Assignment 8 - Dungeon Generator

Thom Mott
CMPM146, University of California, Santa Cruz

ABSTRACT
This paper describes a procedural dungeon generation system that creates playable dungeon layouts using randomness and simple constraints using a backtracking approach. Dungeons are built by selecting doors and placing compatible room types while preventing overlap and enforcing size limits. The generator ensures there is always a valid path from the starting room to a target room, with additional dead-end rooms used for special spaces such as treasure rooms. The system produces consistent results through seeded randomness and generates varied dungeon layouts that are visually coherent and suitable for gameplay.

Repository Link: https://github.com/Spebby/Dungeon-Generator
Hash: 4c5af57

1 INTRODUCTION

Procedural content generation in games presents a compelling challenge, creating coherent playable spaces that are not only fun is a tough challenge. For this project, I developed a backtracking-based algorithm, as required by assignment spec, which can generate varied, interconnected room layouts, while maintaining performance suitable for real-time game integration. The system was originally designed for CMPM146 and later adapted for use in CMPM121, demonstrating its extensibility.

2 GENERATION ALGORITHM

2.1 Overview and Architectural Decisions

The generation system operates on a recursive backtracking approach. Dungeons are built incrementally by selecting doors from a frontier, and attempting to place compatible attaching rooms, picked from an "archetype". "Archetypes" are a categorisation for rooms based on their door configuration and size defined by designers at built time. Within archetypes, rooms are given individual weights and archetypes themselves are given weights for selection purposes. This abstraction is essential for scalability, as room variants increased dramatically for the CMPM121 version. The generation process remained tractable by selecting archetypes first, then choosing specific variants within those archetypes once the dungeon graph has been completed.

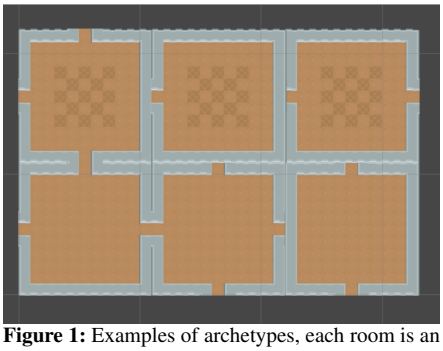

Figure 1: Examples of archetypes, each room is an example of a room in a given archetype.

The core data structures were designed to minimise allocation overhead, while maintaining flexibility required for backtracking. Custom wrapper types, such as `RentBuffer` were defined to keep allocations low. `RentBuffer` itself is a thin wrapper around `ArrayPool`, and was necessary for managing allocation pressure, which plagues recursive algorithms. Explanations of these types can be found in the performance optimisation section.

1. Occupancy tracking is implemented as a `RentBuffer<Vector2Int>`, containing all occupied grid cells, copied and extended on each recursive call.
2. A `RentBuffer<Door>` contained unprocessed doors and comprised the Frontier. It was rebuilt at each recursion level.

3. The room graph was constructed using a `Dictionary<Vector2Int, RoomNode>`, mapping grid coordinates to room nodes for pathfinding and special room placement. `RoomNodes` maintain a `HashSet` of connected neighbours, ensuring a bidirectional placement.

2.2 Room Selection and Placement

The generation process begins with a randomly selected starting room, placed at the origin, its doors comprise the initial frontier, and generation proceeds through recursive calls of a GenerateWithBacktracking method, defined as so. Room selection utilises a weighted random sampling approach, implemented through a custom "Weighted Bag" structure, which I discuss in detail in the performance optimisation section.

```
bool GenerateWithBacktracking(in RentBuffer<Vector2Int> occupancy,
                              in RentBuffer<Door> doors, int depth)
```

Each iteration follows this sequence:

