Lecture 2
Search
Lecture Plan

1. Concepts:
   I. State space
   II. Search Tree

2. Basic Search
   I. Graph Search
   II. Tree Search

3. Local Search
   I. Discussion on local and global optima

4. Shortest Path
   i. Dynamic Programming for Path Finding
   ii. (Heuristic Search)
Algorithms

1. Breadth First Search
2. Depth First Search
3. Iterated Deepening
4. Bidirectional Search
5. Best First Search
6. Greedy Hill Climb with Random Restarts
7. Simulated Annealing
8. Local Beam Search
9. Stochastic Local Beam Search
10. Dynamic Programming for Path Finding
Basic State Space (A)

A directed graph.

Nodes: States

Edges: Transitions
Search Scenario 1

Scenario

• Find a goal state from a given origin state.
• The search space is finite.
• The search space may contain directed loops.
• Each state has only finitely many transitions from it.
• The goal state is (locally) identifiable.

Potential application:

• Finding a route from a specific origin to a known goal.
• Finding a series of actions to transform the environment from how it is to how you would like it to be.
Graph Search

- We have:
  - A set of visited nodes. Initially empty.
  - A set of frontier nodes, whose existence we know of but whose neighbors we are ignorant of. Initially contains only the origin.

- At each step we:
  - Select a node from the frontier and 'expand' it. This involves performing all possible transitions on the node to find its neighbors.
  - If the goal is one of the neighbors, we terminate the search. Else we:
    - Remove the node from the frontier.
    - Add the node to the visited set.
    - Add those neighbors that are not already in the visited set or frontier set to the frontier.
Search Tree

To find a path and not just a goal node, we keep track of the 'search tree'.

This records each transition 'explored' when expanding a node and the nodes involved.

We can think of a transition being explored if it leads to the goal or to a node being added to the frontier.
Graph Search: Example

Step: 0 (Initial setup)

Visited:

Frontier: 1

We start with our origin as the only node in the frontier. At each step we expand the left most node in the frontier...

Examined State Space: 1

Search Tree: 1

1
Graph Search: Example

Step: 1
Visited: 1
Frontier: 2 3

Examined State Space:

Unless we find our goal, the neighbours we find are placed in the frontier.

The node we expand is placed in the visited set.

Explored transitions are added to the search tree.

Search Tree:
Graph Search: Example

Step: 2
Visited: 1 2
Frontier: 3 4 5

Examined State Space:

Search Tree:
Graph Search: Example

Step: 3

Visited: 1 2 3
Frontier: 4 5 6

Examined State Space:

Search Tree:
Graph Search: Example

Step: 4
Visited: 1 2 3 4
Frontier: 5 6

We expand node 4 and find a path to 5. But 5 is already in the frontier, so it is not added again. Nor is the transition added to the search tree.

Examined State Space:

Search Tree:

Notice that the topologies (structure) of the examined state space and the search tree are now different. One is a graph, the other a tree.
Graph Search: Example

Step: 5
Visited: 1 2 3 4 5
Frontier: 6 7

Examined State Space:

Search Tree:
Graph Search: Example

Step: 6
Visited: 1 2 3 4 5 6
Frontier: 7

We expand node 6 and find a path to 1. But 1 is in the visited set, so it is not added to the frontier. Nor is the transition added to the search tree.

Examined State Space:

Search Tree:
Graph Search: Example

Step: 7

Visited: 1 2 3 4 5 6 7

Frontier:

Examined State Space:

Search Tree:

We expand node 7 and find our goal node, 8. The search is terminated successfully.
Graph Search: Example

Step: 7
Visited: 1 2 3 4 5 6 7
Frontier:

Examined State Space:

Search Tree:

And we can backtrack on the search tree to find the path to the goal node from our origin.
Search Algorithms: BFS & DFS

Algorithms:

1. Breadth first search
   
   **Frontier is first in first out queue.** Thinking of the state space as consisting of layers $t$ transitions away from the origin, this means we expand all nodes in a layer before progressing to the next layer. The example we just saw was BFS.

