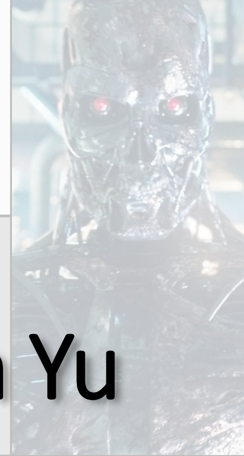
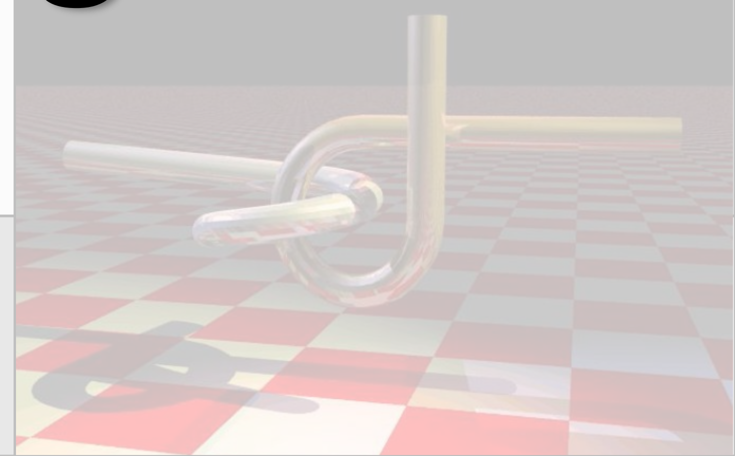
CS 460/560
Introduction to Computational Robotics
Fall 2019, Rutgers University

Lecture 10
Graph Search Algorithms





Instructor: Jingjin Yu

Outline

Solving problems via search

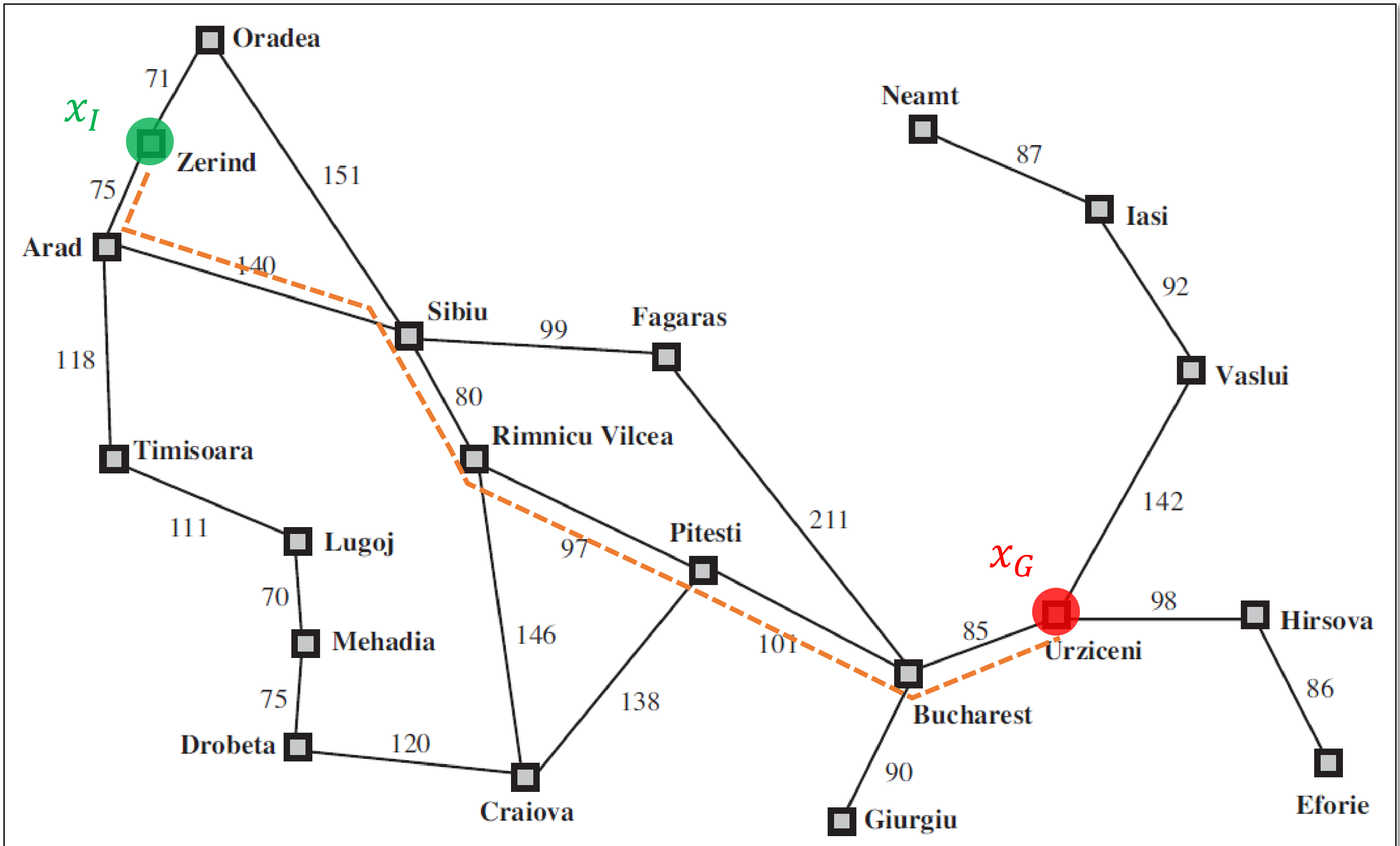
Basic graph search algorithms

- ⇨Graph
- ⇨BFS
- ⇨DFS
- ⇨Uniform cost
- ⇨A*

All pairs shortest path

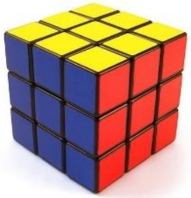
A word on the principle of dynamic programming

Solving Problems through Search: An Example

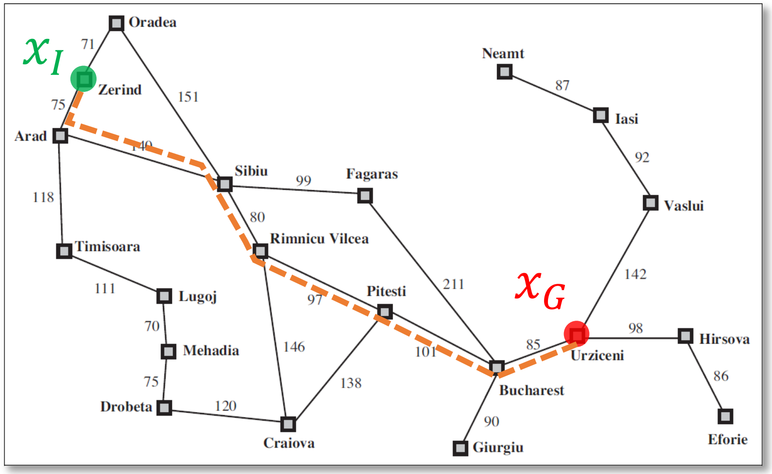


Search for a route in Romania

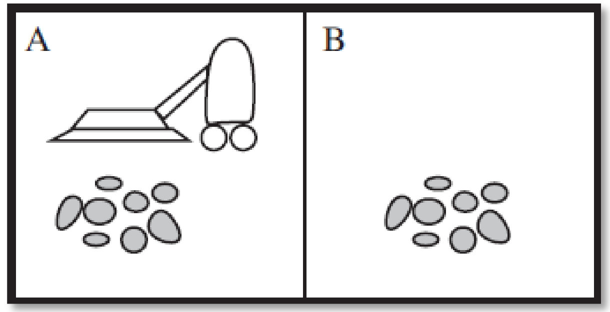
Components of a Search Problem



- ⇨**State space** S: in this case, an edge-weighted graph
- ⇨**Initial** (start) and **goal** (final) states: x_I and x_G
  - ⇨There can be more than one start/goal state: solve one side of a Rubik's cube
- ⇨**Action**: in this case, moving from one state to a nearby state
- ⇨**Transition model**: tuples (s_1, a, s_2) that are valid
  - ⇨Sometimes written as $T(s_1, a) = s_2$
  - ⇨There are usually costs/rewards associated with a transition, $R(s_1, a)$
- ⇨**Solution**: valid transitions connecting x_I and x_G
  - ⇨Optimal solution: solution with lowest cost (e.g., length of the path)



