Part I
Search

Illustration and explanation of Search algorithms

1 Depth-First Search

- Fast, but may end up in dead-loops. Not optimal and doesn't guarantee the best solution. To solve this, an iterative deepening addition might be required.
- Starts with an arbitrary node of the graph, but explores as far as possible along each branch.
- You begin at some node in the graph and continuously explore deeper and deeper into the graph while you can find new nodes that you haven't yet reached (or until you find the solution). Any time the DFS runs out of moves, it backtracks to the latest point where it could make a different choice, then explores out from there. This can be a serious problem if your graph is extremely large and there's only one solution, since you might end up exploring the entire graph along one DFS path only to find the solution after looking at each node. Worse, if the graph is infinite (perhaps your graph consists of all the numbers, for example), the search might not terminate. Moreover, once you find the node you're looking for, you might not have the optimal path to it (you could have looped all over the graph looking for the solution even though it was right next to the start node!)

2 Breadth-First Search

- Slow search, goes through every node. Better than A* for smaller graphs but gets slower the bigger the graph becomes and takes up more memory. Spreads like a wave outwards from the source node.
- Starts at some arbitrary node of the graph and explores the neighbouring nodes first, before moving to the next level neighbours.

3 A*

Informed search algorithm which chooses the path with the lowest heurisitic cost overall. Uses a Best-First search to find the least-cost path from a given initial node to a goal node. Traverses through the graph and follows a path of the lowest expected total cost or distance. Heurisitics must be admissible (not overestimate the cost/distance to the goal).

- $f(n) = g(n) + h(n)$
  - g(n) = cost so far to reach node n
  - h(n) = estimated cost from n to goal
  - f(n) = estimated total costs of path through n to goal

3.1 Heuristic functions

Best heuristic function depends on the context of the data

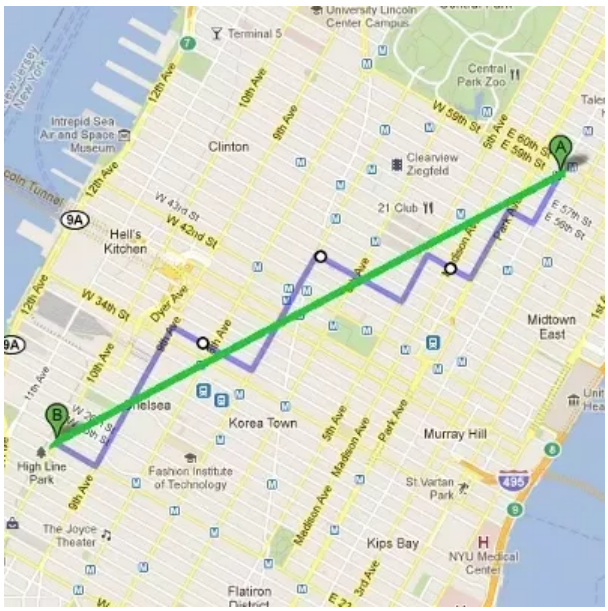

Figure 1: Euclidean (Green) & Manhattan (Blue) distances

3.1.1 Euclidean distance

- Measures the shortest distance in the plane, without any restrictions (bird-flight path)

3.1.2 Manhattan distance

- Works better with high dimensional vectors
  - Measures the shortest distance if you're only allowed to move horizontally or vertically

4 Iterative Deepening

A limit in search levels (usually for Depth-First search; can be used for A* as well), where the limit stops at the maximum specified level amd increases for each level. For A* the limit could be a maximum overall-cost to to not exceed when searching.

Saving memory by not needing to explored list is also good.

5 Local Search

Useful when we are interested in the solution state but not in the path to the goal. Local search only operates on the current state and move into neighbouring states.

5.1 Advantages

- Use very little memory (No recording of explored alternatives like DFS or BFS)
- Able to find reasonable solutions in large or infinite (continuous) state spaces.

5.2 Algorithms

5.2.1 Hill-climbing

- Find a new "better" next overall state by randomly selecting a variable, and trying to find an assignment to that variable which improves the overall score

5.2.2 Random walk

5.2.3 Simulated annealing

Animation of Simulated Annealing

- Starts with a randomized state. For each iteration in a loop, it will move to neighbouring states which decreases the energy (a counter which gets decreased; the higher it is, the worse the solution is), while also only accepting moves within the energy value.
- Basically: Decrease the energy slowly, accepting less bad moves at each energy level until at very low temperatures the algorithm becomes a greedy hill-climbing algorithm.

6 Planning

State-space search can operate in the forward direction (progression) or the backward direction (regression). Sometimes given the branching factor is it more efficient to search backward.

6.1 Progression

Searching forwards is called progression planning. Which means starting in the initial state and using the problem's actions to search forward for the goal state.

6.2 Regression

Searching backwards is called regression planning. Which means starting at the goal state and using the inverse of the actions to search backward for the initial state.

The principal question in **regression** planning is this:

- What are the states from which applying a given action leads to the goal?

Computing the description of these states is called regressing the goal through the action.