2. Depth first search
   
   **Frontier is first in last out stack.** Thinking of the state space as consisting of layers $t$ transitions away from the origin, this means we expand a through all layers until we cannot proceed any further, then backtrack to try other routes.
Search Algorithms: BFS & DFS

BFS
Frontier First In First Out

DFS
Frontier First In Last Out
Search Algorithms: BFS & DFS

PROBLEM: Search spaces are very large.
Storing all nodes that we visit can be intractable.

IDEA: Only store nodes on the frontier

- \( b = \text{'breadth of each node'} = \text{number of transitions out of it.} \)
- \( d = \text{'depth of goal'} = \text{distance in terms of transitions that our current search location (the node we choose for expansion) is.} \)

Generally, the frontier of DFS is much less than the frontier in BFS: \( O(b \times d) \) rather than \( O(b^d) \).

So if we choose not to store the nodes that we have visited, DFS offers enormous memory advantages.

This is tree search (it turns the graph into a tree, from the perspective of the search algorithm).
Tree search

- We have:
  - A set of frontier nodes, whose existence we know of but whose neighbors we are ignorant of. Initially contains only the origin.

- At each step we:
  - Select a node from the frontier and 'expand' it. This involves performing all possible transitions on the node to find its neighbors.
  - If the goal is one of the neighbors, we terminate the search. Else we:
    - Remove the node from the frontier.
    - Add those neighbors that are not already in the frontier set to the frontier.
Search Algorithms: BFS & DFS

QUESTION:

If the conditions specified in our basic search scenario hold, which of these algorithms are guarantee to find the goal node in a finite number of steps when performing a Tree Search?

- A. Both BFS and DFS.
- B. Only BFS
- C. Only DFS
- D. Neither
Search Algorithms: BFS & DFS

QUESTION:

If the conditions specified in our basic search scenario hold, which of these algorithms are guarantee to find the goal node in a finite number of steps when performing a Tree Search?

- A. Both BFS and DFS.
- B. Only BFS
- C. Only DFS
- D. Neither
Search Algorithms: BFS & DFS

DFS will find the goal node if there are no loops in a finite search space. It is not guaranteed to do so in the common situations where:

- The search space is infinite.
- The search space may contain directed loops.

BFS will do so as long as

- Each state has only finitely many transitions from it.
- The goal state is accessible by finite transitions from the origin.
Tree Search: Example

Frontier Node: 

BFS

Search Tree: 1

DFS

Search Tree: 1
Tree Search: Example

Frontier Node: 1

BFS

Search Tree:

DFS

Search Tree:
Tree Search: Example

Frontier Node: 

BFS

Search Tree:

DFS

Search Tree:
Tree Search: Example

Frontier Node: 1

BFS
Search Tree:

DFS
Search Tree:

Notice 1 is placed in the search tree again.
Tree Search: Example

Frontier Node: 1

BFS

Search Tree:

DFS

Search Tree:

DFS is in big trouble... Its entered an infinite loop and will never find the goal!

BFS is also placing duplicate nodes in the search tree. But it will cope with this and reach the goal.
Tree Search: Example

Frontier Node: 

BFS

Search Tree:

DFS

Search Tree:
Tree Search: Example

Frontier Node: 1

BFS

Search Tree:

DFS

Search Tree:

Bye bye...
Tree Search: Example

Frontier Node: 1

BFS

Search Tree:

1
2 3
4 5 6
5 7 1
7

DFS

Search Tree:

1
2 3
6
1
2 3
6
1
2 3
6
Tree Search: Example

Frontier Node: 

BFS

Search Tree:

DFS

Search Tree:
Tree Search: Example

Frontier Node: 

BFS
Search Tree:

DFS
Search Tree:

Success!
Iterated Deepening

Problem: The advantages of DFS are only available in acyclic (loopless) finite search spaces.

