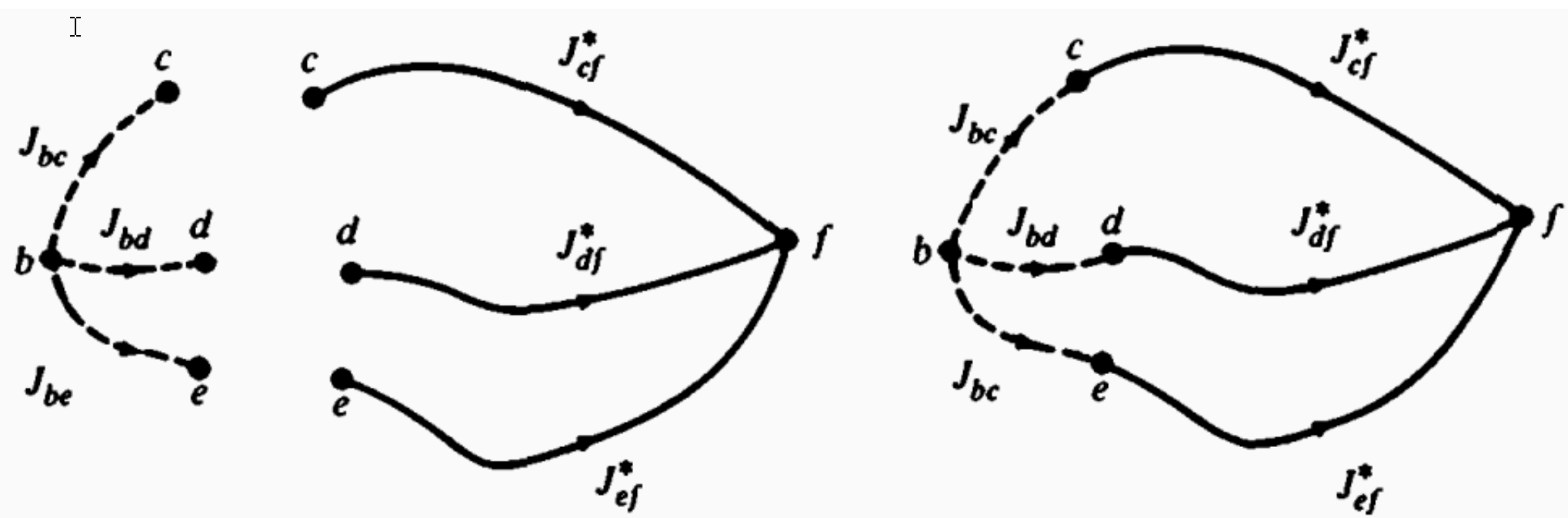
Claim: If *a*-*b*-*e* is the optimal path from *a* to *e*, then *b*-*e* is the optimal path from *b* to *e*.

Implementing Dynamic Programming

- Pick the minimum:

$$J_{bf}^* = \min \{ J_{bc} + J_{cf}^*, J_{bd} + J_{df}^*, J_{be} + J_{ef}^* \}$$