1. If the frontier is empty, validate that the dungeon meets the minimum size and dimensional constraints. If constraints are satisfied, then generation is successful. Otherwise, backtrack.
2. A door is randomly selected from the frontier based on the current RNG state.
3. A valid room archetype is selected given the door's direction using an O(1) lookup from a precomputed table.
4. Candidate archetypes are sampled using the Weighted Bag structure.

```
Span<int> weights = stackalloc int[candidates.Length];
for (int i = 0; i < candidates.Length; i++) { weights[i] = candidates[i].Weight; }
using WeightedBag<RoomArchetype> bag = new(candidates, weights);
while (bag.TryNext(_rng, out RoomArchetype hopeful)) {
    // Attempt placement...
}
```

2.3 Constraint Validation

For each candidate archetype, several constraints are checked in order of computational cost. Overlap detection is the most expensive check, and is done first.

```
ReadOnlySpan<Vector2Int> occMap = hopeful.GetOccupancy(offset);
ReadOnlySpan<Vector2Int> baseOccupancy = occupancy.Buffer.AsSpan(0, occupancyCount);
if (!CanFit(occMap, baseOccupancy)) continue;
```

The `CanFit()` method performs a nested loop over the new room's occupancy cells and all existing occupied cells—an O(n × m) operation that dominates the runtime profile. Each `Vector2Int` comparison checks whether any cell in the prospective room overlaps with already-placed rooms. Then, a door budget check is performed, preventing runaway generation where the frontier grows faster than rooms are consumed, ensuring dungeons stay within the configured maximum size. Once a candidate passes all constraint checks, a new frontier is constructed for the next recursive call, and the new room's occupancy is merged with the existing occupancy map and room connectivity is recorded in the graph, and the next layer of recursion commences. During the unwind phase of recursion, the connections between rooms are recorded in the graph.

A limitation, and potential improvement, is path reconvergence. Once a path splinters, it will never connect to any other path, meaning every divergent path will lead to a dead end, and no loops exist. I had attempted to implement this by checking if a new door's match already existed in the frontier, but the overlap check rejects any placement where rooms touch, even if valid door-to-door connections exist. To properly implement this, the frontier would have to be pre-filtered to identify legitimate reconnection opportunities before performing the overlap check. This remains a priority for future refinement.

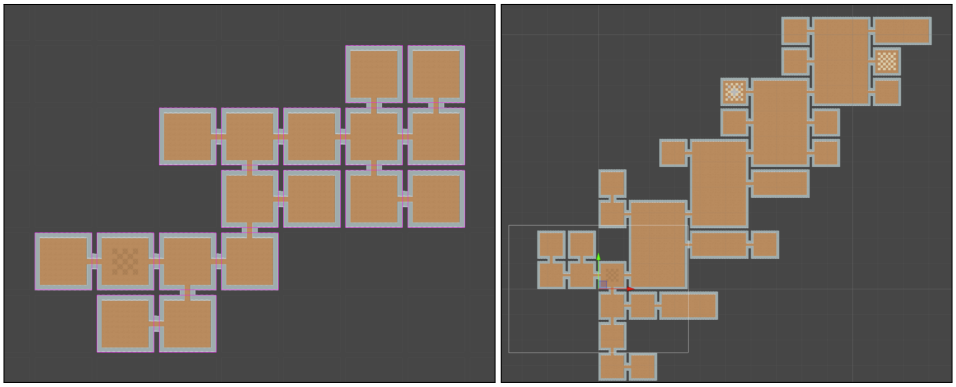

Figure 2. An example of generated output

Occasionally, generation can fail. This can either be due to exhausting all possibilities, or, more likely, hitting the designer-set iteration limit. In the case of failure, if permitted, the algorithm is run again with a different seed (selected using the previous generation's RNG). Additionally, a `failCount` can be defined to prevent this process from happening too many times.

