CMSC 373 Artificial Intelligence
Fall 2025
04-Problem Solving & Search

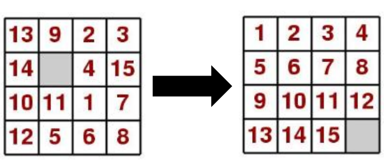
Deepak Kumar
Bryn Mawr College

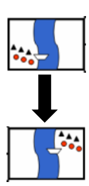
1

Search in AI

- Search in AI is a problem solving technique.
  Not the same as a web search (*ala* Google)

- Given a problem, find a way (path) to get from an initial state to a goal state.





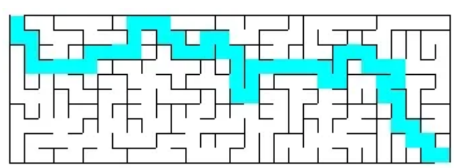


Image: https://medium.com/swlh/solving-mazes-with-depth-first-search-e315771317ae

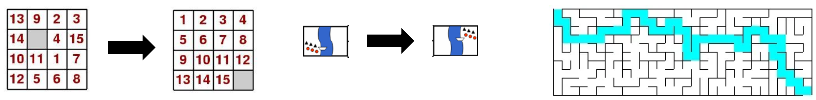
Image: https://personal.math.ubc.ca/~cass/courses/m308-02b/projects/grant/fifteen.html

2

2

Search Formulation

- **State**: A data structure that represents a situation
- **Initial State**
- **Goal State**



- **Search Algorithm**
  Finds a way to get from **initial state** to **goal state** by systematically searching through the **state space**.

State Space: All possible states of the problem

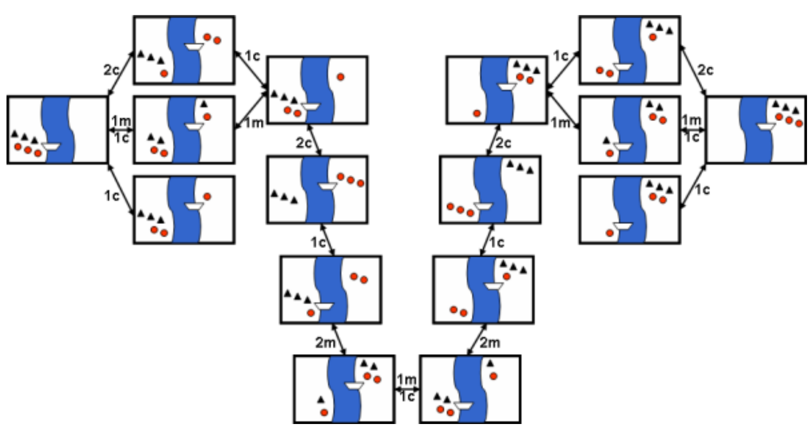
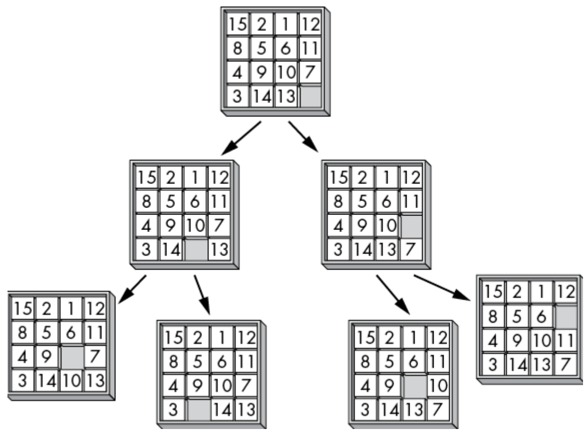