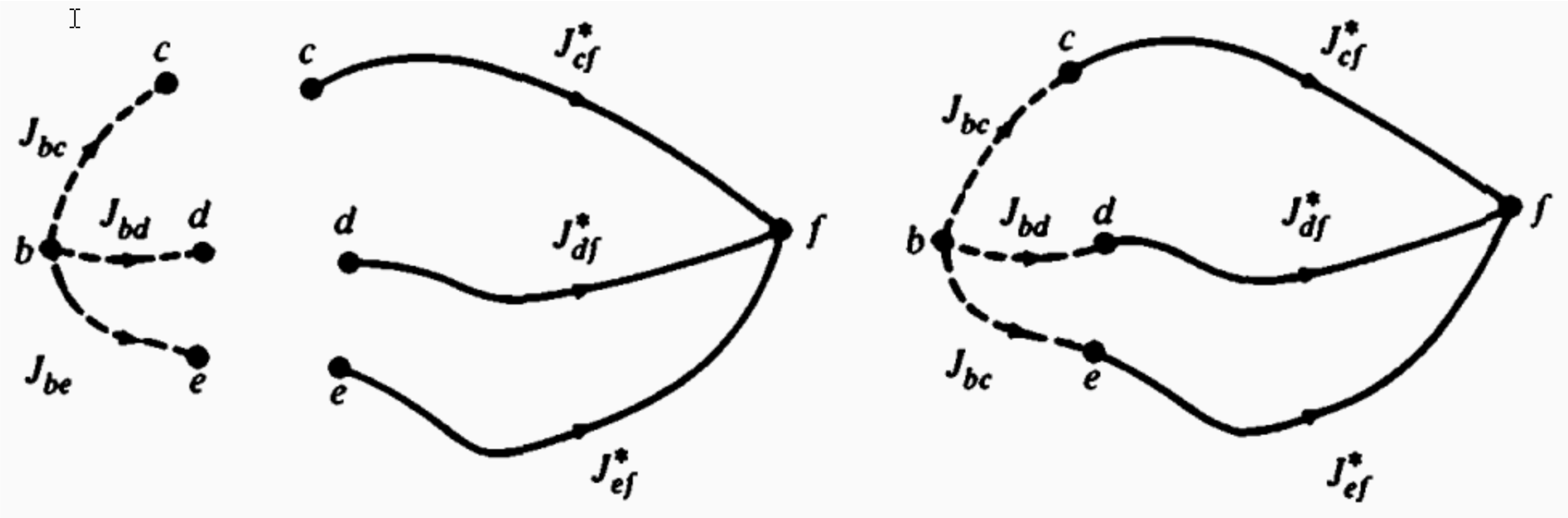
That is, $$J_{bf}^* = \min_{i} \{ J_{bi} + J_{if}^* \}$$



Implementing Dynamic Programming

- How do we know J_{if}^{*}?

$$J_{bf}^{*} = \min_{i} \{ J_{bi} + J_{if}^{*} \}$$

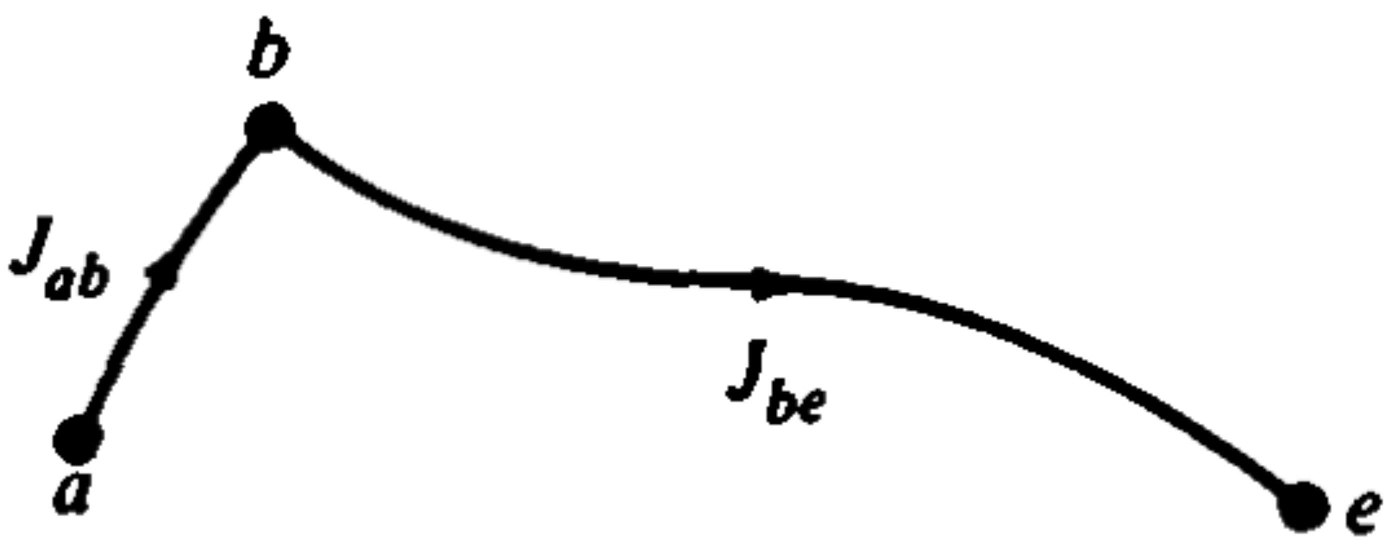


“Memoization”

- DP is overlapping sub-problems + optimal sub-structure
- Memoization refers to the fact that we are solving sub-problems and recording their solutions
- That is, we are making a *memo*.