3 PERFORMANCE & OPTIMISATIONS

3.1 The Allocation Problem

My initial implementation suffered from severe performance issues, dungeons with 25-35 rooms required approximately 30 seconds to generate, sometimes far more. This was unacceptable for any practical application. Profiling revealed that weighted room selection and allocator churn were the primary bottlenecks. The original approach allocated and deallocated many arrays on each iteration.

```
while (validRooms.Count != 0) {
Room hopeful = RoomArchetype.GetRandomRoom(validRooms);
validRooms.RemoveSwapBack(hopeful);
List<Vector2Int> occMap = hopeful.GetOcc(match.GetGridCoords());
if (!CanFit(occMap, occupied)) continue;
// ...
}
```

Each call to `GetRandomRoom` was taking a disproportionate amount of compute time due to needing to reallocate the entire room array and then perform a swapback removal for each weighted selection. Originally, the process looked like this. A typical 30-room dungeon may recurse 40-50 levels deep, with each level performing 5-10 weighted selections. This resulted in hundreds of allocations per generation, each adding to garbage collection pressure. I was able to vastly improve allocations using a dedicated structure, the "Weighted Bag".

3.2 Weighted Bag Implementation

I developed a `WeightedBag<T>` structure that performs weighted selection without incurring the cost of repeated allocations. The original Weighted Selection routine is a form of roulette wheel selection. The key insight, and what Weighted Bag is built around, is that roulette wheel selection doesn't require modifying the underlying item array, only tracking which indices have been selected. The structure makes use of C#'s `ArrayPool` to rent index arrays, and removes selected items via swap-back to avoid reallocated underlying arrays. The struct is defined loosely like this.

```
internal ref struct WeightedBag<T> {
    readonly T[] _items;
```

```
    readonly ReadOnlySpan<int> _weights;
    readonly int[] _indices;
    int _count;
    int _weightSum;

    public WeightedBag(in T[] items, in ReadOnlySpan<int> weights) {
        _items     = items;
        _weights   = weights;
        _indices   = ArrayPool.Rent(_items.Length);
        _indecies  = [1..n];
        _count     = n;
        _weightSum = 0;
        _weightSum = weight.sum();
    }

    public bool TryNext(Random rng, out T result) {
        if (_count == 0) return false;

        float r          = rng.NextDouble() * _weightSum;
        float cumulative = 0f;

        for (int i = 0; i < _count; ++i) {
            int index  = _indices[i];
            cumulative += _weights[index];
            if (!(cumulative >= r)) continue;
            result     =  _items[index];
            _weightSum -= _weights[index];

            // Swap back and shrink
            _count--;
            _indices[i] = _indices[_count];
            return true;
        }

        result = default;
        return false;
    }
}
```

As discussed above, the bag rents from the `ArrayPool` to avoid allocating directly (if at all) iteration to iteration. The `ref struct` declaration ensures the bag is stack-allocated and cannot bleed into the heap. The structure maintains a running total of remaining weighted, removing the need to recalculate the sum on each iteration. With the weighted bag implemented, the prior selection's code can then be rewritten like so.

```
Room[] candidates     = GetValidRooms(matching.GetDirection());
Span<int> weights     = stackalloc int[candidates.Length];
weights = candidates.Weights;
using WeightedBag<Room> bag = new(candidates, weights);

while (bag.TryNext(_rng, out Room hopeful)) {
    RSpan<Vector2Int> occMap = hopeful.GetOcc(match.GetGridCoords());
    if (!CanFit(occMap, occupancy.Buffer.AsSpan())) continue;
    ...
```

```
}
```

3.3 Additional Optimisations

Beyond weighted selection, I implemented several other performance improvements, which when combined with weighted bag vastly improved generation time from 30 seconds to 4 seconds, a 7.5x improvement.