Example: Vacuum-Cleaner World



⇨State space: $\{A, B\} \times \{A_{dirty}, A_{clean}\} \times \{B_{dirty}, B_{clean}\}$

⇨State space size: $2 \times 2 \times 2 = 8$

⇨Action: $\{left, right, suck\}$

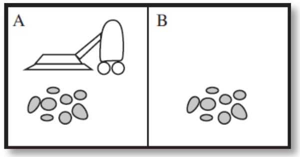
⇨Transition example: $(A, A_{clean}, B_{dirty}) \xrightarrow{\text{right}} (B, A_{clean}, B_{dirty})$

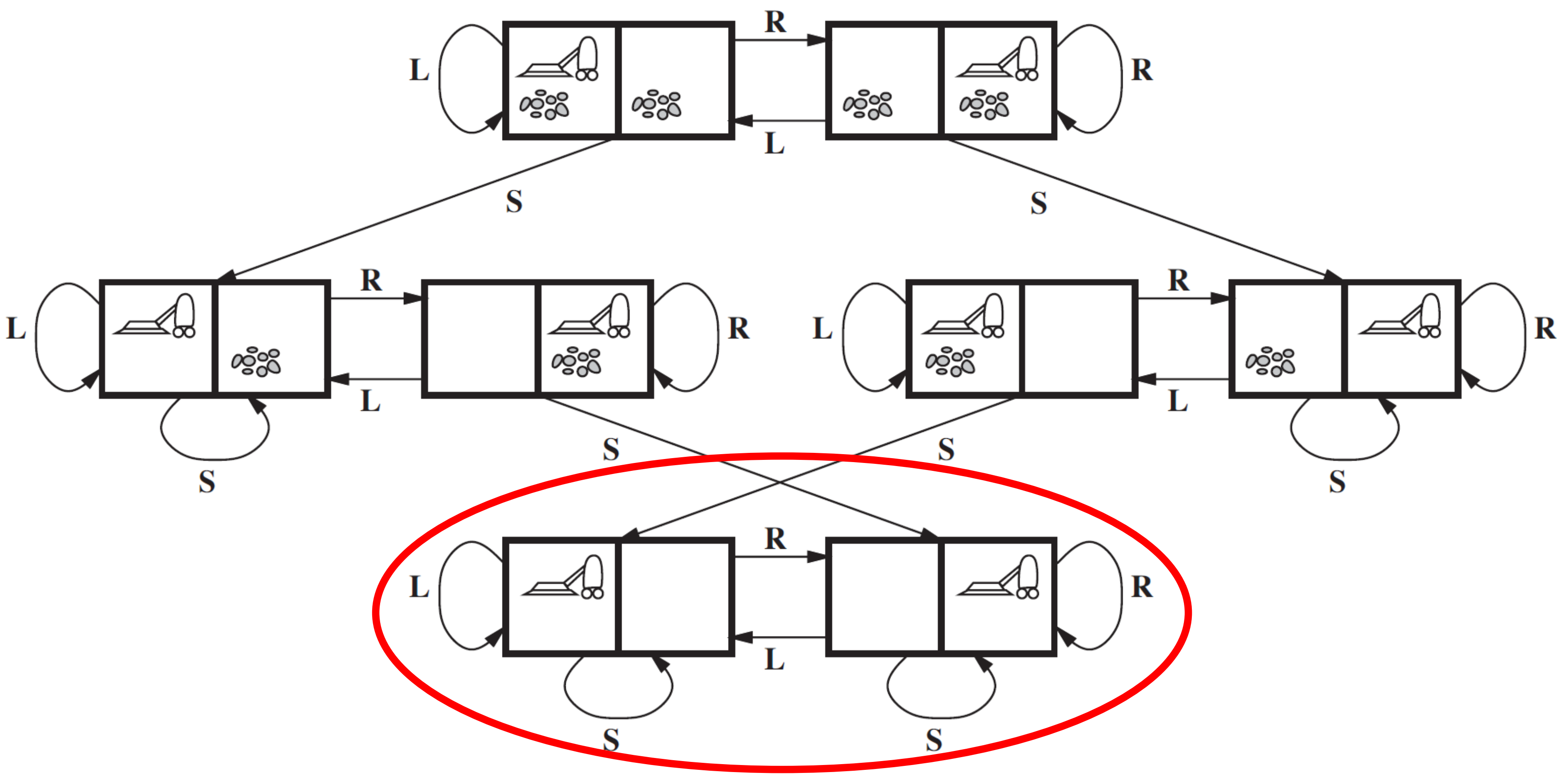
⇨Initial state: can be an arbitrary state

⇨Goal states: $\{A, B\} \times \{A_{clean}\} \times \{B_{clean}\}$

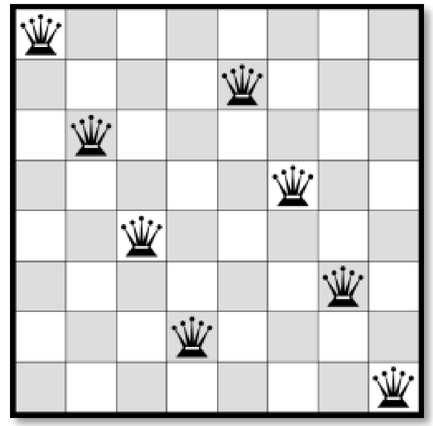
⇨Cost: can be the number of actions in a solution

State Space of the Vacuum-Cleaner World



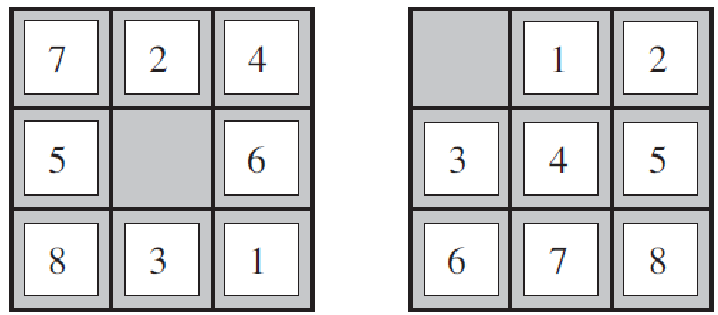


Example: 8-Queens



- ⇨State space: possible locations of 8 queens
  - ⇨State space size: $C(64,8) = \frac{64\times 63\times\cdots\times 57}{8!} \approx 4\times 10^9$
- ⇨Action/transition: place or move a queen
- ⇨Initial state: can be an arbitrary state
- ⇨Goal states: a placement of queens in which no two queens attacking
- ⇨Cost: no clear cost

Example: 8-puzzle



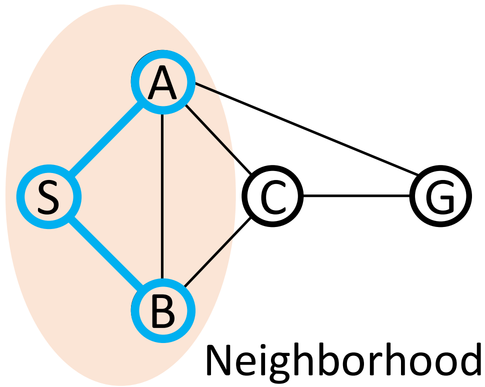
Start State Goal State

- ⇨State space: arrangements of the 8 pieces
  - ⇨State space size: $9! = 362880$
- ⇨Action/transition: shifting a piece to the empty cell
- ⇨Initial state: an arbitrary state
- ⇨Goal state: an arbitrary fixed state
- ⇨Cost: number of actions (moves)

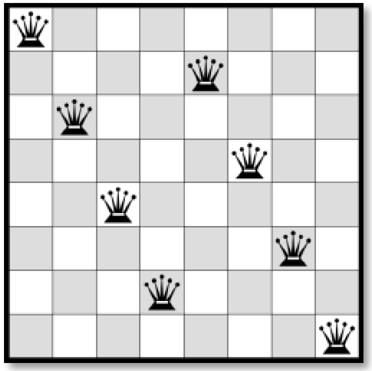
Discrete Search Algorithms

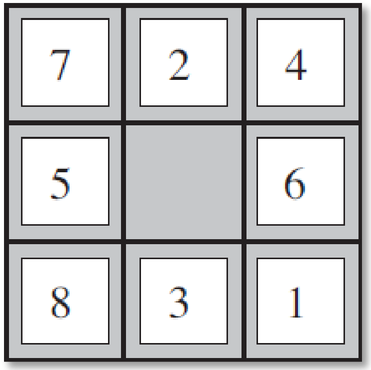
What are discrete search algorithms?