Main advantage of regression search is that it allows us to consider only **relevant** actions.

6.3 Partial-order planning (POP)

The POP algorithms explore the space of plans without commiting to a totally ordered sequence of actions. They work back from the goal, adding actions to the plan to achieve each subgoal. They are particularly effective on problems related to a divide-and-conquer approach.

7 Planning vs Search

Planning is the process of computing several steps of a problem-solving procedure before executing any of them. This problem **can** be solved by search.

The main difference between search and planning is the representation of states.

- In search, states are presented as a single entity (which may be quite a complex object, but its interal structure is not used by the search algorithm)
- In planning, states have structured representations (collections of properties) which are used by the planning algorithm.
- Planning systems do the following:

1. Open up action and goal representation to allow selection
2. Divide-and-conquer by subgoaling
3. Relax requirement for sequential construction of solutions

	Search	**Planning**
States	Data structures	Logical sentences
Actions	Code	Preconditions/outcomes
Goal	Code	Logical sentence (conjunction)
Plan	Sequence from dataset	Constrains on actions

Part II
Min-Max Algorithm (Game Tree Search)

Depending on which search is used, usually we start at the root down to the leaf of the first nodes, from there the whole tree is worked through. Top node is max and then alternating for each level down. Think of each level as two players, playing against eachother. First player wants to pick the winning number (Max), Second player wants to pick the winning number also, but it's the opposite of the First player so that's why it's (Min).

8 Alpha-Beta pruning

Alpha-Beta pruning helps with saving immense amount of computation when working with large game trees, by cutting off branches that doesn't need to be searched. The below link takes you to a video which explains it very well, from minute 22.

MIT OpenCourseWare - Search: Games, Minimax, and Alpha-Beta

Part III
Knowledge Representation

9 Propositional Logic

9.1 Operators

$\neg A$	"not A"	Negation
$A \wedge B$	"A and B"	Conjunction
$A \vee B$	"A or B"	Disjunction
$A \Rightarrow B$	"if A, then B"	Implication
$A \Leftrightarrow B$	"A if and only if B"	Equivalence

9.2 Inference by Enumeration using Truth Tables

9.3 Forward/Backward chaining using Horn Clauses

9.4 Resolution

Algorithm works by using **proof by contradiction**.

If the query is asking if D is true with respect to a knowledge base, using resolution, the query is instead changed to asking if D is false and the test whether there is a contradiction or not.

1. Convert all sentences to CNF
2. Negate the desired conclusion (converted to CNF)
3. Apply resolution rule until either
   (a) Derive false (a contradiction)
   (b) Can't apply any more
4. Resolution refutation is sound and complete
   (a) If we derive a contradiction, then the conclusion follows from the axioms
   (b) If we can't apply any more, then the conclusion cannot be proved from the axioms

10 Conjunctive Normal Form (CNF)

As a normal form, the formula is useful in automated theorem proving. It is similar to the product of sums form used in circuit theory.

Logical statements without the operators: $\wedge, \Rightarrow, \exists, \forall$

10.1 Translate to CNF

1. Push negations into the formula

   (a)

$\neg(A \vee B)$	$\longrightarrow$	$\neg(A) \wedge \neg(B)$
$\neg(A \wedge B)$	$\longrightarrow$	$\neg(A) \vee \neg(B)$

2. Apply the distributive law where a disjunction occurs over a conjunction, repeatedly until it's not possible anymore

   (a)

$A \vee (B \wedge C)$	$\longrightarrow$	$(A \vee B) \wedge (A \vee C)$

3. Drop universal quantifiers

   (a)

$\forall x Person(x)$	$\longrightarrow$	$Person(x)$

4. Eliminate biconditionals

   (a)

$A \Rightarrow B$	$\longrightarrow$	$\neg A \vee B$
$A \Leftrightarrow B$	$\longrightarrow$	$(A \Rightarrow B) \wedge (B \Rightarrow A)$

5. Skolemize variables - Each existing variable is replaced by a Skolem constant

   (a)

$\exists x Rich(x)$	$x \longrightarrow G1$	$Rich(G1)$
$\forall x Person(x) \Rightarrow \exists y Heart(y) \wedge Has(x, y)$	$y \longrightarrow G1$	$Person(x) \Rightarrow Heart(G1) \wedge Has(x, G1)$

10.2 Examples

1.

$A \Leftrightarrow B$	$\equiv$	$(A \Rightarrow B) \wedge (B \Rightarrow A)$
	$\equiv$	$(\neg A \vee B) \wedge (\neg B \vee A)$

2.

