CSC 480 Artificial Intelligence Searching Algorithms

Brian Kwong

2025-04-17

Table of contents

1 Announcements

- Quiz 1 is due **tomorrow night** (Friday April 18th 2025).
- The Project Proposal is also due **tomorrow night** (Deadline has been extended 1 day).
  - This week submission will be a draft and ungraded. You will receive feedback from Professor Canaan.

– You then will have the opportunity to revise your project proposal whose final draft will be due **next Friday** (April 25th 2025). This will be **graded**.

2 Search Algorithms

What makes search informed vs. one that is uninformed?
Informed search has a **heuristics**.

Heuristics:
An estimate of how close you are to an end state/goal.

2.0.1 Uninformed Search

In the previous lecture, we have discussed two common tree search algorithms:

1. Depth First Search:
   All child nodes of a particular branch are explored before moving onto the next branch.
2. Breadth First Search:
   All branches are explored at equal depths.

> **Note**
>
> The primary advantage of DFS is its **low memory** profile as only one branch is needed to be stored in the system at a time, but the derived solution may not be **optimal** and it's not complete due to loops or an infinite number of branches.

For example, in an infinite world game[1] like `Minecraft`, where the world is generated continually, the search becomes computationally infinite.

2.0.1.1 Depth Limited Depth First Search

- Depth Limited Depth First Search:
  Keeps track of the current depth and stops once a maximum depth has been reached.

[1] Technically, a Minecraft world is not infinite as it is limited by Java's `int_32` size which is 2^{31} in each direction but if you have been playing Minecraft for so long you probably should go touch some grass... There are also some weird phenomenons such as the far lands which occur before you reach the theoretical maximum.

```
depth_limited_dfs(root, max_d, d = 0)
  while d < max_d
    for child in root.children:
      depth_limited_dfs(child, max_d, d+1)

depth_limited_dfs(root, max_d)
```

2.0.1.1.1 Advantages of Depth Limited DFS

- Optimal and complete if some solution exists up to n (where n is the maximum search depth).
- Many algorithms use depth limited DFS, such as the DeepBlue chess algorithm.

2.0.1.1.2 Overhead of DFS & Depth Limited DFS

- Overhead of DFS comes from the fact you **continually** revisit the same nodes (parent nodes) as you explore in a bottom-up approach.
- There are also often similar paths that DFS will explore in depth, which causes higher overhead.

> **Note**
>
> - The average **added** overhead is $\frac{1}{avg_branch_factor}$.
> - The total overhead is $1 + \frac{1}{avg_branch_factor}$.
>
> As the branching factor increases, the approximation gets more exact (i.e., sum of geometric sequences).

- The total number of nodes at depth d is b^d. In BFS, all nodes at level n have to be stored, while DFS only needs to remember its parent.
- Check out the spreadsheet for a more in-depth playground on different overhead factors.

DFS and BFS work great if there are no costs or weights to the graph, but realistically there are.

2.1 Uniform Search / Dijkstra's

A priority queue (referred to as the `fringe`) keeps track of the current cost up to that node. We continue to search by enqueuing unexplored nodes into the fringe and dequeuing the one with the **lowest** cost. - We hit a win condition. - We exhaust the heap and do not hit a win condition.

```
def search(root):
  fringe.enqueue(root)
  while not fringe.empty():
    node = fringe.dequeue()
    if(isWin(node)):
      return node
    else:
      for child in node.children:
        fringe.enqueue(child)
  return None
```

You check if the current node is a win condition when you pop off the fringe.

We know we found the shortest path since we **already** have searched all paths that cost less than the current cost `C`.

> **Note**
>
> - DFS is a special case of Dijkstra where the weight between all nodes is constant.
> - Dijkstra is a special case of A* with no heuristics.

Example of Uninformed Search on Basic Graph:

> **Tip**
>
> To view the GIF representations of the search algorithm visit the HTML version of the notes

> **Note**
>
> **Node Color Labels**
>
> - Green:
> Searched and Completed.
> - Yellow:
> Searching.
> - Red:
> In the fringe to be explored.

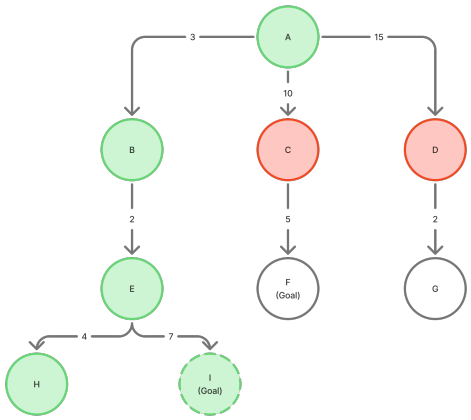

Figure 1: Example of Uniform Search / Dijkstra's Algorithm

2.1.1 Bidirectional Search

Bidirectional search:
A search technique where two searches are initialized, one from the beginning and one from the end.