- ⇨An algorithm whose **data structure** is a graph
- ⇨That is, a set of “nodes” with “neighborhoods”
- ⇨The graph may be explicit
- ⇨Or it may be implicit
  - ⇨ What are the nodes here?
  - ⇨ And neighborhoods?
- ⇨And even not fully known!



Neighborhood of S

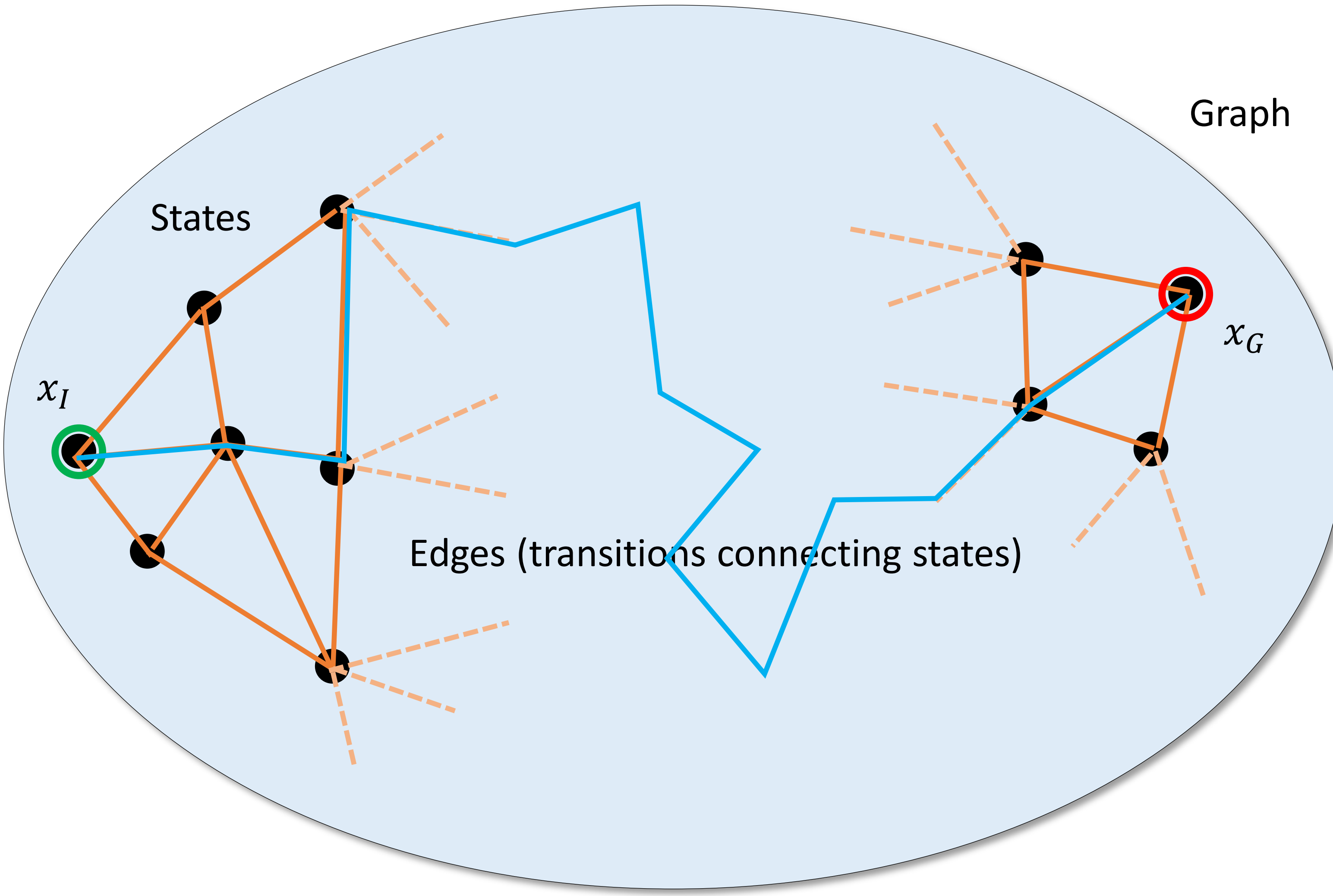

8 Queens



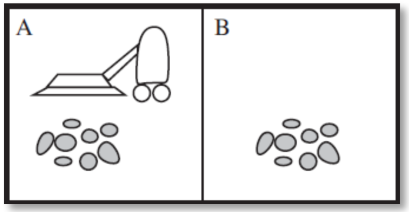
8-puzzle

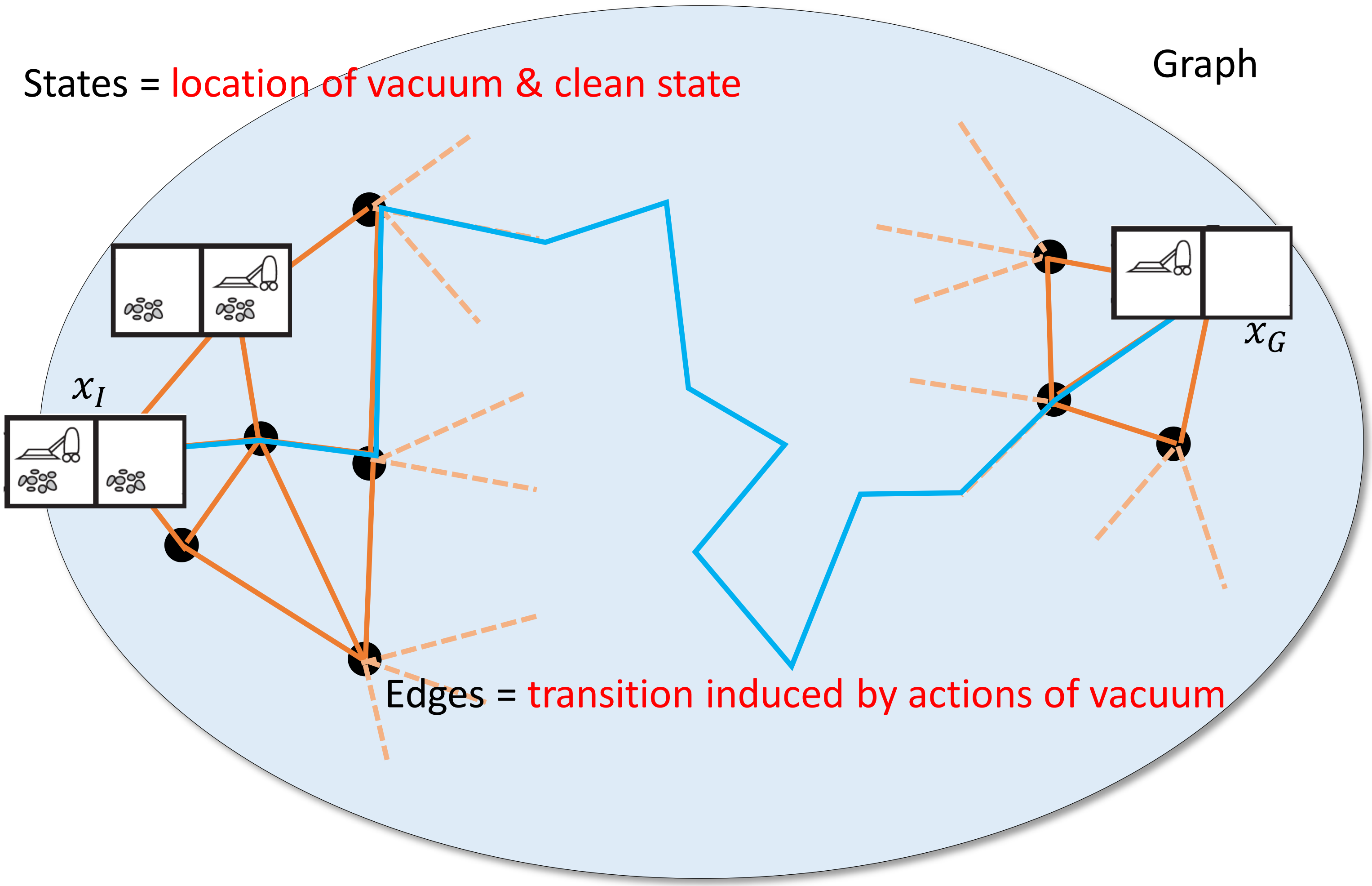

C&C: Red alert (Electronic Arts)