$(A \wedge B) \Leftrightarrow (A \vee B)$	$\equiv$	$((A \bigwedge B) \Rightarrow (A \bigvee B)) \bigwedge ((A \bigvee B) \Rightarrow (A \bigwedge B))$
	$\equiv$	$(\neg(A \bigwedge B) \bigvee (A \bigvee B)) \bigwedge (\neg(A \bigvee B) \bigvee (A \bigwedge B))$
	$\equiv$	$([\neg A \bigvee \neg B] \bigvee (A \bigvee B)) \bigwedge ([\neg A \bigwedge \neg B] \bigvee (A \bigwedge B))$
	$\equiv$	$(\neg A \bigvee \neg B \bigvee A \bigvee B) \bigwedge ([(\neg A \bigwedge \neg B) \bigvee A] \bigwedge [(\neg A \bigwedge \neg B) \bigvee B])$
	$\equiv$	$TRUE \bigwedge ([(\neg A \bigvee A) \bigwedge (\neg B \bigvee A)] \bigwedge [(\neg A \bigvee B) \bigwedge (\neg B \bigvee B])$
	$\equiv$	$[TRUE \bigwedge (\neg B \bigvee A) \bigwedge (\neg A \bigvee B) \bigwedge TRUE)$
	$\equiv$	$(\neg B \bigvee A) \bigwedge (\neg A \bigvee B)$

Part IV
Decision Tree

- A map of the reasoning process, good at solving classification problems
- Represents a number of different attributes and values
  - Nodes represent attributes

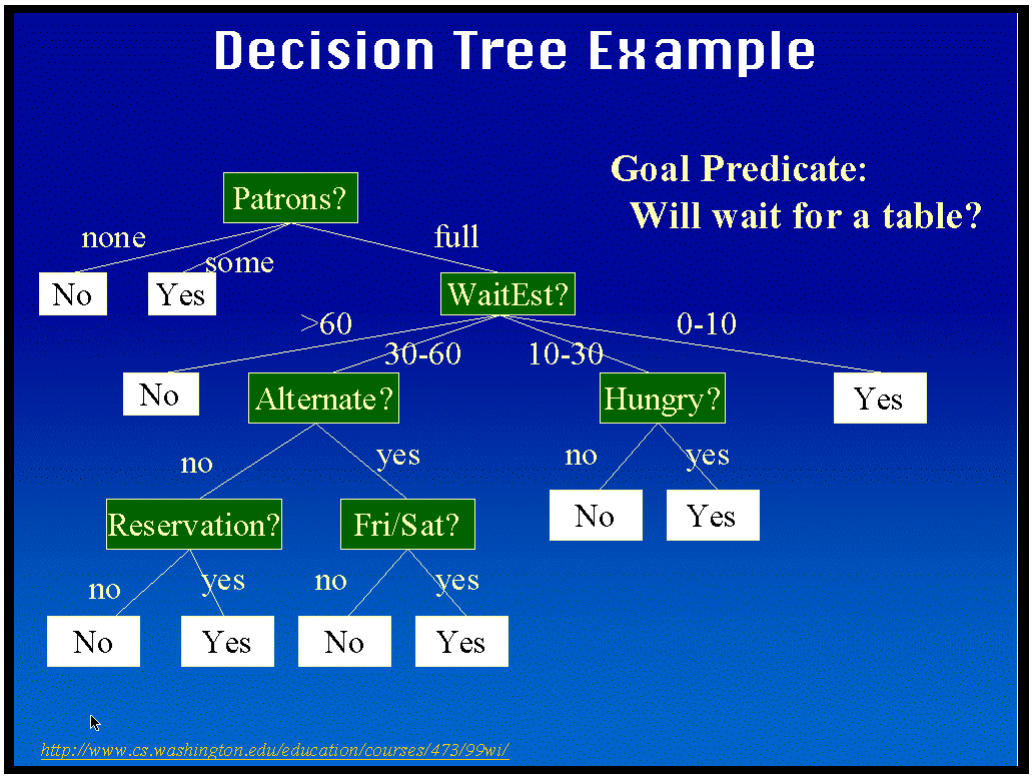

Figure 2: Decision Tree Example

  - Branches represent values of the attribute

- Path through a tree represents a decision
- Can be associated with rules

11 ID3 algorithm

- Builds a decision tree from a set of examples, each consisting of a set of attribute-value pairs, through a greedy approach.
- Summary:
  1. Calculate the entropy of every attribute using a data set
  2. Split the data set into subsets using the attribute for which the resulting entropy is minimum, or the information gain is maximum
  3. Make a decision tree node containing that attribute

4. Recurse on subsets using remaining attributes

- Answers the question: **Are we done yet?** Which means one of two things:
  - All of the data points to the same classification. Allows ID3 to make a final decision.
  - There are no more attributes to divide the data. ID3 only uses each attribute maximum one time per path through the tree. If maximum is reached and remaining data doesn't point to a classification, it is forced to make a final decision $\Longrightarrow$ Usually picks the most popular classification.

Part V
PEAS

- Performance Measure
- Environment
- Actuators
- Sensors

12 Performance Measure

- Be sade
- Reach Destination
- Maximize Profits
- Obey laws

13 Environment

- Urban streets
- Freeways
- Traffic
- Pedestrians
- Weather
- Customers

14 Actuators

- Steering wheel
- Accelerator
- Brake
- Horn

15 Sensors

- Video
- Accelerometers
- Gauges
- Engine sensors
- Keyboard
- GPS