Of these, `RentBuffer` and Lazy Pooling were major gains. `RentBuffer<T>` essentially acted as a wrapper for C#'s `ArrayPool` system, which ensured proper cleanup even during exception unwinding, and made the code's allocation intentions explicit. I leverage the JIT by using `readonly struct`, which will optimise this to direct array access, without wrapper overhead.

```
internal readonly struct RentBuffer<T> : IDisposable {
    public readonly T[] Buffer;
    public readonly int Count;
    public RentBuffer(T[] buf, int cnt) {
        Buffer = buf;
        Count = cnt;
    }

    public void Dispose() => ArrayPool<T>.Shared.Return(Buffer);
}
```

The Lazy Pooling strategy ensures that occupancy data that must persist across recursive calls is rented from the `ArrayPool` only when necessary. Failed placement attempts never allocate merged occupancy buffers, the allocation only occurs after all constraint checks pass and recursion is imminent.

```
// Only allocate when we know we're recursing
Vector2Int[] occBuf = ArrayPool<Vector2Int>.Shared.Rent(occLen);
Span<Vector2Int> occSpan = occBuf.AsSpan(0, occLen);
baseOccupancy.CopyTo(occSpan);
occMap.CopyTo(occSpan.Slice(occupancyCount));
using RentBuffer<Vector2Int> occRentBuffer = new(occBuf, occLen);
```

3.4 Remaining Bottlenecks

Still, there are some major issues. While 4 seconds is pretty good, considering it's only an average case, and some seeds can take far longer due to generation failing, shaving off any additional time is of high priority. While WeightedSelection was certainly the biggest bottleneck, `CanFit()` is right up there as another major bottleneck. To illustrate the problem, you can see that `CanFit()` is defined as so...

```
static bool CanFit(in ReadOnlySpan<Vector2Int> occMap,
                   in ReadOnlySpan<Vector2Int> occupied) {
    foreach (Vector2Int occ in occMap) {
        foreach (Vector2Int t in occupied) {
            if (t == occ) return false;
        }
    }
    return true;
}
```

This is an `O(n x m)` next loop. Once or twice this is no big deal, but at the scale it is being called, guaranteed at least once per room attempt, sometime 10's of times per recursive layer, this cost gets expensive extremely quick. For a 30-room dungeon, with 6 cells per room, that's 180 comparisons, and if weighted bag tries 5-10 candidates before finding one that fits, that's 900 to 1800 comparisons per successful placement. One fix would be to replace the occupancy list with a `HashSet<Vector2Int>`, or spatial hash grid, which would reduce overlap check to just `O(n)`, which could yield a major speedup. Additionally, the fit check could account for the fact that many archetypes share the same dimensions, and we'd skip checks we already know the outcome of.

4 ROOM DESIGN & CONSTRAINTS

4.1 Archetype System

For CMPM121, I redesigned the room catalogue around a cleaner archetype system, rather than enumerating every possible room variant, rooms are categorized by their door configurations. Within each archetype, multiple variants provide visual and gameplay diversity, without complicating generation logic. For CMPM121's larger-scale rooms, designed to accommodate specific enemy types and combat scenarios without ballooning the cost of generation, this was essential.

I wanted to support non-uniform room dimensions, though I did not have the time to properly test and implement this functionality. Still, I designed the architecture such that it could be added without too much additional work. I chose to implement through an occupancy grid. Each room defines its occupancy as an `nxm` rectangle, then "chisles" out the actual shape. The `GetOccupancy()` method returns a span representing only the occupied cells. This approach would allow non-rectangular rooms, such as L-shaped, triangular and interior voids without complicating placement logic.

4.2 Special Room Placement

The generator includes support for special room types. I implemented three, the starting room, target room and the treasure room. The placement rules are hardcoded, but represent candidates for a more modular "graph modifier" system, where each special room type would have associated placement rules, priority values and constraint checks that execute during a finalisation phase.