Graph View of Search: A Recap

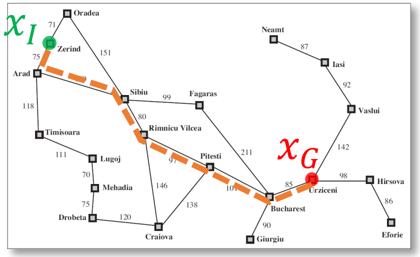


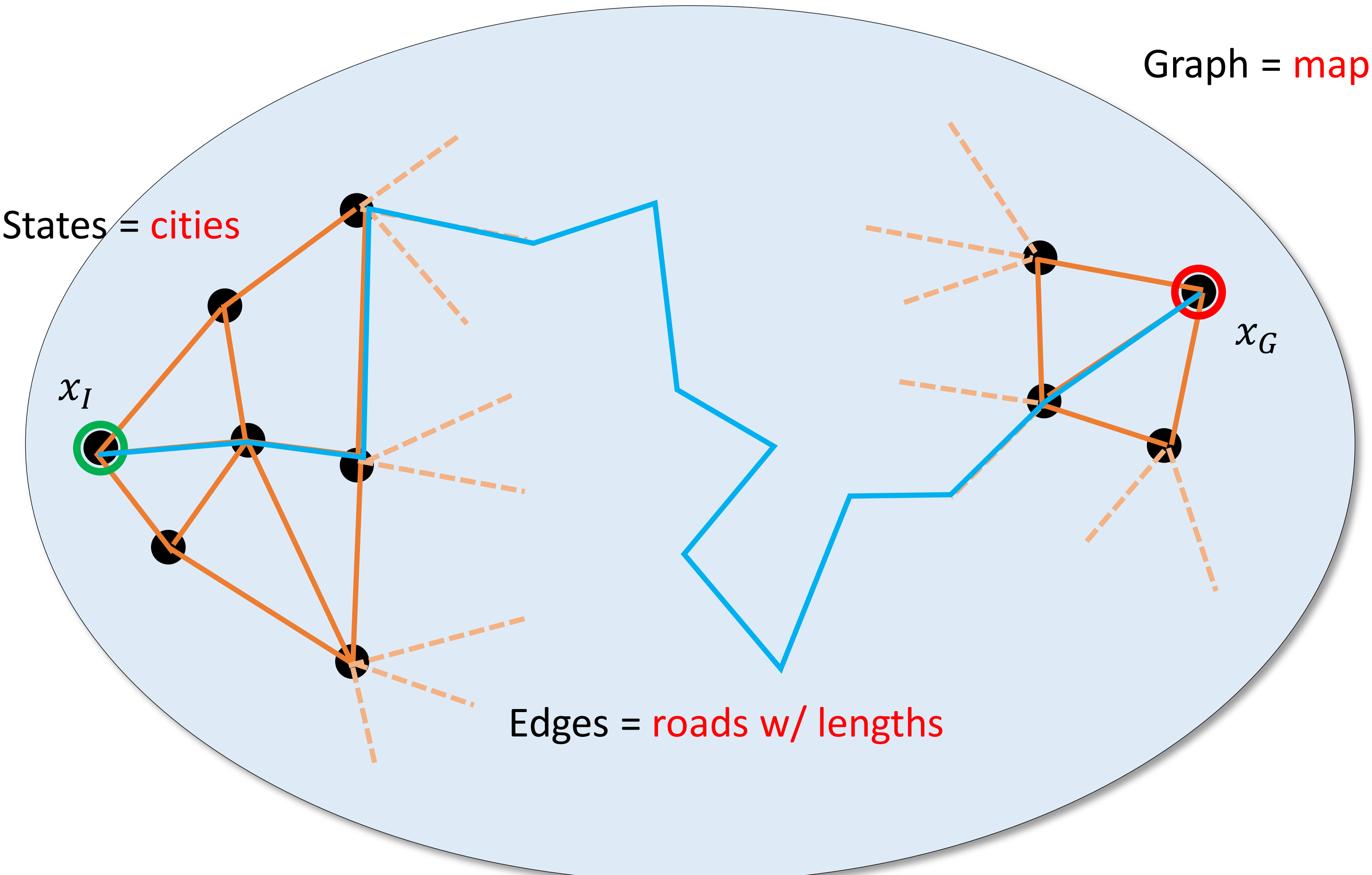
Examples Revisited: Vacuum World



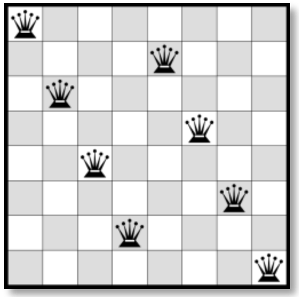


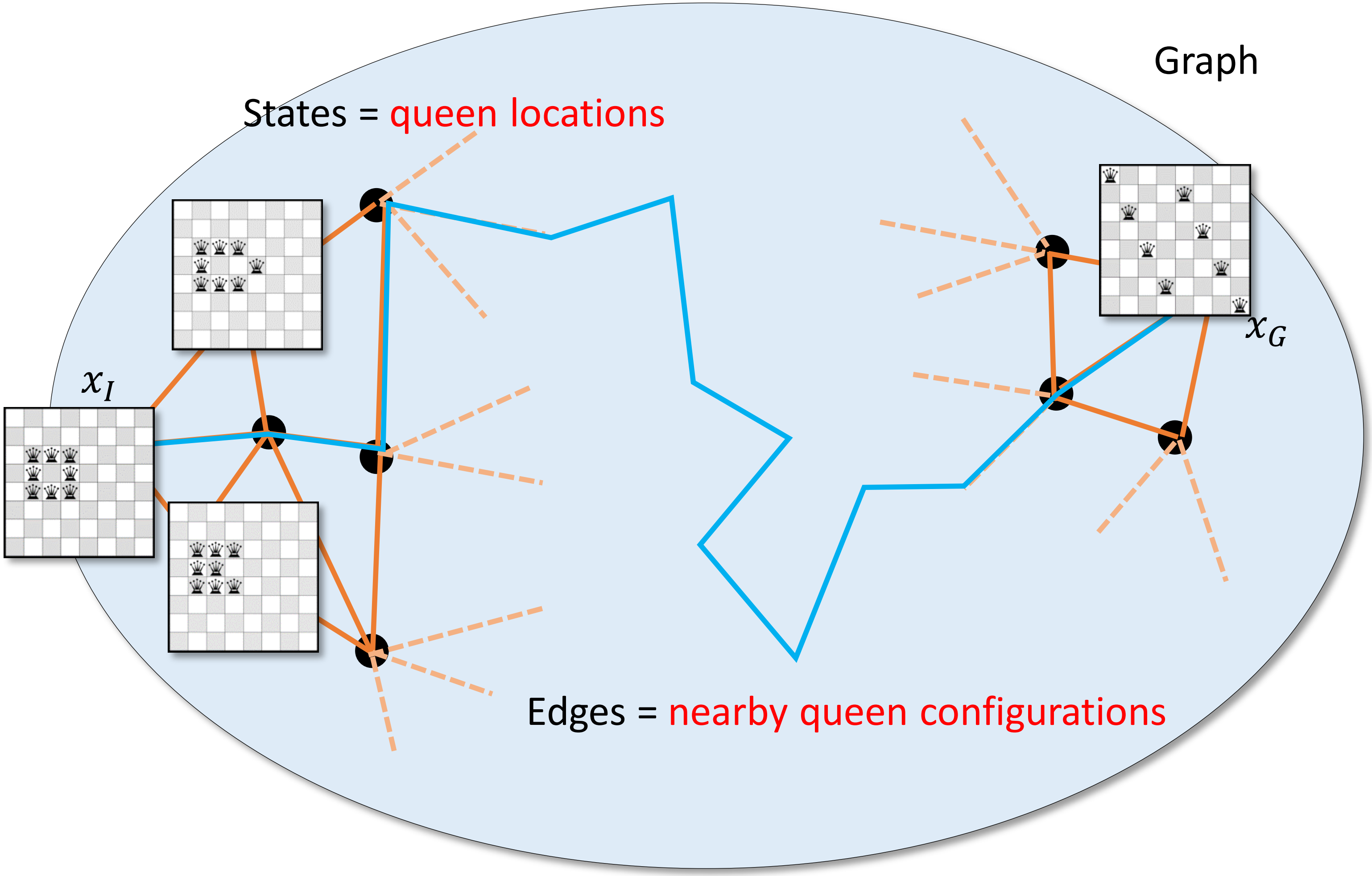
Examples Revisited: Navigation



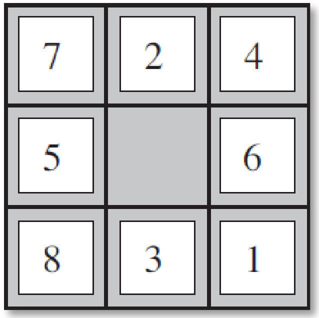


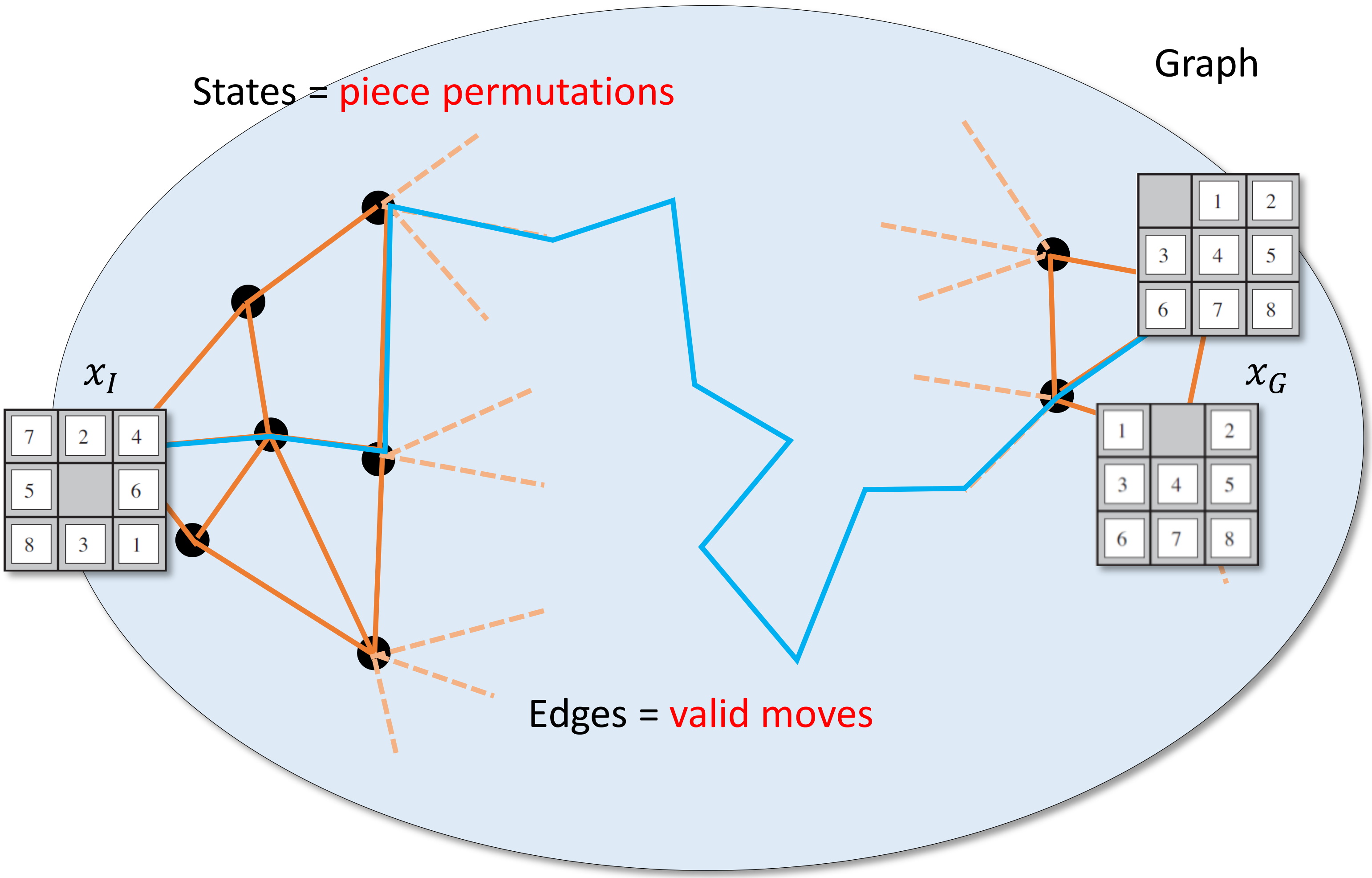
Examples Revisited: Eight Queens





Examples Revisited: Eight Puzzle





Applications of Discrete Search Algorithms

Navigation

Robot motion planning

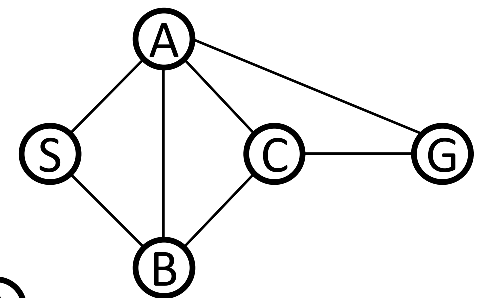
Competitive chess and go

Game AI

Graph Basics

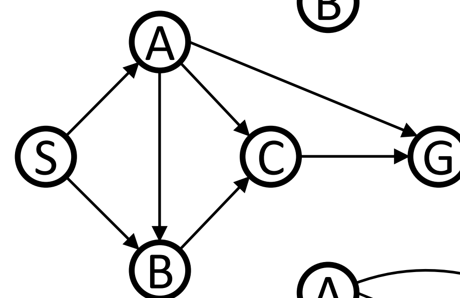
A graph $G = (V, E)$ is a set of vertices V and a set of edges E

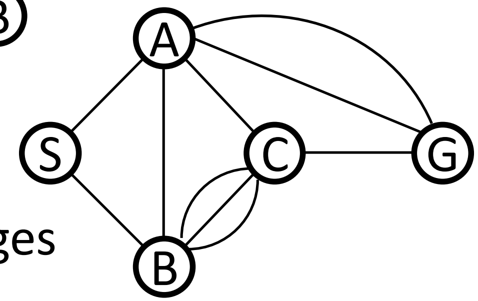
- Example
  - $V = \{A, B, C, G, S\}$
  - $E = \{(A,B), (A,C), (A,G), (A,S), (B,S), (B,C), (C,G)\}$



Variations

- A graph may be **directed**
- There can be **multi-edges** between two vertices
  - This is called a **multi-graph**
  - We will not consider multi-graphs in our course





Basic properties

- An undirected graph with n vertices has **at most** $\frac{n^2-n}{2}$ edges
  - When this happens, the graph is a **complete** graph
- A graph is **connected** if there is a path between any two vertices