Image: https://www.aiai.ed.ac.uk/~gwickler/missionaries.html

State Space: 15-Puzzle

- Aka **Search Tree**

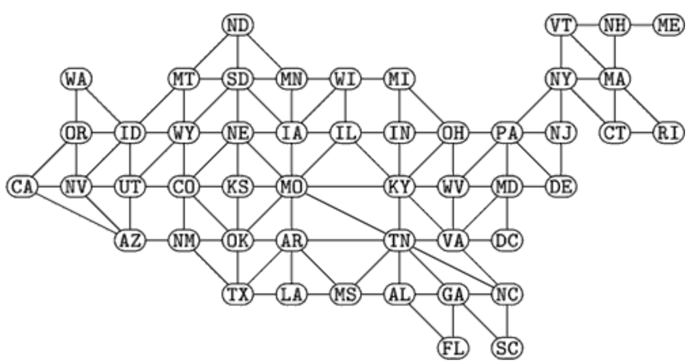


16! = 2×10^{13} different states
= 20,000,000,000,000!!

5

5

State Space: US States



Does not include Alaska & Hawaii
Has 49 vertices
107 edges

6

6

State Space: Towers of Hanoi

- Search Algorithm: Searches through the search space systematically to find a path to the goal.

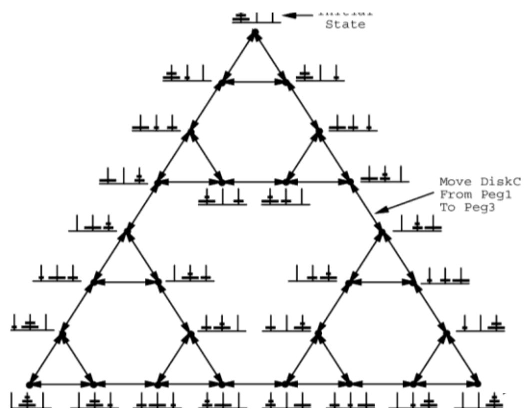


Image: https://www.researchgate.net/publication/2453845_Abstracting_the_Tower_of_Hanoi/figures?lo=1

7

7

Search Algorithms

- **Blind Search**
  Brute force algorithms that can find a path to the goal if one exists.
  But no guarantee that it is **optimal**.
  Examples: **Depth-first search**, **breadth-first search**.

- **Informed Search**
  Guarantees that the path to goal is optimal.
  Examples: **Uniform-Cost Search**, **Greedy Best-first**, **A***, etc.

8

8

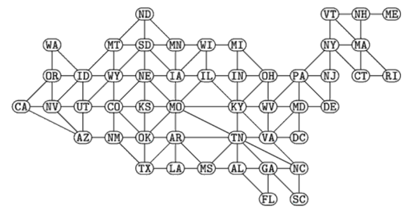
A Generic Blind Search Algorithm

- Uses a data structure, called ***frontier*** (a stack or a queue), to keep track of partially explored paths from initial state. Also uses a data structure (a set), ***explored*** to keep track of states/nodes already explored.

frontier ← a partial path containing the *start node*

explored ← { }

repeat
  p ← remove a partial path from the *frontier*
  if *p* ends in a goal node/state **return** the path *p* as answer
    neighbors ← neighbors of last node (*i*) in *p* that are not in explored
    explored ← last node (*i*) in *p*
    for each node *n* in *neighbors*
      q ← extend *p* to that neighbor, *n*
      frontier ← add *q*
until *frontier* is empty
return that there are no paths from initial state to goal state



Initial State: CA
Goal State: PA
Partial Path: CA-OR-ID-MT
Neighbors of MT: ID, ND, SD, WY

A Generic Blind Search Algorithm

- Uses a data structure, called *frontier* (a stack or a queue), to keep track of partially explored paths from initial state. Also uses a data structure, *explored* to keep track of states/nodes already explored.