Solution:

Algorithm 3: Iterative Deepening

- We run a DFS algorithm repeatedly, with a tuning parameter $t=1, 2, 3 \ldots$. At each iteration, we treat nodes at depth $t$ as if there are no transitions from them. This terminates depth first strings and means we backtrack to examine the remainder of the nodes reachable in up to $t$ transitions from the origin.

ID is guaranteed to reach the goal in finite steps so long as

- Each state has only finitely many transitions from it.
- The goal state is accessible by finite transitions from the origin.

- We go over the same ground repeatedly, but since the number of nodes reachable with $t$ transitions from the origin is generally exponential on $t$ - $O(b^t)$ - this is less of a penalty than one might think. And we keep the memory advantages of DFS.
Iterated Deepening: Example

Frontier Node: 
Terminal Node: 

ID
T=1
Search Tree: 

DFS
Search Tree: 
Iterated Deepening: Example

Frontier Node: ★
Terminal Node: ●
ID
T=1
Search Tree:

DFS
Search Tree:

Iteration 1 terminates
Iterated Deepening : Example

Frontier Node:  
Terminal Node:  
ID
T=2
Search Tree:

So we increase T and begin again...

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node: 🟢
Terminal Node: 🟥

ID
T=2
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node: [Orange Node]
Terminal Node: [Black Node]

ID
T=2
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node: 1
Terminal Node: 6
ID T=2
Search Tree:

DFS
Search Tree:

Iteration 2 terminates
Iterated Deepening: Example

Frontier Node: 1
Terminal Node: 8
ID
T=3
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node: 1
Terminal Node: 8
ID
T=3
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node: ●
Terminal Node: ●

ID
T=3
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node:  
Terminal Node:  
ID 
T=3 
Search Tree:

DFS 
Search Tree:
Iterated Deepening: Example

Frontier Node: 
Terminal Node: 

ID
T=3
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node:  
Terminal Node:  
ID
T=3
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node: 1
Terminal Node: 5

ID
T=3
Search Tree:

DFS
Search Tree:

Iteration 3 terminates
Iterated Deepening: Example

Frontier Node: ★
Terminal Node: ●

ID
T=4
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node: 
Terminal Node: 
ID
T=4
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node: ●
Terminal Node: ▢

ID
T=4
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node:  
Terminal Node:  
ID  
T=4  
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node:  ●
Terminal Node:  ●
ID
T=4
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node:  
Terminal Node:  
ID  
T=4  
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node:  
Terminal Node: 
ID
T=4
Search Tree:

DFS
Search Tree:
Iterated Deepening: Example

Frontier Node: 1
Terminal Node: 8
ID
T=4
Search Tree:

Success!
Bidirectional Search

- If we know the goal state we can run two simultaneous searches: one forward from the initial state and the other backward from the goal, hoping they meet in the middle.
  - Generally $2b^{d/2}$ is much smaller than $b^d$.
- Requires that we can search backwards: Not always possible.
State Space (B)

A directed graph.

A fitness function: takes a state as an argument and returns a value.

Search Scenario

Find the node that maximizes the fitness function.

OR

Find a node that is relatively fit.

Find the statistical model that best explains the data.
Best First Search

- Frontier is ordered by fitness. At each step in the search process, we expand the fittest currently known node.

PROBLEM: Search spaces are very large. Even if we proceed using a tree search, Best First Search will end up storing a lot of nodes.

But, if we think of the node we choose to expand at each step as our current location, we can hope that simply continuing to choose the best node from whatever our current location is will lead us to the optimally fit node.

So we do not need to store either nodes that we have visited, nor a frontier set.
Local Search

- We have:
  - The 'current' node, initially the origin.
  - A 'transition' function, \( f(n,N) = m \), that takes the current node, \( n \), and a set of nodes, \( M \), and returns a node, \( m \) from that set.
  - A 'termination' function, \( g(n,m,T) \), that takes the current node, \( n \), another node, \( m \), and possible tuning parameters, \( T \), as arguments and returns a boolean value that decides whether the algorithm terminates.