- A connected graph with $n-1$ edges is a **tree**

A Generic Graph Search Algorithm

```
input: G = (V,E), x_I, x_G
AddToQueue(x_I, Queue);  // Add x_I to a queue of nodes to be expanded
while(!IsEmpty(Queue))
     x ← Front(Queue);                  // Retrieve the front of the queue
     if(x.expanded == true) continue;    // Do not expand a node twice
     x.expanded = true;                 // Mark x as expanded
     if(x == x_G) return solution;      // Return if goal is reached
     for each neighbor n_i of x // Add all neighbors of to the queue
          if(n_i.expanded == false) AddToQueue(n_i, Queue)
return failure;
```

Different graph search algorithms (breadth first, depth-first, uniform-cost, …) differ at the function `AddToQueue`

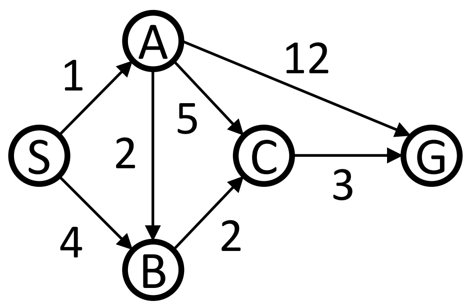
To retrieve the actual path, use **back pointers**

Classical Search Algorithms

- ⇨Breadth-first search (BFS)
  - ⇨Always add new nodes at the **end** of the queue
- ⇨Depth-first search (DFS)
  - ⇨Always add new nodes in the **front** of the queue
- ⇨Uniform-cost (Dijkstra’s)
  - ⇨Always keep the node with the **best cost** in the front of the queue
- ⇨A*
  - ⇨Similar to uniform-cost, but also uses a **guess** of distance to goal