- It on average cuts the time by the $\sqrt{x}$ where x is the cost of regular search.

Bidirectional search should be used if the following conditions are met:

1. Goal state is known ahead of time and unique.
2. States are easily checked for equality (know when the two states meet up).
3. Transition function must be easily invertible.
4. (Optional) Are able to search concurrently (Two different cores) or you would need to context switch between the two searches.

> **Note**
>
> You need to store the "frontier" to know if you have a convergence point, therefore bidirectional search is done with BFS.
>
> - You can cache the preferred path for better efficiency.

The following is a chart outlining different:

Uninformed Search, Their Advantages, and Subsequent Costs:

Method	Complete?	Optimal?	Space Complexity	Time complexity	Data structure for fringe	Notes
BFS	Yes (finite b)	Yes (if costs are constant)	Exponential	Exponential	FIFO (queue)	Complete, Optimal, bad memory complexity.
Uniform-Cost	Yes	Yes (even with variable costs)	Exponential	Exponential	Priority Queue	Handles different costs.
DFS	No	No	Linear ~ b*m	Exponential	LIFO (stack)	Low memory requirement, but fewer guarantees than BFS.
Backtracking DFS	No	No	Linear ~ m	Exponential	Only 1 node in fringe (but may use stack as part of backtracking implementation)	Even lower memory requirement, but a bit more complicated than DFS.
Depth-Limited	No	No	Linear ~ bl	Exponential	Stack (variation of BFS)	Alleviates DFS problems with infinite spaces and loops.

Method	Complete?	Optimal?	Space Complexity	Time complexity	Data structure for fringe	Notes
Iterative Deepening	Yes (if reachable with available memory)	Yes (if reachable with available memory)	Linear ~ bd	Exponential	Stack (variation of BFS)	Combines low memory of DFS with completeness and optimality of BFS. Re-explores shallow nodes many times (not too bad).
Bi-directional	Yes (if using BFS)	Yes (if using BFS)	Exponential (square root of complexity with BFS)	Exponential (square root of complexity with most other strategies)	Depends on which strategy each side is using	Needs to generate predecessors, needs to enumerate goals, at least one side needs to store all frontier nodes.

2.2 Informed Search

Informed Search:
Is the process of adding a heuristic or estimate on how close we are (cost of reaching) an end state.

In general, informed search performs better than uninformed search.

> **Note**
>
> In the following section, assume the heuristics are given.

> **Warning**
>
> Heuristics do not solve all problems. For large datasets such as chess, heuristics still do not provide the capability to search the entire possible game space.

`F(n)` is a function that **estimates**[2] how useful a particular node is.

- `N`:
  A node in the queue ready to be explored.
- `F(n)`:
  An estimate of how good it is to expand the node.
- `G(n)`:
  The true cost from the root to the current node.
- `H(n)`:
  An estimate of the cost of the best path from n -> goal node.

Uninformed search is a variation of informed search where `F(n)` = `G(n)`.

2.2.1 Greedy Algorithm

Greedy Algorithm:
A search algorithm that bases its decision purely on heuristics, ignoring all costs that are incurred to get to that node (i.e., `G(n)` = `F(n)`).

> **Warning**
>
> Greedy is incomplete just like DFS as it gets stuck on loops or infinitely large graphs.

```
# Fringe is sorted by node.heuristic in increasing order
# ======== Note =============
# The "win condition" heuristic is 0
def greedy(root):
  fringe.enqueue(root)
  while not fringe.empty():
    node = fringe.dequeue()
    if(isWin(node)):
```

[2] How do you know what is best though? You **don't**, so you estimate by a heuristic.

```
      return node
    else:
      for child in node.children:
        fringe.enqueue(child)
return None
```

Tip

To view the GIF representations of the search algorithm visit the HTML version of the notes

Note

Node Color Labels

- Green:
  Searched and Completed.
- Yellow:
  Searching.
- Red:
  In the fringe to be explored.

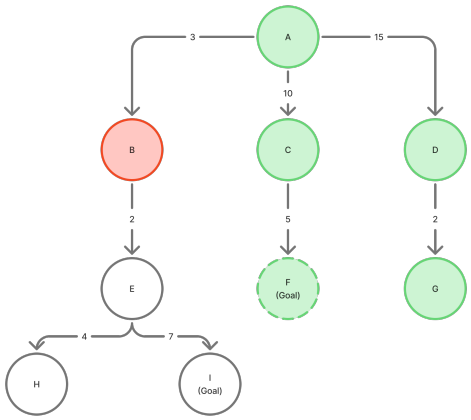

Figure 2: Example of Informed Greedy Algorithm

2.2.1.1 When Should You Use Greedy?

1. Your heuristics are really good.
2. Faster than `A*` if a solution can be found in a similar number of steps.
3. Resource-limited like in embedded systems; less resource-intensive than `A*`.
4. If the cost of search is more important than the optimality of a search.
5. The cost to the current node is sunk in or backtracking is costly/unnecessary.