The starting room is simple enough, the room at the origin, and the first room placed, will always be a starting room. Target rooms are also a guaranteed room type, and are placed at the deadend farthest from the spawn point.

```
1  List<RoomNode> endrooms = RoomNode.GetEndRooms(_roomGraph[STARTING_POS]);
2  ReplaceRoom(endrooms[Mathf.RoundToInt(TargetRoomDistance
3                       * (endrooms.Count - 1))], target);
```

The `TargetRoomDistance` parameter (0-1) controls placement along the sorted list of endrooms, calculated through BFS. At 1.0, it selects the farthest, 0.5 the mid-way, and so on. This gives designers control over pacing without hardcoding specific placement logic. For the sake of simplicity, only single cell rooms are considered. Multi-cell rooms are ignored.

The treasure room follows similar logic, but with additional constraints. It can only spawn if there's at least one endroom available after the target room is placed, and through a `CanSpawnTreasureRoom()` call, can be further restricted based on game state.

5 CHALLENGES & FUTURE WORK

5.1 Technical Challenges

The primary challenge throughout development was achieving acceptable performance within the constraints of recursive backtracking, an inherently inefficient algorithm. I touched on this in the weighted selection part, but array allocations were a major concern for me. I ended up having to refactor the room class to pre-calculate the door positions and store it in a cache. For doors and occupancy, I make use of an additional internal buffer which holds the offsets that are calculated by `GetDoors()` and `GetOccupancy()`. A `ReadOnlySpan` is returned to give the map generator access to the positions for the purpose of computation. If we need this data to persist, as in the case of recursion, then and only then will we rent from the `ArrayPool`.

Early optimisation attempts focused on algorithmic improvements (better heuristics, smarter frontier ordering) when the real issue was allocation overhead. Weighted selection wasn't inherently slow, allocating arrays on every call made it slow. The Weighted

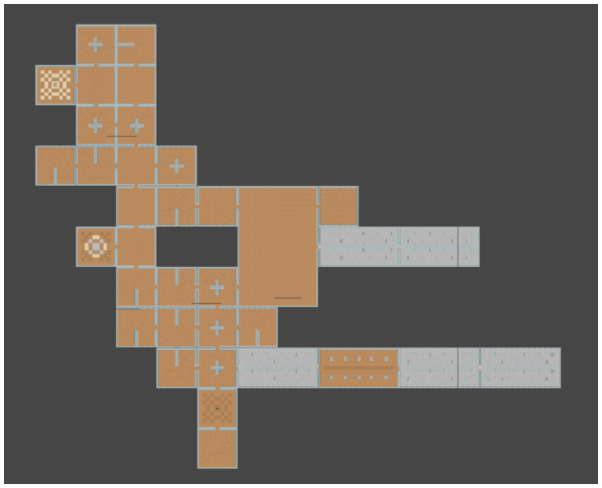

Figure 3. A large map generated with many special rooms.

Bag optimisation didn't change algorithmic complexity, but delivered a massive speedup by eliminating allocations. Modern C# performance features greatly helped in this endeavour, giving me fine control over allocations through `Span<T>`, `stackalloc`, `ref struct` and `ArrayPool<T>`. These features let me write code that's both safe and nearly as fast as unmanaged C++ code, though I'd think the design of a language like Zig, exposing explicit allocations, would have helped this process far more.

5.2 Improvements

The most significant flaw is the slowness of `CanFit()`, but there are some other major things that irk me.

1. Path Reconvergence would go a long way in making dungeons feel more interesting to explore, and likely would reduce failure rates during generation.
2. The Special Room system is rigid, and doesn't allow for more sophisticated room replacement without modifying the generation code more heavily. Ideally, special rooms could have more impact on dungeon generation (eg. hidden rooms, with obscured entrances)
3. While I laid the foundation for it, lacking Non-Rectangular rooms is a major limitation, and some systems would need updating. Doors are currently positioned assuming rooms occupy a full nxm rectangle, concave boundaries need explicit door coordinates, rather than automatic boundary detection. `Room.Place()` also assumes a bottom-left origin, which may not be appropriate for all archetypes.
4. Room weights are static, but generation feel could be greatly improved if generation context could modify weights. Archetypes with many doors could be discouraged as the room count is close to being hit.