Class 4 - Greedy Heuristics and Hill Climbing

Guiding Genetic Algorithms with Language Models for Combinatorial Optimization

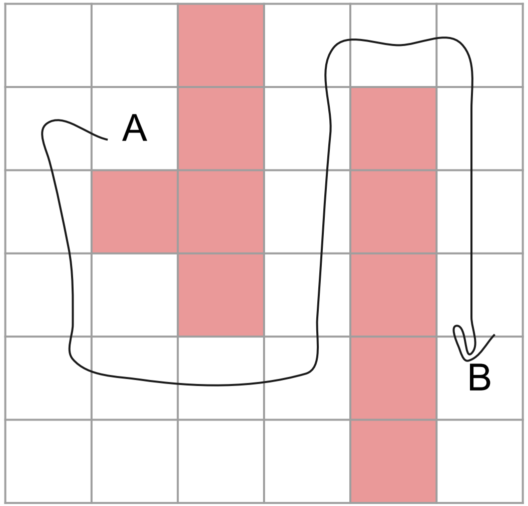
MISE Research Program - July 2025

Pedro Paredes

Informed Search

Recall: Finding a Path on a Grid

Suppose you have a grid and you want to find a ***shortest*** path from point A to point B avoiding obstacles



DFS Solution

```python
def dfs_shortest_path(r, c, goal, path_len, visited):
    if (r, c) == goal:
        return path_len

    # explore four directions
    bs = 100000
    for dr, dc in ((1,0), (-1,0), (0,1), (0,-1)):
        nr, nc = r + dr, c + dc
        inside = 0 <= nr < len(grid) and 0 <= nc < len(grid[0])
        if inside and grid[nr][nc] == 0 and (nr, nc) not in visited:
            visited.add((nr, nc))
            bs = min(bs, dfs_shortest_path(nr, nc, goal, path_len + 1, visited))
            visited.remove((nr, nc))
    return bs
```

Informed Search - Dijkstra’s Algorithm

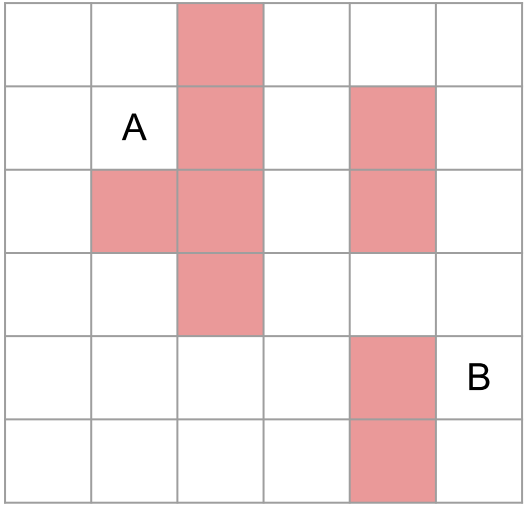
The simple DFS solution is pretty inefficient since we repeatedly search the same paths and we search paths that are not very promising

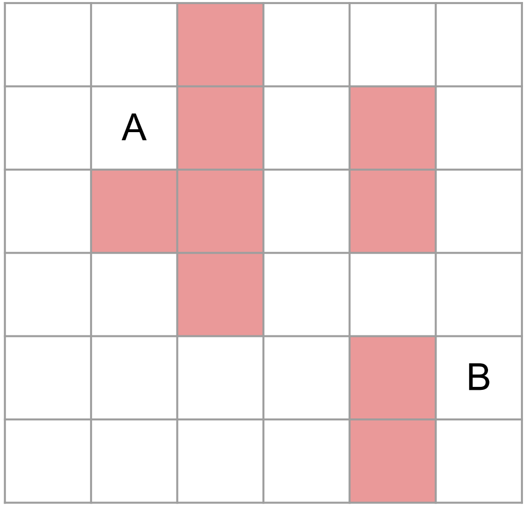
We can make our search more efficient using ***informed search*** (also known as ***Dijkstra’s Algorithm***):

- Instead of going depth-first, let’s look at more promising paths first, i.e., shorter
- Instead of keeping track of visited cells, we will keep track of the shortest path to get to each of them. If we visit a cell and the current distance is greater than or equal to the previous best, ignore this path.