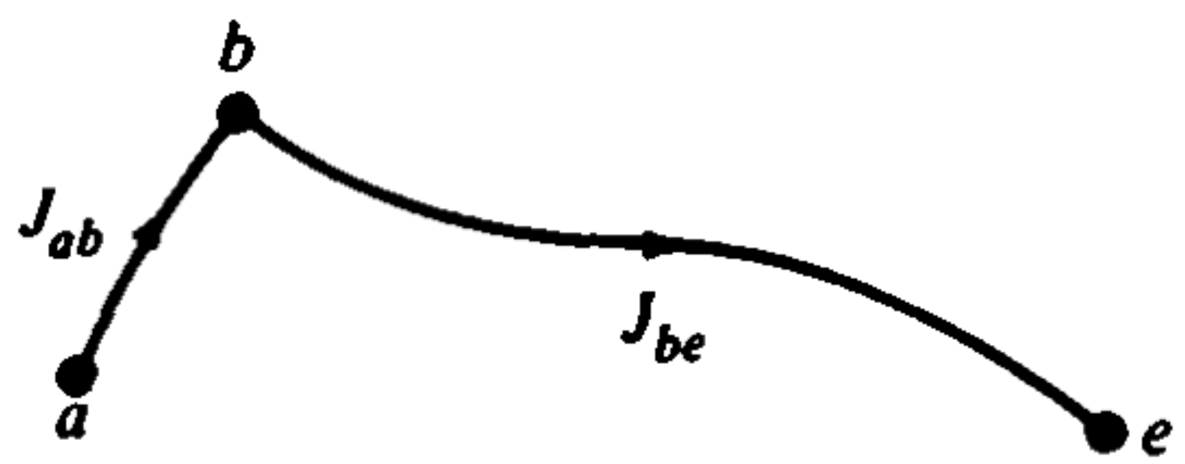
Not all problems have optimal substructure

- Consider the problem of computing the *longest* path from from *a* to *e*
  - with no repeated vertices
- *Does this problem have optimal substructure?*



Not all problems have optimal substructure

- Claim: If *a-b-e* is the longest path from *a* to *e*, then *b-e* cannot be the longest path from *b* to *e*.



Lets use better notation

- Find the shortest path from x_O to x_G
- $C(x_i, x_j)$ = cost of edge from x_i to x_j
  - If no edge between x_i to x_j, then $C(x_i, x_j) = inf$
- $V(x_i)$ = cost to go from x_i to x_G
  - also called the value function
- $N(x_i)$ = vertices that are (outgoing) neighbors of x_i

$$V(x_i) = \min_{x_j \text{ in } N(x_i)} \{ C(x_i, x_j) + V(x_j) \}$$

$$V(x_G) = 0$$

Bellman Equation

$$V(x_i) = \min_{x_j \text{ in } N(x_i)} \{ C(x_i, x_j) + V(x_j) \}$$

$$V(x_G) = 0$$

Can we solve this recursion?

Building a table

$$V_k(x_i) = \min_{x_j \text{ in } N(x_i)} \{ C(x_i, x_j) + V_{k-1}(x_j) \}$$

$V_k(x_i)$ = cost to go from x_i to x_G using no more than k edges.

An optimal path in a graph with n vertices cannot have more than n-1 edges.

Therefore, table has size n x $(n - 1)$.

More on the Value Function

- Also called *cost-to-go* function
- If we know *V(x)* we know optimal paths from *all* states to the goal state
  - comes at the expense of efficiency
  - we will soon see a more efficient single start, single goal path planning algorithm
- *V(x)* gives you a feedback policy!
  - even if the robot deviates from the optimal path, we still know how to reach the goal

Value Function

- *Claim*: If you know the value function, you can compute the optimal path directly.

Value Function

- *Claim*: If you know the value function, you can compute the optimal path directly.

- Just do steepest gradient descent:
  - At any state x choose to go to a neighbor that has the lowest cost-to-go. Repeat.

Cost-to-come function

$$V_k(x_i) = \min_{x_j \text{ in } N(x_i)} \{ C(x_j, x_i) + V_{k-1}(x_j) \}$$

Instead of building the table from goal to start, we can fill it from start to goal.

Redefine $V(x_i)$ to be *cost-to-come* instead of *cost-to-go*.

$V_k(x_i)$ = cost to come from x_O to x_i using no more than k edges.

$N(x_i)$ = incoming neighbors.