- At each step we:
  - 'Expand' the current node, \( n \). This involves performing all possible transitions on it to find its neighbors, which are \( M \).
  - Find the potential transition \( m = f(n,M) \) and see if we should transition to \( m \) or terminate and return the current node as the output of the algorithm by checking \( g(n,m,T) \).
Greedy Hill Climb

Transition and termination functions:

- At each step in the search process, we transition to the fittest neighbor node if it fitter than the current node. Otherwise we terminate the search, selecting the current node as the solution.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.

We expand the local node and evaluate the fitness of its neighbors.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.

We find that the fittest neighbor is fitter than our current node, so we transition
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness.
Notice bidirectional edges.

We forget about the unselected neighbors. (But we will continue to show them in black.)
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness.
Notice bidirectional edges.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.

We didn’t specify what to do with ties. Let’s choose one of them arbitrarily.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.

We didn’t specify what to do with ties. Let’s choose one of them arbitrarily.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.

None of the neighbors of our current location are better (fitter). So we terminate.
Greedy Hill Climb: Example

Numbers in node are now the node’s fitness. Notice bidirectional edges.

Here is the path we took...
Greedy Hill Climb & Local Maxima

Problem:
Greedy Hill Climb & Local Maxima

Problem:
Greedy Hill Climb & Local Maxima

Question: How might we try to avoid getting stuck at local optima?
Greedy Hill Climb with Random Restarts

Repeat the GHC algorithm x times.
Simulated Annealing

Idea: We can avoid local maxima if we sometimes go down...

- At each step, our transition function randomly selects a single neighbor node to potentially transition to.

- Given our current node, \( n \), and the selected neighbor node, \( m \), the algorithm will decide whether to transition from \( n \) to \( m \) probabilistically. Typically we make the transition with probability 1 if \( m \) is fitter than \( n \). Otherwise the chance that we make the transition depends on the relative fitness of \( n \) and \( m \) and on a tuning parameter, called the temperature (from physical annealing), or \( t \).

- \( t \) gradually approaches 0 according to a specified schedule, and the chances of choosing to a less fit state approaches 0 as \( t \) approaches 0.
Simulated Annealing

T-Value: 1  Move: 0

Probability of transition from n to m:
if f(m) > f(n) then: 1
else: \( t \times \frac{f(m)}{f(n)} \)

T-Schedule: T decreases by .1 every 5 moves until it reaches 0.

We terminate when no transitions are possible.
Simulated Annealing

T-Value: 1  Move: 0

Randomly selected initial state.
Simulated Annealing

T-Value: 1  Move: 1

Randomly selected neighbor of current state.

It is less fit than our current state. The probability that we will transition is:

1 * 2/3 = 2/3.
Simulated Annealing

T-Value: 1  Move: 1

We did transition.

We will often go ‘downhill’ early in the algorithm while the T value is still high...
Simulated Annealing

T-Value: 1  Move: 2

This time the randomly selected neighbor is fitter than our current state, so we are certain to transition.
Simulated Annealing

T-Value: 1  Move: 2
Simulated Annealing

T-Value: 1  Move: 3
Simulated Annealing

T-Value: 1    Move: 3
Simulated Annealing

T-Value: 1    Move: 4
Simulated Annealing

T-Value: 1  Move: 4

We transitioned downhill again, and are now back where we started!
Simulated Annealing

T-Value: 1  Move: 5
Simulated Annealing

T-Value: 1  Move: 5
It's the sixth move, so the temperature is decreased, making it less likely we will transition downhill.
Simulated Annealing

T-Value: .9  Move: 6
Simulated Annealing

T-Value: .9  Move: 7
Simulated Annealing

T-Value: .9   Move: 7
Simulated Annealing

T-Value: .9    Move: 8
Simulated Annealing

T-Value: .9   Move: 8

This time we did not make the downhill transition.