Breadth First search

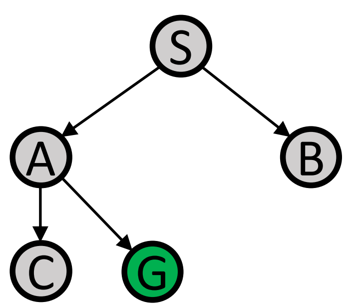
Problem graph (a weighted directed graph)



=

Neighbor list
S: A, B
A: B, C, G
B: C
C: G
G:

Running BFS graph search (we do not use weights here)



Q: S

Q: A, B

Q: B, C, G

Q: C, G

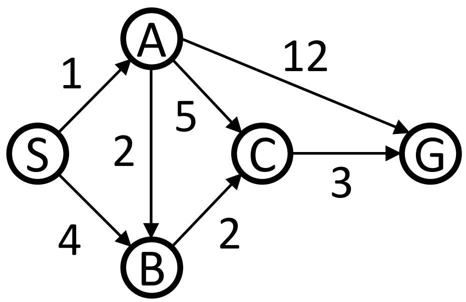
Q: G

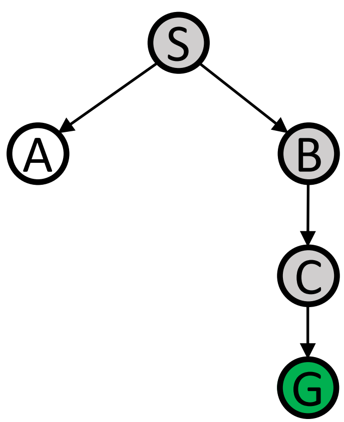
Q:

Path can be retrieved by storing S in A and A in G as back pointers.

Depth First search

Running DFS graph search (again we do not use weights here)





Q: S

Q: B, A

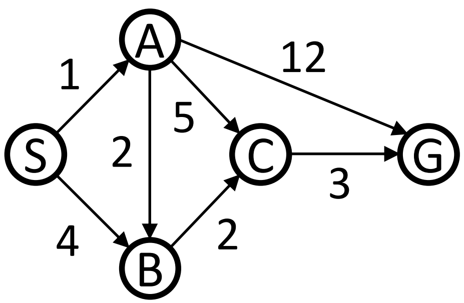
Q: C, A

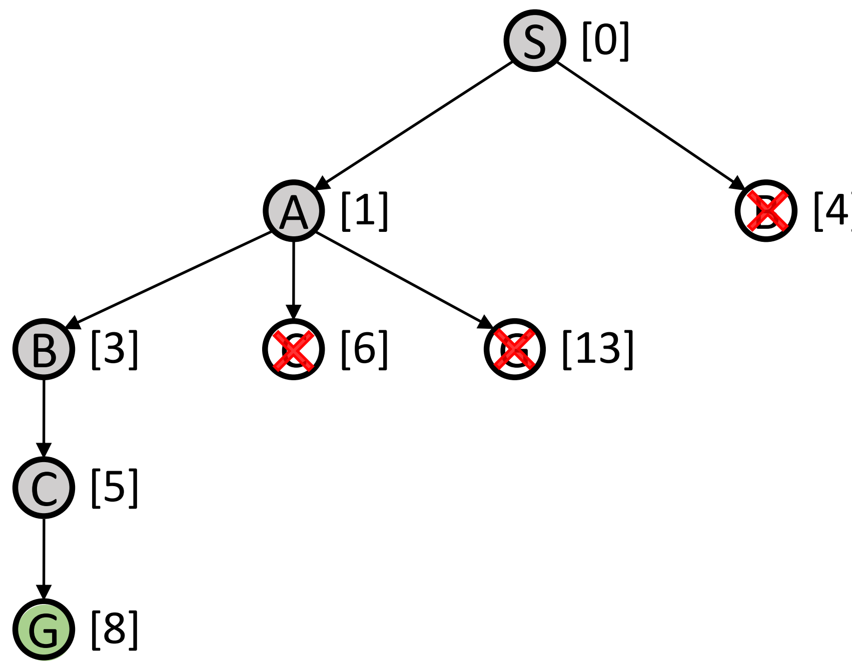
Q: G, A

Q: A

Uniform-Cost Search

Maintain queue order based on current cost





⇨Produces **optimal** path!

⇨This is basically the Dijkstra's algorithm

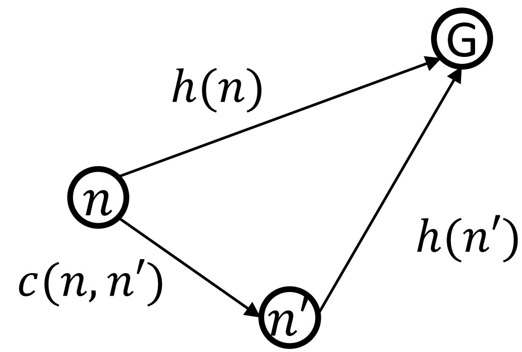
Admissible and Consistent Heuristic

- Assume the cheapest path from x to a goal is $c(x)$, an **admissible heuristic** satisfies

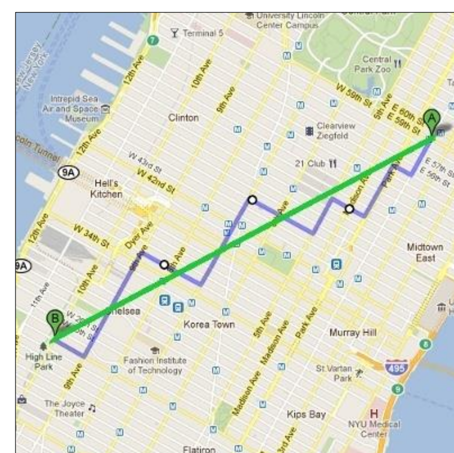
$$\boxed{h(x) \leq c(x)}$$

- A **consistent** heuristic is defined as

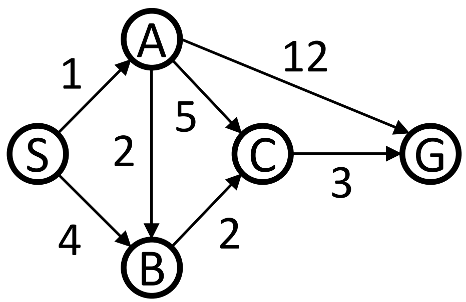
$$\boxed{h(n) \leq c(n, n') + h(n')}$$



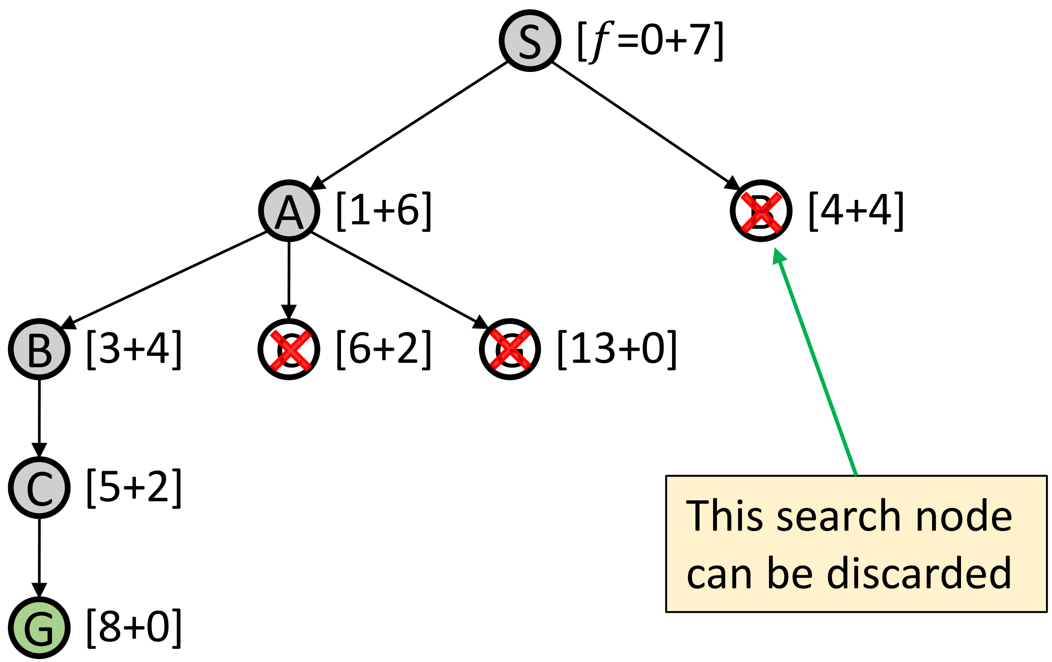
  - A form of triangle inequality
- A **consistent** heuristic is **always admissible**
  - The reverse is not always true
- Example of heuristic functions
  - Manhattan distance
  - Straight-line distance
    - Consistent