frontier ← a partial path containing the *start node*

explored ← { }

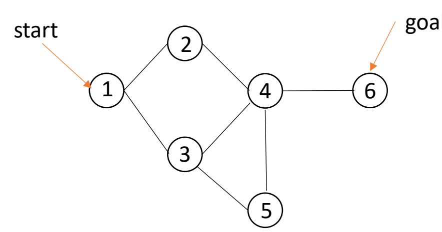
repeat
  p ← remove a partial path from the *frontier*
  if *p* ends in a goal node/state return the path *p* as answer
    neighbors ← neighbors of last node (*i*) in *p* that are not in explored
    explored ← last node (*i*) in *p*
    for each node *n* in *neighbors*
      q ← extend *p* to that neighbor, *n*
      frontier ← add *q*
until *frontier* is empty
return that there are no paths from initial state to goal state

Depth-first Search: *frontier* is a stack
Breadth-first Search: *frontier* is a queue

Trace on board

- Breadth-first Search (frontier is a queue)
- Depth-first Search (frontier is a stack)

A Toy Example



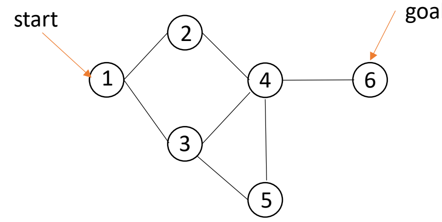
frontier ← a partial path containing the *start node*

explored ← { }

repeat
  p ← remove a partial path from the *frontier*
  if *p* ends in a goal node/state return the path *p* as answer
    neighbors ← neighbors of last node (*i*) in *p* that are not in explored
    explored ← last node (*i*) in *p*
    for each node *n* in *neighbors*
      q ← extend *p* to that neighbor, *n*
      frontier ← add *q*
until *frontier* is empty

return that there are no paths from initial state to goal state

A Toy Example



frontier ← a partial path containing the *start node*

explored ← { }

repeat
p ← remove a partial path from the *frontier*

if *p* ends in a goal node/state return the path *p* as answer

neighbors ← neighbors of last node (*i*) in *p* that are not in explored

explored ← last node (*i*) in *p*

for each node *n* in *neighbors*

q ← extend *p* to that neighbor, *n*

frontier ← add *q*

until *frontier* is empty

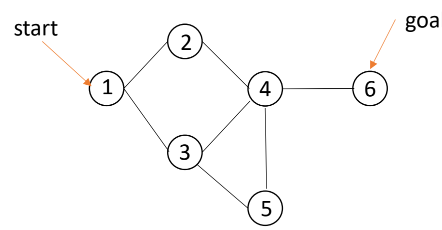
return that there are no paths from initial state to goal state

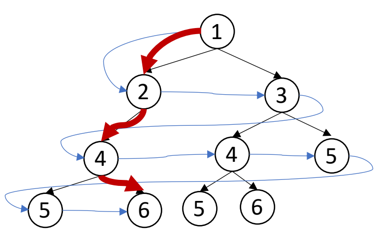
if n = goal
return q as answer

13

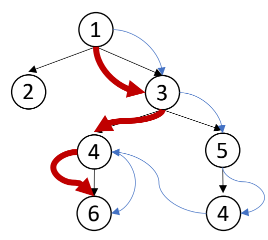
13

Search Trees





Breadth-first Search



Depth-first Search

14

14

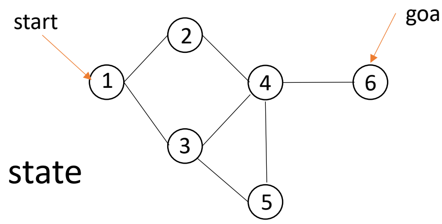
The Complexity of Search

- How long will it take for a blind search to find a path to goal if one exists?

  Two concepts:

  Branching Factor, *b*
  b is the number of successors/neighbors of a state

  Search Depth, *d*
  d is the depth at which the goal exists



15

15

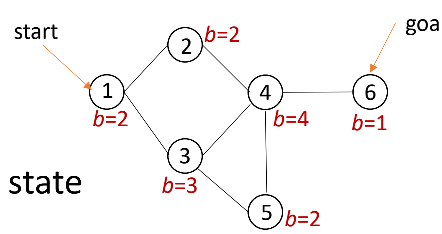
The Complexity of Search

- How long will it take for a blind search to find a path to goal if one exists?