But it still counts as a move.
Simulated Annealing

T-Value: .9  Move: 9
Simulated Annealing

T-Value: .9      Move: 9
Simulated Annealing

T-Value: .9  Move: 10

The algorithm means that transitions to nodes that are equally fit as the current node will occur with probability $T$. 
Simulated Annealing

T-Value: .9    Move: 10
Simulated Annealing

T-Value: .8

Down goes the temperature again...
Simulated Annealing

T-Value: .8    Move: 11
Simulated Annealing

T-Value: .8    Move: 12
Simulated Annealing

T-Value: .8    Move: 12

We seem to be moving towards the global optimum, but it is really just random movement at the moment.
Simulated Annealing

T-Value: .8  Move: 13
Simulated Annealing

T-Value: .8     Move: 13
Simulated Annealing

T-Value: .8
Move: 14
Simulated Annealing

T-Value: .8  Move: 14
Simulated Annealing

T-Value: .8  Move: 15
Simulated Annealing

T-Value: .8  Move: 15
Simulated Annealing

T-Value: .7
Move: 16
Simulated Annealing

T-Value: .7
Move: 16
Simulated Annealing

T-Value: 0.7  Move: 17
Simulated Annealing

T-Value: .7  Move: 17
Simulated Annealing

T-Value: .7  Move: 18
Simulated Annealing

T-Value: .7  Move: 18
Simulated Annealing

T-Value: 0.7  Move: 19
Simulated Annealing

T-Value: .7  Move: 19
Simulated Annealing

T-Value: .7  Move: 20
Simulated Annealing

T-Value: .7    Move: 20
Simulated Annealing

T-Value: .6  
Move: 21
Simulated Annealing

T-Value: .6  Move: 21
Simulated Annealing

T-Value: .6  Move: 22
Simulated Annealing

T-Value: .6
Move: 22
Simulated Annealing

T-Value: .6  Move: 23
Simulated Annealing

T-Value: .6  Move: 23
Simulated Annealing

T-Value: .6  Move: 24
Simulated Annealing

T-Value: .6  Move: 24
Simulated Annealing

T-Value: .6   Move: 25
Simulated Annealing

T-Value: .6   Move: 25
Simulated Annealing

T-Value: .5  Move: 26

It's getting more and more difficult to move downhill...
Simulated Annealing

T-Value: .5   Move: 26
Simulated Annealing

T-Value: .5  Move: 27
Simulated Annealing

T-Value: .5    Move: 27
Simulated Annealing

T-Value: .5  Move: 28
Simulated Annealing

T-Value: .5  Move: 28
Simulated Annealing

T-Value: .5  Move: 29
Simulated Annealing

T-Value: .5
Move: 29
Simulated Annealing

T-Value: .5  Move: 30
Simulated Annealing

T-Value: .5  Move: 30
Simulated Annealing

T-Value: .4  Move: 31
Simulated Annealing

T-Value: .4  Move: 31
Simulated Annealing

T-Value: .4  Move: 32
Simulated Annealing

T-Value: .4
Move: 32
Simulated Annealing

T-Value: .4  Move: 33
Simulated Annealing

T-Value: 0.4  Move: 33
Simulated Annealing

T-Value: .4  Move: 34
Simulated Annealing

T-Value: .4  Move: 34
Simulated Annealing

T-Value: .4  Move: 35
Simulated Annealing

T-Value: .4  Move: 35
Simulated Annealing

T-Value: .3
Move: 36
Simulated Annealing

T-Value: .3  Move: 36
Simulated Annealing

T-Value: 0.3  Move: 37
Simulated Annealing

T-Value: .3   Move: 37
Simulated Annealing

T-Value: .3     Move: 38
Simulated Annealing

T-Value: .3  Move: 38
Simulated Annealing

T-Value: .3  Move: 39
Simulated Annealing

T-Value: .3  Move: 39
Simulated Annealing

T-Value: .3        Move: 40
Simulated Annealing

T-Value: .3  Move: 40
Simulated Annealing

T-Value: .2  Move: 41
Simulated Annealing

T-Value: .2  Move: 41
Simulated Annealing

T-Value: .2    Move: 42
Simulated Annealing

T-Value: .2  Move: 42
Simulated Annealing

T-Value: 0.2       Move: 43
Simulated Annealing

T-Value: .2  Move: 43
Simulated Annealing

T-Value: .2  Move: 44
Simulated Annealing

T-Value: .2
Move: 44
Simulated Annealing

T-Value: .2  Move: 45