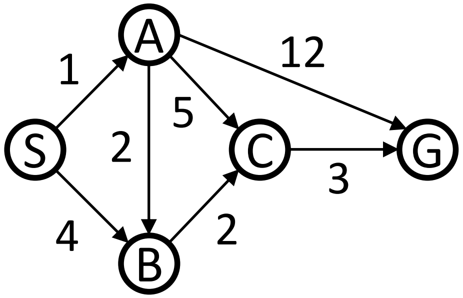
A* Search w/ a Consistent Heuristic

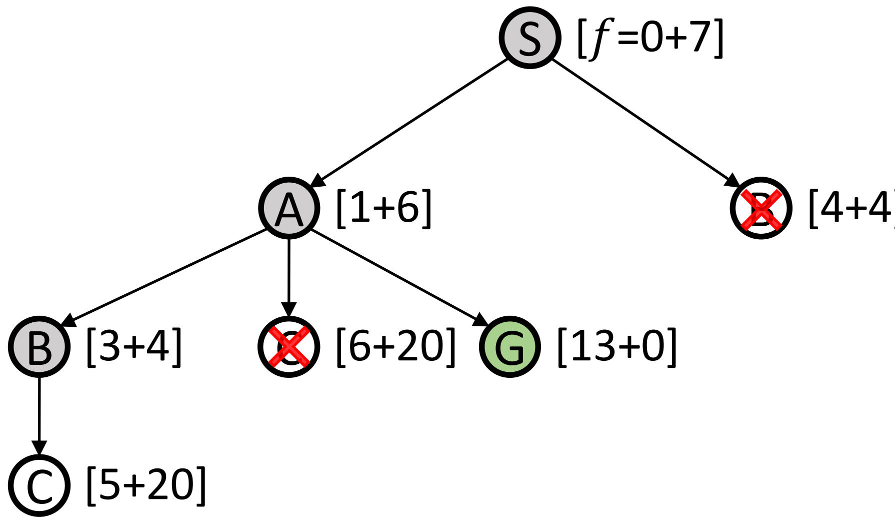


State	$h(x)$
S	7
A	6
B	4
C	2
G	0



A* Search w/ an Inadmissible Heuristic



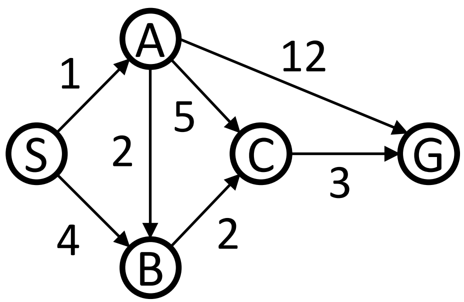


State	$h(x)$
S	7
A	6
B	4
C	20
G	0

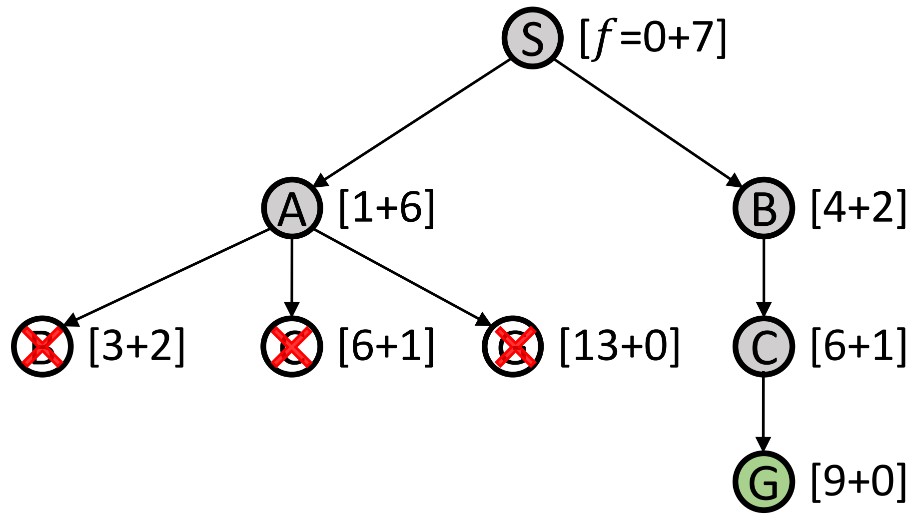
Not optimal!

$h(x)$ is **inadmissible**, e.g., $h(\mathrm{C}) = 20 > 3 = c(\mathrm{C})$, the actual cost from C to G

A* Search w/ an Inconsistent Heuristic



State	$h(x)$
S	7
A	6
B	2
C	1
G	0



Not optimal!

$h(x)$ is **inconsistent**, e.g., $h(S) > c(S, B) + h(B)$, $h(A) > c(A, B) + h(B)$

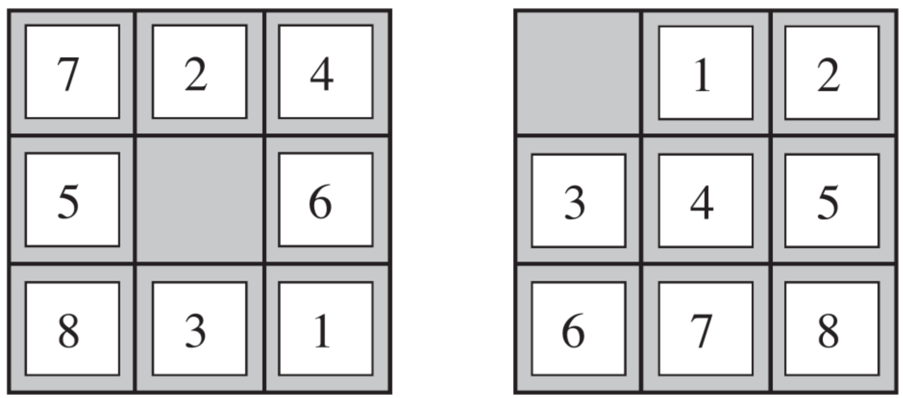
Heuristic Function Design

For route finding problems, Euclidean distance is consistent

⇨Also very efficient!

Designing heuristic functions can be **non-trivial**

Consider two heuristics for the 8-puzzle

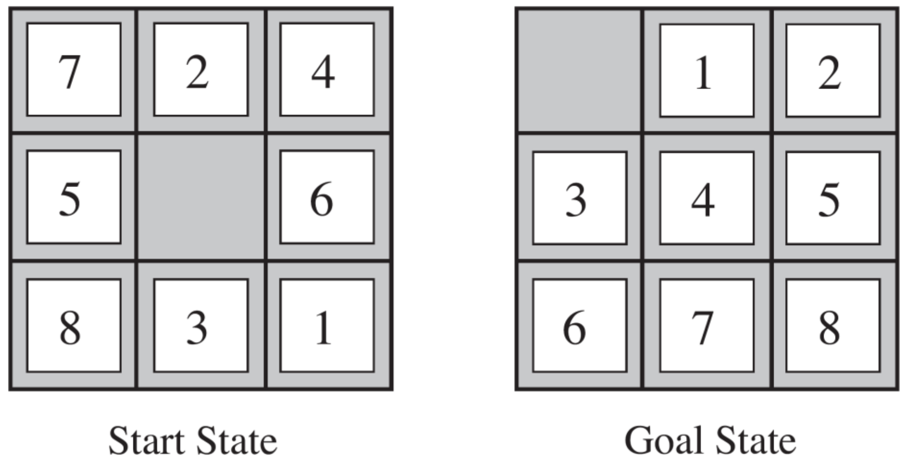


Start State Goal State

⇨h_1: number of misplaced pieces

⇨h_2: sum of Manhattan distances to goal for all pieces

Heuristic Function Design



h_1: #misplaced game pieces = 8

h_2: sum Manhattan distances = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18

Both heuristics are admissible and consistent

⇨How to choose?