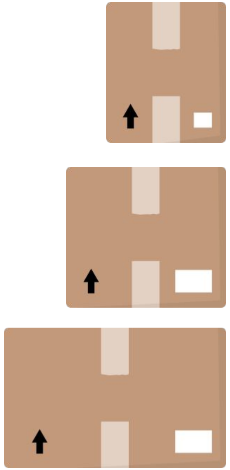
To prioritize shorter paths first, we will use a data structure known as a ***heap***, which lets us find the minimum element in it in O(log n) time.

```python
import heapq

def dijkstra_shortest_path(start, goal):
    rows, cols = len(grid), len(grid[0])
    dist = {start: 0}
    pq = [(0, start[0], start[1])]

    while pq:
        d, r, c = heapq.heappop(pq)
        if (r, c) == goal:
            return d
        if d != dist[(r, c)]:
            continue

        for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
            nr, nc = r + dr, c + dc
            inside = 0 <= nr < rows and 0 <= nc < cols
            if inside and grid[nr][nc] == 0:
                nd = d + 1
                if nd < dist.get((nr, nc), float('inf')):
                    dist[(nr, nc)] = nd
                    heapq.heappush(pq, (nd, nr, nc))
    return None
```


A
B


A
B

Packing a Truck



x 9

x 5

x 3

How many trucks do we need to carry all of the packages?

Truck size

Informed Search for Packing a Truck

We will use a similar approach as before, but we need to encode the problem:

- A state is a tuple *(i, bins)*, which means we have placed the first *i* packages and we have a list *bins* with the remaining capacity of each non-empty bin
- A transition is placing one more item, so we can either place it in any bin that has capacity for it, or we open a new bin
- The most promising states are the ones with less bins

Because the state space can be really large, we don’t keep track of the best way to get to each state

```python
import heapq

def dijkstra_binpack(weights, C):
    N = len(weights)
    start = (0, ())
    pq    = [(0, start)]

    while pq:
        g, (i, bins) = heapq.heappop(pq)
        if i == N:                          # all items placed
            return g                        # optimal #bins

        w = weights[i]                      # current item size

        # 1) try every existing bin that fits
        for idx, free in enumerate(bins):
            if free >= w:
                new_bins = list(bins)
                new_free = free - w
                if new_free == 0:
                    new_bins.pop(idx)   # bin now full
                else:
                    new_bins[idx] = new_free
                new_state = (i + 1, tuple(new_bins))
                heapq.heappush(pq, (g, new_state))

        # 2) open a fresh bin for this item
        if w > C:
            return None
        new_bins  = bins + (C - w,)
        new_state = (i + 1, new_bins)
        ng = g + 1
        heapq.heappush(pq, (ng, new_state))

    return None                             # should not reach here on valid input
```

Heuristics

Heuristics

A ***heuristic*** is a non-optimal approximation to a problem

Often we use intuition about a problem to come up with heuristics

We can use heuristics in several ways:

- To pick which states to explore first
- To discard non promising solutions

A* Algorithm

The A* algorithm is an algorithm that combines heuristics with informed search

A* Algorithm for Bin Packing

See code demo

Hill Climbing

Hill Climbing

Hill climbing is a method to find a solution to a problem by always moving in the direction of a "better" solution

- Start from a random state s
- Move to a neighbor with better score (break ties randomly)
- If we get stuck and didn't find a solution, restart

A neighbor of a state is a state obtained by a local change: e.g., moving one queen in the n-queens example

Hill Climbing for N-Queens

See code demo for N-queens

What's next?

Next class monday

No assignment sheet, study this week's lectures and solve all remaining problems

Class 5: Introduction to Genetic Algorithms