Simulated Annealing

T-Value: .2    Move: 45
Simulated Annealing

T-Value: .1
Move: 46
Simulated Annealing

T-Value: .1  Move: 46
Simulated Annealing

T-Value: .1   Move: 47
Simulated Annealing

T-Value: .1   Move: 47
Simulated Annealing

T-Value: .1  Move: 48
Simulated Annealing

T-Value: 0.1
Move: 48
Simulated Annealing

T-Value: .1  Move: 49
Simulated Annealing

T-Value: .1  Move: 49
Simulated Annealing

T-Value: .1  Move: 50
Simulated Annealing

T-Value: .1      Move: 50
Simulated Annealing

T-Value: 0    Move: 51

The T-value has reached 0.

The algorithm will now function as a GHC.

But since we are already at a local maxima, we terminate.
Simulated Annealing

We saw how the algorithm worked:

1. Initially basically random movement.
2. Then a strong uphill bias, but with occasional ‘expeditions’ away from the route the GHC algorithm would take.
3. Finally, moving only uphill to reach the nearest local maximum.

Admittedly, we still didn’t get to the global optimum!

We would have had a higher chance of doing so with a slower cooling schedule.

But something else is going on here too...
Attractive Local Optima

The probability of terminating on a particular optima is not necessarily monotonically increasing with its fitness. Much depends on the state space.
Simulated Annealing and Attractive Local Optima

- The search we just saw was unable to break free of this attractive local optima – it spent nearly all its time in the blue nodes. The nature of SA means that it is unlikely to go on long expeditions downhill, especially as the temperature cools.

- Lots of literature on how to get the best out of SA through different cooling schedules and transition functions.
Local Beam Search

- We select \( n \) initial nodes. Each of these will be termed a ‘particle’.
- At each step:
  - Create a set, \( S \), of nodes consisting of the nodes the particles are currently at and the neighbors of these nodes.
  - The particles transition to the \( n \) fittest nodes in \( S \). (Different strategies can be used to break ties.)
- The search terminates when no transitions are made.

If you don’t pay attention here, it can seem that what is being proposed is identical to running \( n \) random restarts, except that we do it concurrently instead of consecutively.

But note that we do not choose the best neighbor for each climb and have the climb transition to it. We choose the \( n \) best neighbors among all climbs, and have the climbs transition to these.

The individual climbs are now communicating with each other. This is a simple example of Swarm Intelligence.
Local Beam Search

Number of particles: 3

Randomly selected initial states.
Local Beam Search

Number of particles: 3

We select the best three states from these nodes and their neighbors, breaking ties arbitrarily, except that we will choose a node that a particle is at over a neighbor node.
Local Beam Search

Number of particles: 3

And our particles transition to them...
Local Beam Search

Number of particles: 3
Local Beam Search

Number of particles: 3
Local Beam Search

Number of particles: 3
Local Beam Search

Number of particles: 3

None of our particles will transition to another node.

So we terminate.
Local Beam Search

- We found the global optimum.
  - Don’t conclude too much from that – it’s a tiny state space.
  - Certainly don’t assume that LBS is ‘better’ than SA. They both have advantages and disadvantages: You should think about the problem that you are working on when choosing search algorithms.

