Tufts
UNIVERSITY

UNINFORMED SEARCH

Fabrizio Santini | COMP 131

VERSION 1.3

TODAY ON AI

- Unplanning vs. planning agents
- Solve problems by searching
- Depth-first search
- Breath-first search
- Uniform-cost search
- Questions?

Unplanning vs. planning agents

SECTION 01

Unplanning agents do not consider the future consequences of their actions, but see the world as a single percept:

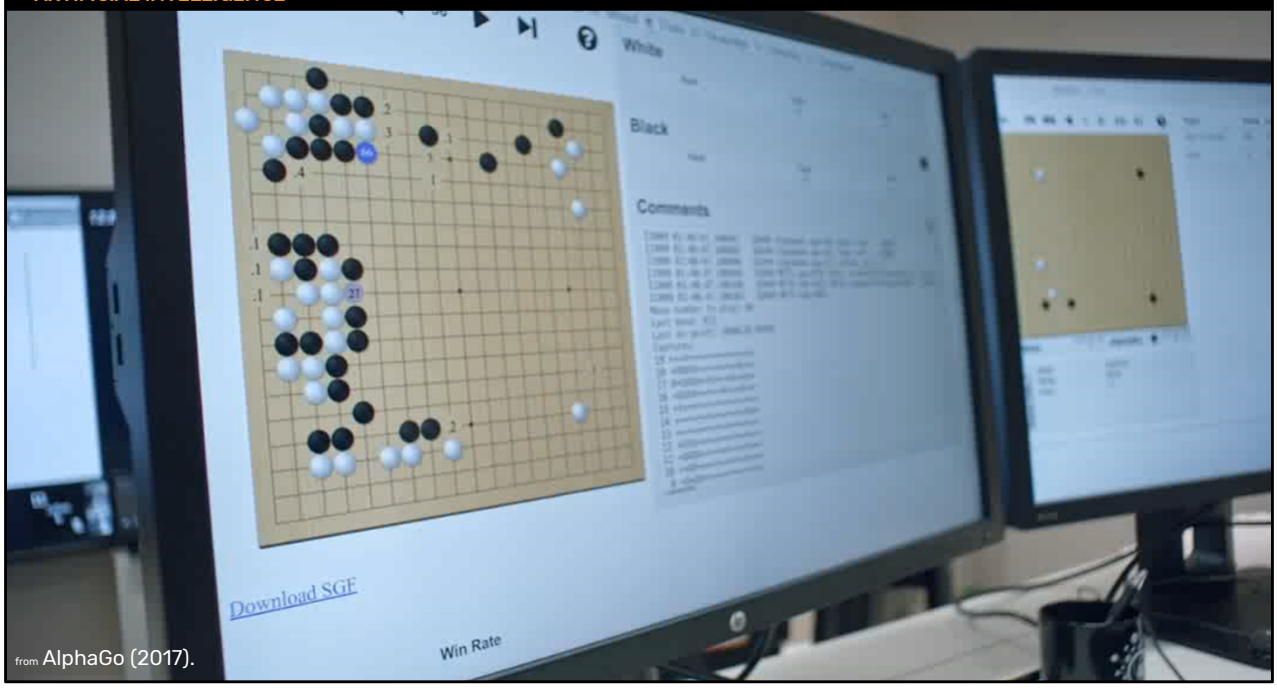
- May have a **model of the current state** of the world
- **Choose actions** based solely on the current percept

Planning agents do consider the future consequences of their actions, and see the world as it would be:

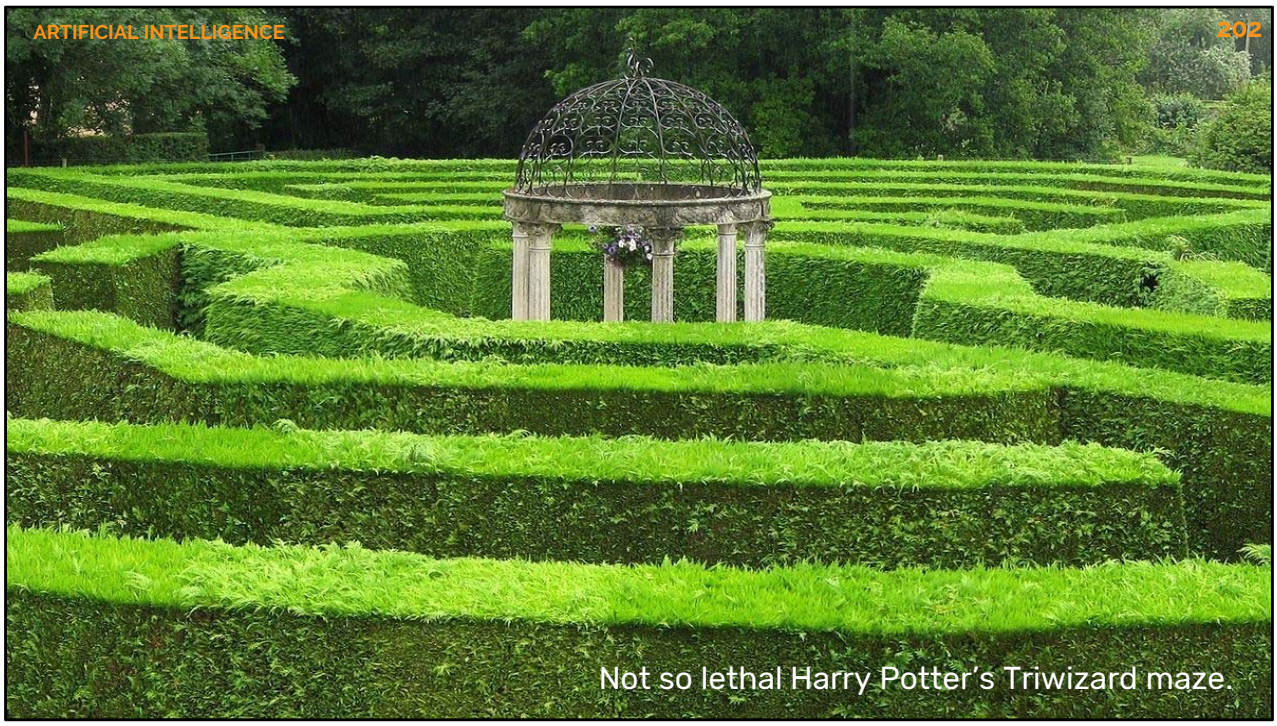
- Must have a **model of how the world evolves** in response to actions
- Must **formulate** a goal
- **Choose actions** based on hypothetical consequences of those actions

Solve problems by searching

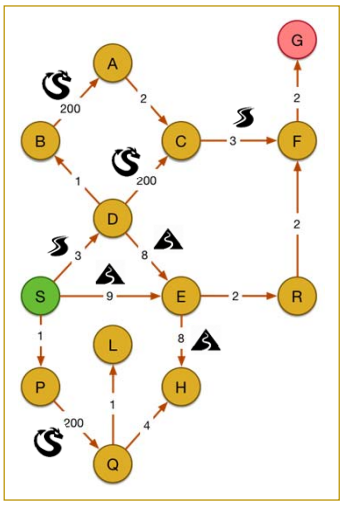
SECTION 02



from AlphaGo (2017).



Not so lethal Harry Potter's Triwizard maze.



- **STATES:** A, B, C, D, E, F, G, H, P, Q, R, S
- **INITIAL STATE:** State S
- **SUCCESSOR FUNCTION:** Go to the next node with cost specified
- **GOAL TEST:** Did we reach G?

How do we find a rational solution?

A **searching problem** is defined by five components:

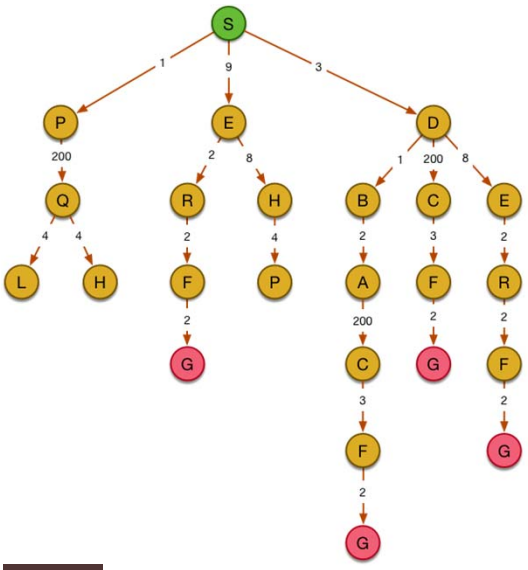
- The **initial state** that the agent starts in
- A description of the **possible actions** available
- **Successor function** or transition model: the effect of each action
- The **goal test** determines the end of the search
- A **path cost function** that assigns a numeric cost to the paths