  Two concepts:

  Branching Factor, *b*
  b is the number of successors/neighbors of a state

  Search Depth, *d*
  d is the depth at which the goal was found



Average branching factor = 14/6 = 2.3

16

16

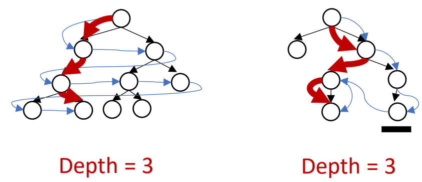
The Complexity of Search

- How long will it take for a blind search to find a path to goal if one exists?

  Two concepts:

  Branching Factor, *b*
  b is the number of successors/neighbors of a state

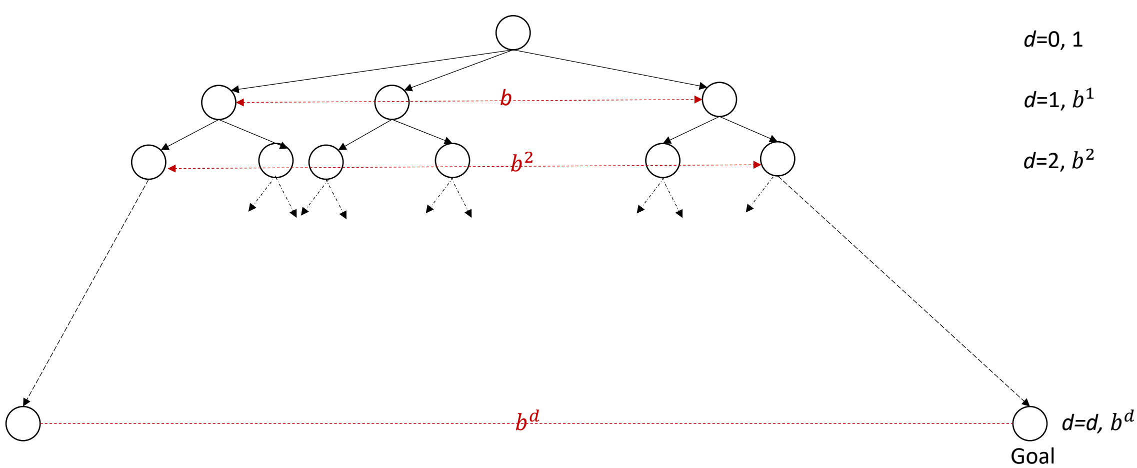
  Search Depth, *d*
  d is the depth at which the goal was found



17

17

In general, worst case



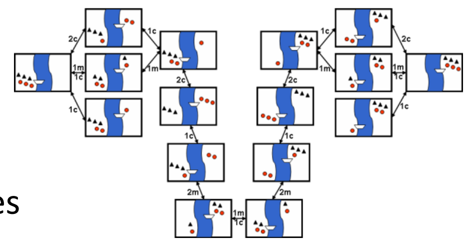
Worst case the algorithm will search b^d states/nodes. i.e. O(b^d)

18

18

M&C Puzzle

Average branching factor is ~1.4



For a solution length of 11,
a search algorithm will explore 1.4^{11} states

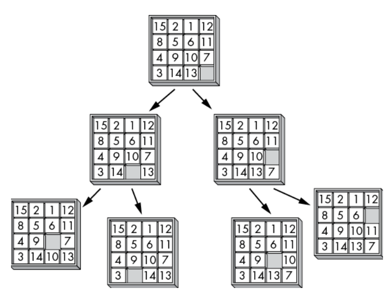
1.4^{11} = ~41

“Piece of cake!”