- All the particles ended up clustered together.
  - This is a serious problem for LBS. It looked good this time because they clustered together around the global maximum, but generally that would not be the case.
  - This clustering behaviour often happens very quickly.
  - The idea was to use LBS to get a greater awareness of the state space than GHC provided. This clustering behaviour limits the effectiveness of this strategy.
Stochastic Local Beam Search

- We select \( n \) initial nodes. Each of these will be termed a ‘particle’.
- At each step:
  - Create a set, \( S \), of nodes consisting of the nodes the particles are currently at and the neighbors of these nodes.
  - The particles transition to \( n \) nodes in \( S \) drawn at random but weighted by their relative fitness.
- The search terminates when no neighbor node is better than any of the current nodes.

Avoids the clustering issue of LBS and is generally preferable.
State Space (C)

A directed graph.

Edges have cost values.
Search Scenario

Find an optimal path (in terms of the cost function) from a given origin state to a given goal state.

- The search space is finite.
- The search space and edge costs are known.
- The search space is layered.
  - The nodes at which a n-step path from the origin terminates are disjoint. The search space and edge costs are known.

Potential applications:
Finding the shortest route from location A to location B.
Dynamic Programming for Path Finding

1. Assign the origin the value 0.

2. Proceed iteratively through each layer, \( L \), of the state space. Treat the origin as the 0 layer and begin at layer 1:
   
   1. Where \( m \rightarrow n \) is the value of an edge from node \( m \) to node \( n \), assign each node \( n \) in \( L \) the value:
      \[
      \min_{m \in L - 1} \left( \text{value}(m) + \min_{m \rightarrow n} \right)
      \]

   2. Add a pointer from the node \( m \) that minimizes the above equation to \( n \).

3. Backtrack from the Goal to the origin using these pointer to find the optimal path.
### Dynamic Programming for Path Finding

We start by assigning the value 0 to our origin.
Dynamic Programming for Path Finding

Iterating through the layers, for each node in each layer, $L$, we assign it:

$$\min \left( \text{value}(m) + \rightarrow_{mn} \right), \ m \in L - 1.$$ 

Then we add a pointer from the node to the node in layer $L-1$ that satisfied the minimization.
We’ll introduce the relevant edge values as we go...
Since there is only one node in layer 0, and it has an assigned value 0, the nodes of layer 1 are just assigned the edge costs to them from the origin.
Dynamic Programming for Path Finding

Layer 0  Layer 1  Layer 2  Layer 3  Layer 4

0  1  4  7  G

And we add a pointer from the node to the origin.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>O</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>O</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
Dynamic Programming for Path Finding

From | To | Cost
---|---|---
0   | 1  | 4  
0   | 2  | 2  
0   | 3  | 7  

Layer 0
Layer 1
Layer 2
Layer 3
Layer 4
Dynamic Programming for Path Finding

Layer 0 | Layer 1 | Layer 2 | Layer 3 | Layer 4
---|---|---|---|---
0 | 0 | 4 | 4 | 7
1 | 1 | 2 | | 5
2 | 2 | | 8 | 9
3 | | | | G

From | To | Cost
---|---|---
O | 1 | 4
O | 2 | 2
O | 3 | 7
Dynamic Programming for Path Finding

Layer 0

Layer 1

Layer 2

Layer 3

Layer 4

From | To | Cost
--- | --- | ---
O   | 1   | 4
O   | 2   | 2
O   | 3   | 7
Dynamic Programming for Path Finding

Layer 0 | Layer 1 | Layer 2 | Layer 3 | Layer 4
---|---|---|---|---

0 4 1 2 2 7 3

From | To | Cost
---|---|---
0 1 | 4
0 2 | 2
0 3 | 7
Now things get interesting.