- A **state space graph** is a mathematical representation of a portion (or all) the states of search problem:
  - Nodes are world configurations
  - Edges represent actions taken

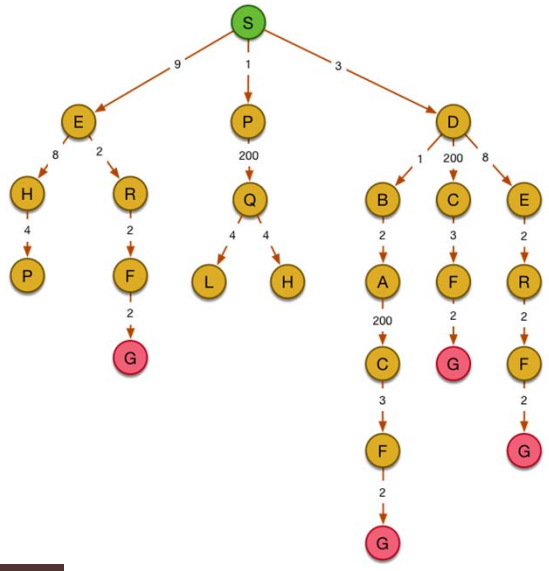
- A **search tree** is an instantiation of the state space graph with the start state as a root

Slide 10

A2 Revisit this slide: The definitions of state space, state space graph, and search tree are confusing.
Author, 6/15/2017



SOLUTION 1



SOLUTION 2

Key concepts:

- **Frontier**: a set of partial plans under consideration
- **Expansion**: expand potential plans
- **Exploration strategy**: explore as few nodes as possible

```
1  function Tree-search(PROBLEM) return SOLUTION, or FAILURE
2  initialize the frontier using the initial state of PROBLEM
3  loop do
4    if the frontier is empty then
5        return FAILURE
6    choose a leaf node according to a strategy
7    remove it from the frontier
8    if the node contains a goal state then
9      return the corresponding SOLUTION
10   expand the chosen node
11   add the resulting children to the frontier
12 end
```

- In a search tree the main question is: **which frontier nodes to explore**?

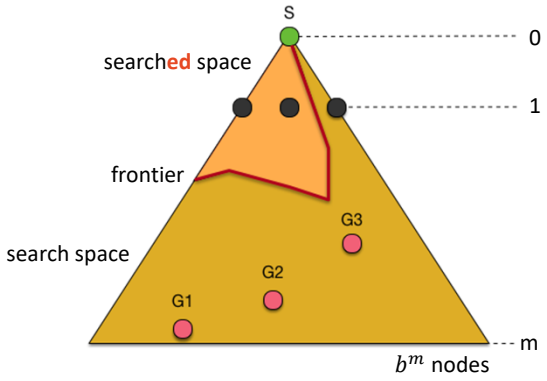
Every time we evaluate a search algorithm with want to ask these questions:

- **Node expansion**: How does the strategy select the next nodes?
- **Completeness**: Does it guarantee to find a solution if one exists?
- **Optimality**: Does it guarantee to find the least cost path?
- **Time complexity**: How long does it take to run?
- **Space complexity**: How much space does it take to run?

b: branching factor
m: maximum depth
s_i: solutions

Time and space complexity

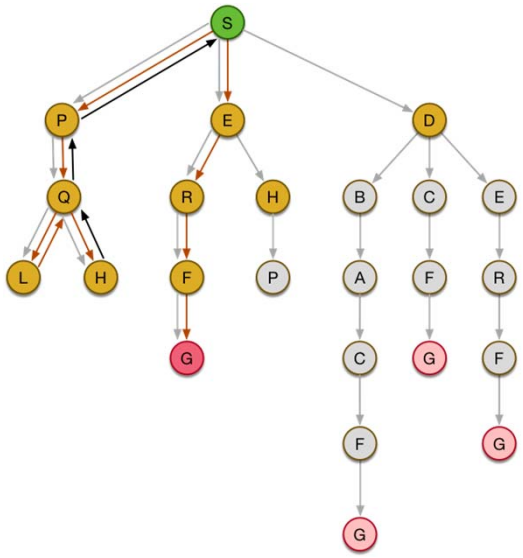
$$1 + b + b^2 + \cdots + b^m = O(b^m)$$



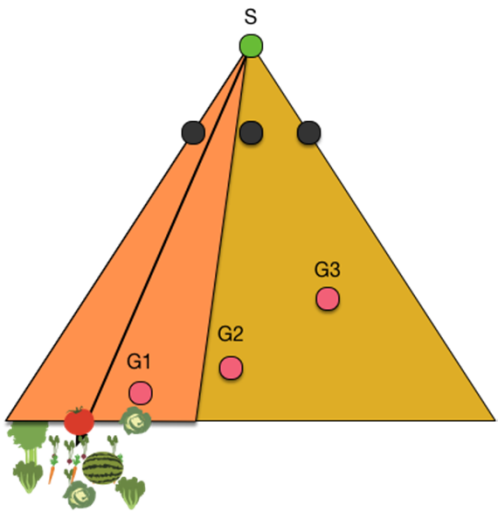
Depth-first search

SECTION 03

- **Node expansion**: Expand the deepest node first
- **Implementation**: Use a Last-Input-First-Output (LIFO) stack



- **Completeness**: Only if m is finite
- **Optimality**: No as it finds the left-most solution regardless of the cost
- **Time complexity**: $O(b^m)$
- **Space complexity**: $O(b^m)$
  where m is the maximum depth of the search space



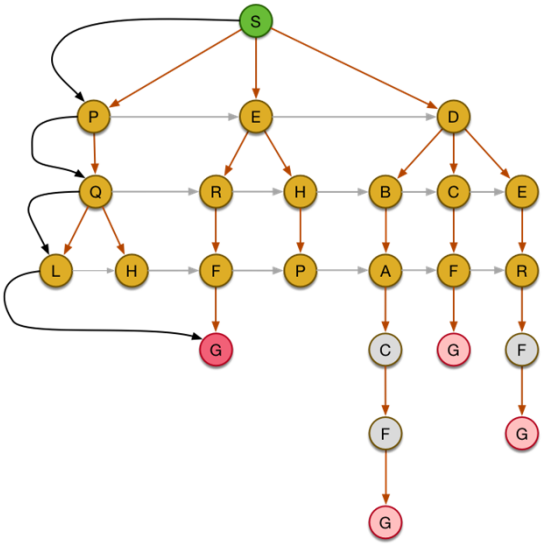
GOOD Uhm… nothing really

BAD If the tree depth can infinitely expand, we might never reach the goal

Breadth-first search

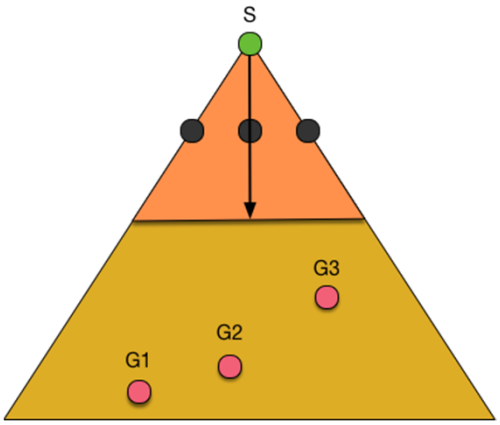
SECTION 04

- **Node expansion**: Expand the shallowest node first
- **Implementation**: Use a First-Input-First-Output (FIFO) queue



- **Completeness**: Only if b is finite
- **Optimality**: Yes, if the costs are equal
- **Time complexity**: $O(b^g)$
- **Space complexity**: $O(b^g)$

where g is the depth to which a goal was found



GOOD In certain conditions, BFS is complete and optimal

BAD BFS might take too much memory and time to find a solution. Also the solution might not be optimal

Depth	Nodes	Time	Memory
2	64	80 microseconds	12.5 kilobytes
4	4,096	5.1 milliseconds	800 kilobytes
6	262,144	327 milliseconds	50 megabytes
8	16,777,216	20 seconds	3.1 gigabytes
10	1,073,741,824	22 minutes	200 gigabytes
14	4,398,046,511,104	63 days	800 terabytes
16	281,474,976,710,656	11 years	50 petabytes

Time and memory requirements for BFS. The numbers shown assume branching factor $b = 8$; 800K nodes/second; 200 bytes

Uniform-cost search (Dijkstra search)

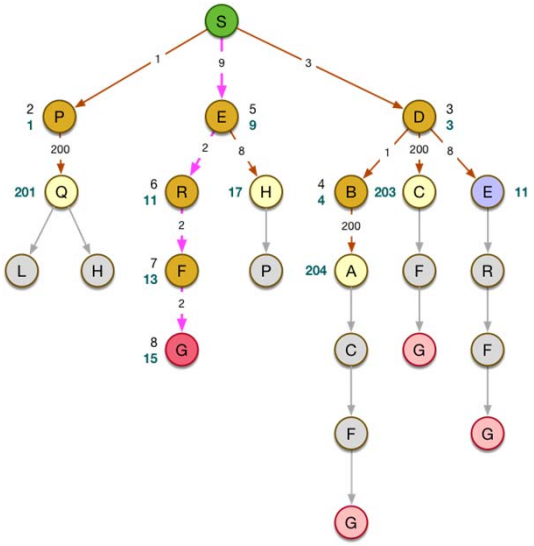
SECTION 05

This strategy introduces the idea of **graph search** (as opposed to tree search): there is no need to visit again states that have been visited before.

While exploring the frontier, states being expanded are marked as visited.

```
1  function Uniform-cost-search(PROBLEM) return SOLUTION, or FAILURE
2  initialize the frontier using the initial state of PROBLEM
3  loop do
4    if the frontier is empty then
5      return FAILURE
6    pop node from frontier with min_c g(c)
7    if the node contains a goal state then
8      return the corresponding SOLUTION
9    expand the chosen node
10   for each child
11     if child is not in the frontier or visited then
12       insert child in frontier
13     else if child is in frontier with higher cost then
14       replace child in frontier with child
15     end
```

- **Node expansion**: Expand the cheapest node first
- **Implementation**: Use a Priority queue, where the priority is the cumulative (path) cost of the node



P 2 1 VISITED
L NOT VISITED
E 11 PRUNED
G 8 15 GOAL
Q 201 FRONTIER
G GOAL NOT VISITED

Slide 23

A1 Author, 5/20/2017

PRIORITY QUEUE

A priority queue is an abstract data type similar to a queue. Each element is associated with a priority.

Any element with the highest (or lowest) priority is served before any other element with a lower (or higher) priority.

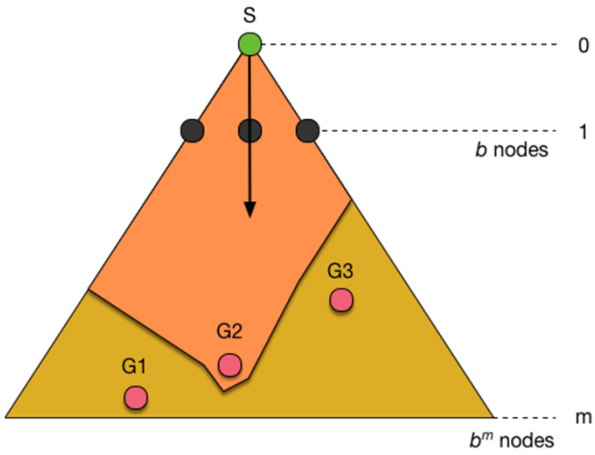
Operation	Binary	Binomial	Fibonacci
find-min	$O(1)$	$O(\log n)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$
insert	$O(\log n)$	$O(1)$	$O(1)$
decrease-key	$O(\log n)$	$O(\log n)$	$O(1)$
merge	$O(n)$	$O(\log n)$	$O(1)$

If the solution costs C^* and the arcs cost at least ε, then the worst-case depth is $\frac{C^*}{\varepsilon}$

- **Completeness**: Yes, only if ε is positive
- **Optimality**: Yes
- **Time complexity**: $O\left(b^{1+\left\lfloor\frac{C^*}{\varepsilon}\right\rfloor}\right)$
- **Space complexity**: $O\left(b^{1+\left\lfloor\frac{C^*}{\varepsilon}\right\rfloor}\right)$

GOOD UCS is complete and optimal

BAD UCS explores in every direction without any information about the goal location



QUESTIONS ?