Image: https://www.aiai.ed.ac.uk/~gwickler/missionaries.html

15-Puzzle

- Average Branching Factor is ~3
- Average number of moves to a solution is ~50
- That is a search algorithm will need to explore 3^{50} states



3^{50} = 717,897,987,691,852,588,770,249

or ~7.1789799 x 10^{23}

15-Puzzle

- Average Branching Factor is ~3
- Average number of moves to a solution is ~50
- That is a search algorithm will need to explore 3^{50} states

  3^{50} = 717,897,987,691,852,588,770,249

  or ~7.1789799 x 10^{23}

Image: https://www.freecodecamp.org/news/combinatorics-handle-with-care-ed808b48e5dd/

21

21

Combinatorial Explosion/Complexity Barrier

- If search is a ubiquitous requirement in AI problems. How do we confront the complexity??
- One solution: use bigger, faster computers
- Another solution: Find better search algorithms
- Towards *informed* search algorithms

22

22

Informed Search Algorithms

- Try to use additional information available in the problem specs
  More efficient than blind searches

- Provide an **optimal solutio**n (if one exists)

- Examples of information:

  Solutions/Actions may have an associated cost:
  a measure of distance, number of moves, amount of time, $cost,…

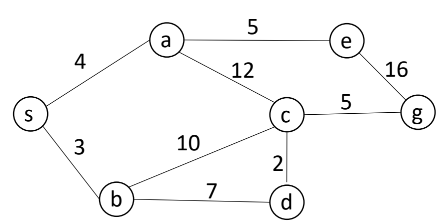
  May make use of **heuristic** measures
  estimate of remaining distance/cost/time (but not exact!)

23

23

Information

- Numbers on edges denote costs
  Could be time in min/hours
  Could be distance
  etc.



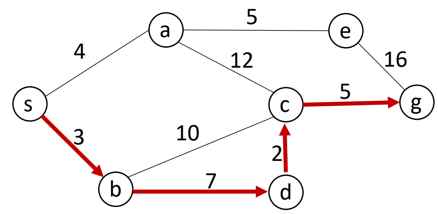
- What is optimal path from **s** to **g**?

24

24

Information

- Numbers on edges denote costs
  Could be time in min/hours
  Could be distance
  etc.



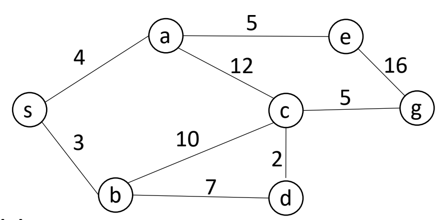
- What is optimal path from **s** to **g**?

Cost of optimal path is 17

- Define path cost function, **g(n)** as cost of path from start node to node, n
  Example:
  Cost of path g(s-b-c) = 13

Best-First Search *aka* Uniform Cost Search

Explores the most promising partial path based on **g(n)**

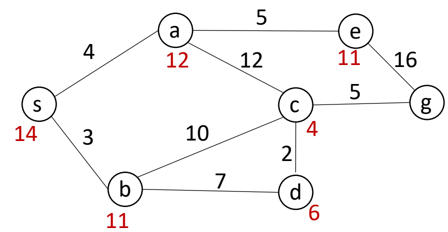


frontier ← a partial path containing the *start node*
explored ← { }
repeat
  p ← remove a partial path from the *frontier* **with the smallest g(n)**
  if *p* ends in a goal node/state return the path *p* as answer
    neighbors ← neighbors of last node (*i*) in *p* that are not in explored
    explored ← last node (*i*) in *p*
    for each node *n* in *neighbors*
      q ← extend *p* to that neighbor, *n*
      frontier ← add *q*
until *frontier* is empty
return that there are no paths from initial state to goal state

Trace on board…

More Information - Heuristics

- Numbers on edges denote costs
  Could be time in min/hours
  Could be distance
  etc.

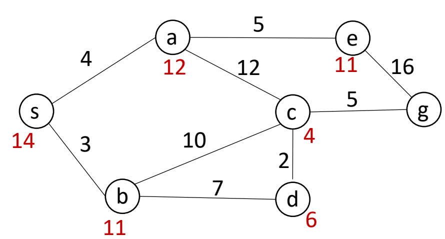


- Define cost function, **h(n)** as cost of path from a node to goal
  Example:
  Cost of path h(b) = 11

  h is a **heuristic**. An informal (but useful) estimate.

27

Greedy Best-First Search



Explores the most promising partial path based on **h(i)**

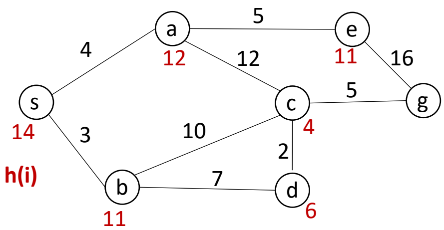
frontier ← a partial path containing the *start node*
explored ← { }
repeat
  p ← remove a partial path from the *frontier* **with the smallest h(i), *i*** is the last node in partial path
  if *p* ends in a goal node/state return the path *p* as answer
    neighbors ← neighbors of last node (*i*) in *p* that are not in explored
    explored ← last node (*i*) in *p*
    for each node *n* in *neighbors*
      q ← extend *p* to that neighbor, *n*
      frontier ← add *q*
until *frontier* is empty
return that there are no paths from initial state to goal state

Trace on board…

28

28

A*Search



Explores the most promising partial path based on total cost **f(i) = g(i) + h(i)**

frontier ← a partial path containing the *start node*
explored ← { }
repeat
  p ← remove a partial path from the *frontier* **with the smallest f(i),** ***i*** is the last node in partial path
  if *p* ends in a goal node/state return the path *p* as answer
    neighbors ← neighbors of last node (*i*) in *p* that are not in explored
    explored ← last node (*i*) in *p*
    for each node *n* in *neighbors*
      q ← extend *p* to that neighbor, *n*
      frontier ← add *q*
until *frontier* is empty
return that there are no paths from initial state to goal state

Trace on board…

29

29

More about A* And Heuristics

- A* is guaranteed to find the optimal path, if one exists
  i.e. A* is **complete**.
- The heuristic has to be **admissible** to guarantee optimal path.
  i.e. it has to be an **underestimate** of the actual cost.

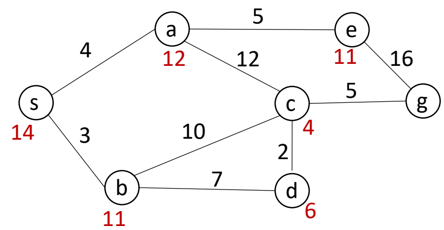
30

30

More about A* And Heuristics

- A* is guaranteed to find the optimal path, if one exists
  i.e. A* is **complete**.

- The heuristic must be **admissible** to guarantee optimal path.
  i.e. it has to be an **underestimate** of the actual cost.



Applications of A*

- Robotics
  Path planning

- Problem Solving
  Puzzles

- GPS Navigation

- And many many more!

Key Ideas

- Problem Solving as search
- Combating combinatorial explosion
- Using heuristics
- Many applications

Vocabulary

Problem Solving as Search
State
Initial State
Goal State
Search Algorithms
State Space
Search Trees
Branching Factor
Search Depth
Search Complexity
Combinatorial Explosion
Complexity Barrier

Search Algorithms
- Blind Search
  - DFS
  - BFS
- Informed Search
  - Uniform-Cost (Best-First)
  - Greedy Best-First
  - A*

Cost Function (g)
Heuristic Function (h)
Total Cost Function(f)

References

- M. Wooldridge: *A Brief History of Artificial Intelligence*. Flatiron Books, 2020.
- Nils Nilsson, *Artificial Intelligence: A New Synthesis*, Morgan Kauffman, 1998.

35