For each node in layer 2 we assign it:
\[
\min \left( \text{value}(m) + \overrightarrow{mn} \right), \quad m \in L - 1.
\]

Let's start with node 4...
Dynamic Programming for Path Finding

We choose the minimizer of:
Node 1: 4 + 4 = 8
Node 2: 2 + 7 = 9
Node 3: 7 + 4 = 11

Edges into node 4.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Dynamic Programming for Path Finding

We choose the minimizer of:
Node 1: 4+4=8
Node 2: 2+7=9
Node 3: 7+4=11
Dynamic Programming for Path Finding

We choose the minimizer of:
- Node 1: 4+4=8
- Node 2: 2+7=9
- Node 3: 7+4=11

Edges into node 4.
Dynamic Programming for Path Finding

We choose the minimizer of:
Node 1: 4+4=8
Node 2: 2+7=9
Node 3: 7+4=11

Edges into node 4.
Dynamic Programming for Path Finding

We choose the minimizer of:

Node 1: 4+4=8
Node 2: 2+7=9
Node 3: 7+4=11

Edges into node 4.
Dynamic Programming for Path Finding

Layer 0 Layer 1 Layer 2 Layer 3 Layer 4

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

We choose the minimizer of:
Node 1: 4+4=8
Node 2: 2+7=9
Node 3: 7+4=11

Edges into node 4.
We choose the minimizer of:
Node 1: 4+4=8
Node 2: 2+7=9
Node 3: 7+4=11

Edges into node 4.
Dynamic Programming for Path Finding

We choose the minimizer of:
Node 1: 4+4=8
Node 2: 2+7=9
Node 3: 7+4=11
Dynamic Programming for Path Finding

We choose the minimizer of:

Node 1: 4+4=8
Node 2: 2+7=9
Node 3: 7+4=11

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Dynamic Programming for Path Finding

We choose the minimizer of:
Node 1: 4+4 = 8
Node 2: 2+7 = 9
Node 3: 7+4 = 11
Dynamic Programming for Path Finding

For node 5, we choose the minimizer of:
- Node 1: 4 + 3 = 7
- Node 2: 2 + 12 = 14
- Node 3: 7 + 10 = 17

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

Edges into node 5.
Dynamic Programming for Path Finding

For node 5, we choose the minimizer of:
Node 1: 4+3=7
Node 2: 2+12=14
Node 3: 7+10=17

From | To | Cost
-----|----|-----
1    | 5  | 3   
2    | 5  | 12  
3    | 5  | 10  

O
G
Dynamic Programming for Path Finding

For node 5, we choose the minimizer of:
Node 1: 4+3=7
Node 2: 2+12=14
Node 3: 7+10=17
Dynamic Programming for Path Finding

For node 5, we choose the minimizer of:

Node 1: 4+3=7
Node 2: 2+12=14
Node 3: 7+10=17

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
For node 6, we choose the minimizer of:
Node 1: 4+5=9
Node 2: 2+4=6
Node 3: 7+3=10
Dynamic Programming for Path Finding

For node 6, we choose the minimizer of:
Node 1: 4+5=9
Node 2: 2+4=6
Node 3: 7+3=10

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>
Dynamic Programming for Path Finding

For node 6, we choose the minimizer of:
Node 1: 4+5=9
Node 2: 2+4=6
Node 3: 7+3=10

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>
Dynamic Programming for Path Finding

For node 6, we choose the minimizer of:
Node 1: 4+5=9
Node 2: 2+4=6
Node 3: 7+3=10
Dynamic Programming for Path Finding

Note that nodes ALWAYS have pointers to the previous layer: This indicates the previous step of the optimal path to the node from the origin.

The do NOT always have pointers to the next layer. (See node 3). Such nodes are dead ends – they are certainly not on the optimal path from the origin to the goal.
Dynamic Programming for Path Finding

Let’s fast forward...
The goal is treated like any other node.
Dynamic Programming for Path Finding

We have finished the forward part of the algorithm. Now we trace the optimal path back to the origin from the goal using the pointers.
Dynamic Programming for Path Finding

And we are done!
Dynamic Programming for Path Finding

- If we have $m$ layers each with $n$ nodes, dynamic programming reduces an optimal path finding problem from $n^m$ to $m(n^2)$. 