NOTE: Even though a solution is found, it's not optimal as greedy stops as soon as a solution is found.

2.2.2 A*

A Star (`*`):
A search algorithm that combines the advantages and techniques of both greedy and Dijkstra algorithms. It's commonly used in industry for pathing, mapping, and gaming applications. `F(n)` = `G(n)` + `H(n)`.

> - `G(n)`:
>   The true cost from the root to the current node.
> - `H(n)`:
>   A heuristic function that estimates how far away the current position is from the goal.

```
# Fringe is sorted by node.value in increasing order
# ======== Note =============
# Root's cost is 0
# The "win condition" heuristic is 0
def aStar(root):
  fringe.enqueue(root)
  while not fringe.empty():
    node = fringe.dequeue()
    if(isWin(node)):
      return node
    else:
      for child in node.children:
        child.value = node.cost + child.heuristic
        fringe.enqueue(child)
  return None
```

Tip

To view the GIF representations of the search algorithm visit the HTML version of the notes

Note

Node Color Labels

- Green:
  Searched and Completed.
- Yellow:
  Searching.
- Red:
  In the fringe to be explored.

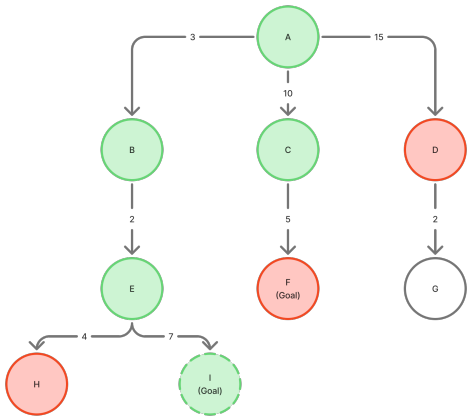

Figure 3: Example of Informed A Star Algorithm

When a node is enqueued to the fringe, its `f(n)` is calculated and placed as metadata of the node.

Even though a win condition can be seen in the queue, unlike greedy, `A*` does not pick it immediately if there are nodes with lower `F(n)` values.

However, we can eliminate all items in the queue that are greater than `F(n)` of that node since we know at maximum we know a solution at maximum would be `F(n)` of that node.

2.2.2.1 Choosing Heuristics

Caution

When choosing heuristics, you should not choose heuristics that are an overestimate of the real cost as it creates delusions.

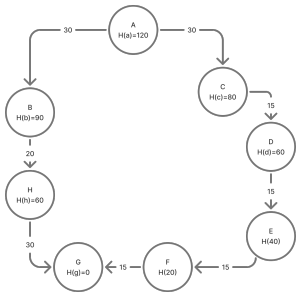

Figure 4: The following example shows an example of an **overestimate** of heuristics which would make A* waste time discovering a non-optimal route.

Tip

A common way to choose a heuristic is to pick the **optimal** best-case scenario (i.e., a world where you have no challenges/obstacles/rules to prevent you from reaching your goal).

2.2.2.1.1 Common Heuristics for A*

- Euclidean Distance:
  The diagonal or crow distance between any two points on a 2/3D grid, calculated by the distance formula $\mathrm{d} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$.
- Manhattan Distance:
  The total distance between two points on a 2/3D grid where valid movement can only be in one axis at a time $\mathrm{d} = |(x_2 - x_1)| + |(y_2 - y_1)| + |(z_2 - z_1)|$.

In general, Manhattan distance is a better distance heuristic (for most situations) modeling realistic pathing since it's rare that you can move in two (or more) directions at once.

2.2.2.2 Admissible Heuristics

- Admissible Heuristics:
  A heuristic is admissible if it underestimates or equals the true cost from the current node to the end node.

> Tip
>
> If you have multiple admissible heuristics from A –> B, then the **maximum** of those heuristics should be used as it most closely mimics the real cost from A –> B.

2.2.2.2.1 Example with 8 Puzzle

8 Puzzle is a sliding board puzzle where there are 8 numbered pieces on a `3 x 3` board. The goal is to arrange the puzzle pieces in a specific order such as:

5 7 6 4 2 1 3 8	1 2 3 4 5 6 7 8
Original 8 Puzzle Ex	*Solved 8 Puzzle Ex*

Two common heuristics used in 8 puzzle:

1. Number of misplaced pieces.
2. The Manhattan distance between the current position and the correct position.

In this scenario, the Manhattan distance would be a more realistic and therefore dominant heuristic as in the game only a piece can only be moved to an adjacent cell that's open. The Manhattan distance would represent the number of attempts in the **ideal** configuration played perfectly.