⇨Generally, using the largest h is preferred: closer to the actual cost

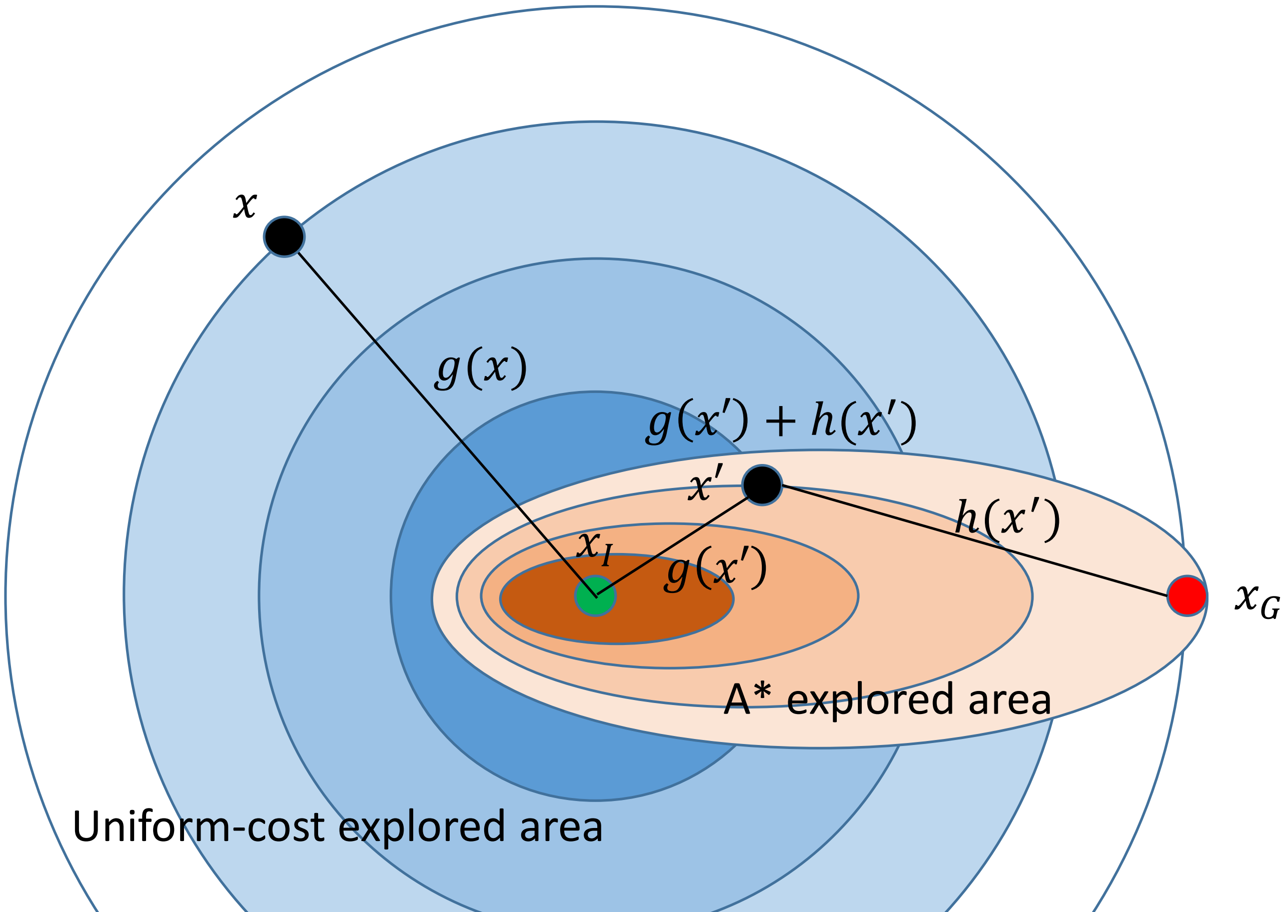
⇨Because h are **underestimates**

⇨In this case, we can use h_2 or simply $h = \max\{h_1, h_2\}$

Advantage of A* Search

Both A* and uniform-cost (i.e., Dijkstra's) are optimal. Why A*?

- ⇨Because A* biases the search toward the goal
  - ⇨ A* may visit much fewer nodes
- ⇨Similarly, better heuristic → smaller explored area



All Pairs Shortest Paths (Floyd-Warshall)

Floyd-Warshall is a type of dynamic programming

- ⇨So are most other optimal search algorithms (e.g., Dijkstra, A*)
- ⇨Pseudo-code

```
let dist be a |V| × |V| array of minimum distances initialized to ∞
for each vertex v
   dist[v][v] ← 0
for each edge (u,v)
   dist[u][v] ← w(u,v) // the weight of the edge (u,v)
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][j] > dist[i][k] + dist[k][j] then
            dist[i][j] ← dist[i][k] + dist[k][j]
end if
```

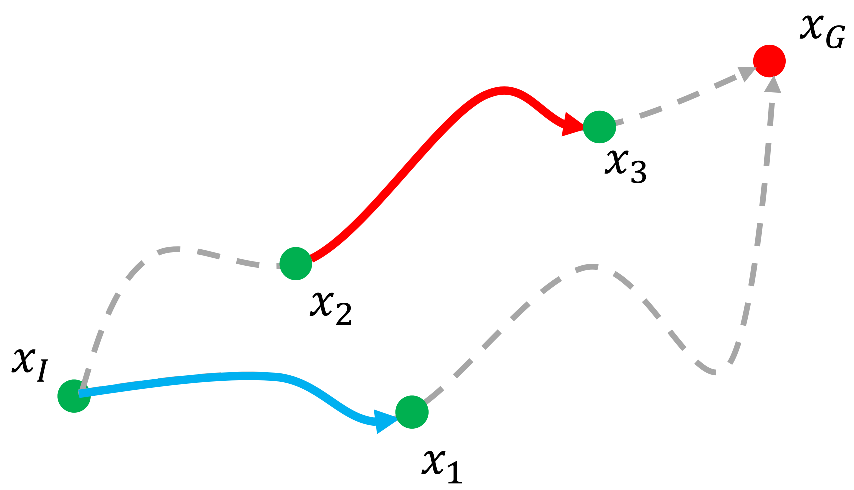
- ⇨Does not directly produce paths
- ⇨A path tree from any vertex can be constructed by adding back pointers
- ⇨Runs in time $O(|V|^3)$

Algorithm outline adapted from Wikipedia page on Floyd-Warshall

The Principle of Dynamic Programming

Dynamic programming is a type of recursion

- ⇨ It breaks a big problem into smaller pieces
  - ⇨ E.g., $P(n) = P(n_1) + P(n_2)$ with $n = n_1 + n_2$
- ⇨ The problem must have structures that allow computation to be reused
  - ⇨ E.g., for path optimality, any segment of an optimal path must also be optimal



- ⇨ Divide-and-conquer search algorithms, e.g., merge-sort, are special cases
  - ⇨ $P(n) = P(n_1) + P(n_2)$ for $n_1 = n_2 = \frac{n}{2}$
- ⇨ Dijkstra's and A* are also types of dynamic programming

Dynamic Programming in Search

Recall `AddToQueue` is the crucial step of graph search

- ⇨BFS and DFS do not care about edge costs
- ⇨Uniform cost and A* do
- ⇨This is in fact dynamic programming!
- ⇨Priority of unvisited = cost-to-come + estimated cost-to-go
- ⇨I.e., $f = g + h$
  - ⇨ g, cost-to-come, is fixed
  - ⇨ h, estimated cost-to-go, determines algorithm behavior
  - ⇨ Uniform cost: $h = 0$
  - ⇨ A*: h is consistent
  - ⇨ Other behaviors are possible by changing h

D* Algorithm Brief Intro

D* and D*-lite stand for "dynamic" A*

- ⇨Supports previously unknown obstacles
- ⇨Initially, for parts of the environment that is unknown, assume no obstacle
- ⇨Runs A* backwards to find an initial optimal solution
- ⇨Then, execute the path
- ⇨If we hit an obstacle along the way
  - ⇨ Update the node itself to be unavailable
  - ⇨ Put all its descendent nodes on the queue for search again
  - ⇨ Do the above step recursively
  - ⇨ Restart the A* search to find an optimal path
- ⇨Repeat the previous two steps
- ⇨One may view D* as running many A* searches
- ⇨A new A* search will be run as a previously unknown obstacle is met
- ⇨More details: https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf