COMP 424 - Artificial Intelligence
Lecture 1: Course Introduction and Uninformed search methods

Instructor: Joelle Pineau (jpineau@cs.mcgill.ca)
Today's presenter: Melanie Lyman-Abramovitch

Class web page: www.cs.mcgill.ca/~jpineau/comp424

Unless otherwise noted, all material posted for this course are copyright of the instructor, and cannot be reused or reposted without the instructor’s written permission.

Course objectives

- To develop an understanding of the basic concepts of AI: classes of problems, mathematical models, algorithms.

- To have an informed opinion about AI-related technologies.

- To interact with people working in the field.
Course topics

• Search
  - Basic tools
• Game playing
• Logical reasoning
  - Logical representations
• Classical planning
  - midterm
• Probabilistic reasoning
• Learning probabilistic models
• Reasoning with utilities
• Sequential reasoning and decision-making.
• Learning complex sequential decisions.
• Applications

About the course


• Evaluation:
  1. Individual quizzes in class (5%)
  2. Individual assignments (20%)
  3. Project: Implementation and written report (20%)
  4. Midterm (15%)
  5. Final exam (40%)

• Lecture notes (slides), assignments, solutions, project information, all available on class website.

• If all goes well, lectures will be recorded and available on myCourses.
Course Project

- Design and implement a **game playing program**.
- Use ideas from the course to design your algorithm.
- Basic code (for a random player) will be provided in Java.
- You can use any programming language.
- Enter your program in a tournament against programs from other students.
- Evaluation: 50% code and competition results, 50% written report.
  - Written report must be clear, complete, including appropriate references.
  - Code must compile and run.
- Project info will be posted as it becomes available on class webpage.

---

Coursework policy

- **Format of coursework**: Some programming, some theory, some problem-solving, some application.
- **Quizzes will be given in class**, can be solved collaboratively, but must be submitted individually on myCourses by 2pm on the same day.
- **Assignments are individual**, submitted through myCourses by 11:59pm on the due date.
- **Project** should be submitted also on myCourses. **Late submissions are subject to 20% penalty up to 5 days**; after this will not be accepted.
Coursework policy

- **Format of coursework**: Some programming, some theory, some problem-solving, some application.
- **Quizzes will be given in class**, can be solved collaboratively, but must be submitted individually on myCourses by 2pm on the same day.
- **Assignments are individual**, submitted through myCourses by 11:59pm on the due date.
- **Project** should be submitted also on myCourses. **Late submissions are subject to 20% penalty up to 5 days**; after this will not be accepted.
- Late quizzes or assignments will NOT be accepted.
- No make-up midterm will be offered.
- Marks for any incomplete item (incl. not submitted or incorrect work) except the project will be shifted to the final exam.

Prerequisites

Courses: (COMP 206 or ECSE 321) COMP 251.

At least one university level course in probability and/or statistics, e.g. MATH-203, MATH-323, ECSE-305.

Basic knowledge of a programming language is required. Basic knowledge of calculus and linear algebra is also assumed.
Expectations

- You must have taken the pre-reqs to register.
- Come to class prepared, do the readings, follow the lectures.
- Ask questions, be engaged in your learning, be willing to work hard.
- Keep up with the assigned coursework.
- Respect the coursework policy.
- Use technology (email, discussion boards) appropriately.

Questions?
Goal for today’s class

• Identify defining elements of generic search problems

• Review uninformed search algorithms
  1. Breadth-first search
  2. Uniform cost search
  3. Depth-first search
  4. Depth-limited search
  5. Iterative deepening

• Define criteria for evaluating search algorithms

Search in AI

• One of the first topic studied in AI:

• Central component to many AI systems:
  – Theorem proving
  – Game playing
  – Automated scheduling
  – Robot navigation

  Domain-specific problem representation + Generic search algorithm
  = Problem solution
Example: Eight-Puzzle

![Start State](image1)
![Goal State](image2)

How would you build an AI agent to solve this problem?

Defining a search problem
Defining a search problem

- **State space** $S$: all possible configurations of the domain.

- **Initial state** $s_0 \in S$: the start state

- **Goal states** $G \subseteq S$: the set of end states

- **Operators** $A$: the actions available
  - Often defined in terms of mapping from state to successor state.

---

Defining a search problem (cont’d)

- **Path**: a sequence of states and operators.

- **Path cost**, $c$: a number associated with any path.

- **Solution** of search problem: a path from $s_0$ to $s_g \in G$

- **Optimal solution**: a path with minimum cost.
Example: Eight-Puzzle

- States?
- Goals?
- Operators?
- Path cost?

Example: Eight-Puzzle

- States? Configurations of the puzzle.
- Goals? Target configuration.
- Operators? Swap the blank with an adjacent tile.
- Path cost? Number of moves.
Example: Robot path planning

More generally....

- **States?** Position, velocity, map, obstacles.
- **Goals?** Get to target position without crashing.
- **Operators?** Small steps in several directions.
- **Path cost?** Length of path, energy consumption, time to goal, ...
Example: Metric path planning

- States? Grid cells.
- Goals? Grid cell of yellow dot.
- Operators? Move to grid cells in 4 cardinal directions (except obstacles).
- Path cost? Number of grid cells visited.

Example: Grid Cells
Example: Topological path planning

- **States?** Nodes in the graph (= rooms).
- **Operators?** Follow edge to adjacent rooms.
- **Path?** Sequence of nodes.
- **Path cost?** Edge cost.

How to pick the right representation?
How to pick the right representation?

At a minimum, need enough information to represent all visited states.

State space representation? Number of states?

- List of cell addresses for each red piece + list of cell addresses for each white piece.
- For each cell address, whether it has a red piece or a white piece or nothing.
How to pick the right representation?

At a minimum, need enough information to represent all visited states.

State space representation? Number of states?

- List of cell addresses for each red piece +
  list of cell addresses for each white piece.
  \[ O(|S|) = (32+1 \text{ addresses})^{24 \text{ pieces}} \]
- For each cell address, whether it has a red
  piece or a white piece or nothing.
  \[ O(|S|) = (2+1 \text{ colours})^{32+1 \text{ addresses}} \]

Basic assumptions (for next few lectures)

- **Static** (vs dynamic) environment
- **Observable** (vs unobservable) environment
- **Discrete** (vs continuous) states
- **Deterministic** (vs stochastic) environment

The **general** search problem does not make these assumptions,
but most of the search algorithms discussed today require them.
Representing search: Graphs and Trees

- Visualize the state space search in terms of a graph.

- Graph defined by a set of vertices and a set of edges connecting the vertices.
  - Vertices correspond to states.
  - Edges correspond to operators.

- We search for a solution by building a search tree and traversing it to find a goal state.
**Example**

Search tree nodes are NOT the same as the graph nodes!

**Data structures for search tree**

- **Defining a search tree node:**
  - Each node contains a state id (from the states in the graph).
  - Node also contain additional information:
    - The parent state and the operator used to generate it.
    - Cost of the path so far.
    - Depth of the node.

- **Expanding a search tree node:**
  - Applying all legal operators to the state.
  - Generating nodes for all the corresponding successor states.
Generic search algorithm

- **Initialize** the search tree using the *initial state* of the problem

- **Repeat**
  1. If no candidate nodes can be expanded, return failure.
  2. Choose a node for expansion, according to some search strategy.
  3. If the node contains a *goal state*, then
     - return the corresponding *path*.
  4. Otherwise expand the node, by:
     - applying each applicable *operator*,
     - generating the *successor state*, and
     - adding the resulting nodes to the tree.

Coding a Generic Search Problem in Java

```java
public abstract class Operator {} 

public abstract class State {
    abstract void print();
}

public abstract class Problem{
    State startState;
    abstract boolean isGoal (State crtState);
    abstract boolean isLegal (State s, Operator op);
    abstract Vector getLegalOps (State s);
    abstract State nextState (State crtState, Operator op);
    abstract float cost(State s, Operator op);

    public State getStartState() { return startState; }
}
```
Coding an Actual Search Problem

public class EightPuzzleState extends State {
    int tilePosition[9];
    public void print() {
    }
}

public class EightPuzzleProblem extends Problem{
    boolean isLegal (EightPuzzleState s,
    EightPuzzleOperator op){
        // check if blank can be moved in the desired direction
    }
}

Specialize the abstract classes, and add the code that does the work

Example

Now expand a little further...
Example

Problem: Search trees can get very big!

Implementation details

• Need to keep track of the nodes to be expanded: the frontier.

• Implement this using a queue:
  1. Initialize queue by inserting a node for the initial state.
  2. Repeat
     a) If the queue is empty, return failure
     b) Dequeue a node.
     c) If the node contains a goal state, return path.
     d) Otherwise expand the node by applying all legal operators to the state.
     e) Insert the resulting nodes in the queue.

Search algorithms differ in their queuing function.
Uninformed (blind) search

- If a state is not a goal, you cannot tell how close to the goal it might be.
- Hence, all you can do is move systematically between states until you stumble on a goal.
  
  E.g.
  - Breadth-first search
  - Depth-first search

- In contrast, informed (heuristic) search uses a guess on how close to the goal a state might be. (More on this next class.)

Breadth-First Search (BFS)

- Enqueue nodes at the end of queue.
- All nodes at level i get expanded before all nodes at level i+1.
Example

- In what order are nodes expanded using Breadth-first search?

Example

- Label all start states as $V_0$. 

Example

- Label all successors of states in \( V_0 \) that have not been labeled as \( V_1 \).

![Diagram showing a network with states and transitions labeled with steps (0, 1, 2).

Example

- Label all successors of states in \( V_1 \) that have not been labeled as \( V_2 \).
Example

- Label all successors of states in $V_2$ that have not been labeled as $V_3$.

![Graph](image1)

Example

- Label all successors of states in $V_3$ that have not been labeled as $V_4$.

![Graph](image2)
Example

- **To recover the path:** Follow pointers back to the parent node.

![Diagram of a graph with nodes and arrows indicating path lengths]

Revisiting states

- What if we revisit a state that was already expanded?
- What if we visit a state that was already in the queue?
- Example:
Revisiting states

- Maintain a closed list to store every expanded node.
  - More efficient on problems with many repeated states.
  - Worst-case time and space requirements are $O(|S|)$ ($|S|$ = #states)

- In some cases, allowing states to be re-expanded could produce a better solution.
  - When repeated state is detected, compare old and new path to find lowest cost path.
  - In large domains, may not be able to store all states.

Depth-First Search (DFS)

- Enqueue nodes at the front of queue.
- Nodes at the deepest level get expanded before shallower ones.
Example

In what order are nodes expanded using Depth-first search?

Order of operators matter!

Solution *if you expand nodes in clockwise order, staring at 9 o'clock*: {Start, d, b, a, c, e, r, f, Goal}
Example

In what order are nodes expanded using Depth-first search?
Order of operators matter!

Solution (if you expand nodes in clockwise order, staring at 9 o’clock):
{Start, d, b, a, c, e, r, f, Goal}

What if we expand nodes counter-clockwise, from 9 o’clock?
Key properties of search algorithms

• Completeness:
• Optimality:
• Space complexity:
• Time complexity:
Key properties of search algorithms

- **Completeness**: Are we assured to find a solution, if one exists?
- **Optimality**: How good is the solution?
- **Space complexity**: How much storage is needed?
- **Time complexity**: How many operations are needed?

**Other desirable properties:**
- Can the algorithm provide an intermediate solution?
- Can an inadequate solution be refined or improved?
- Can the work done on one search be reused for a different set of start/goal states?
Search complexity

- Evaluated in terms of two characteristics:
  
  - Branching factor of the state space ("b"): how many operators (upper limit) can be applied at any time?

    *E.g. for the eight-puzzle problem: branching factor is 4, although most of the time we can apply only 2 or 3 operators.*

  - Solution depth ("d"): how long is the path to the closest (shallowest) goal state?

---

Key properties of **BFS and DFS**

- Completeness:
- Optimality:
- Space complexity:
- Time complexity:
Analyzing BFS

• **Good news:**
  – Complete.
  – Paths to different goals can be explored at the same time.
  – Guaranteed to find shallowest path to the goal if unit cost per step.

    *Will not necessarily find optimal path if cost per step is non-uniform.*

---

• **More bad news:**
  – Exponential time complexity: $O(b^d)$ [This is same for all uninformed search algorithms.]
  – Exponential space complexity: $O(b^d)$ [This is not good!]

---

*COMP-424: Artificial intelligence* 61 Joelle Pineau © 2015

---

*COMP-424: Artificial intelligence* 62 Joelle Pineau © 2015
Uniform Cost Search

- Goal: Fix BFS to ensure an optimal path with general step costs.

**Important distinction:**
- **Unit cost** = Problem where each action has the same cost.
- **General cost** = Actions can have different costs.

- Approach:
  - Use a **priority queue** instead of a simple queue.
  - Insert nodes in the increasing order of the **cost of the path** so far.

- Properties:
  - Guaranteed to find **optimal solution** for with **general step costs** (same as BFS when all operators have the same cost).
Example

• In what order are nodes expanded using Uniform cost search?

Example - solved

Priority queue = \{\{(Start, 0)\}\}
= \{\{(Start, 0), (p,1), (d,3), (e,9)\}\}
= \{\{(Start, 0), (p,1), (d,3), (e,9), (q,16)\}\}
= \{\{(Start, 0), (p,1), (d,3), (b,4), (e,5), (c,11), (q,16)\}\} [Find faster path to e.]
= \{\{(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (c,11), (q,16)\}\}
= \{\{(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (c,11), (r,14), (q,16)\}\}
= \{\{(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (c,11), (r,14), (q,16)\}\}
= \{\{(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (q,10), (c,11), (r,14)\}\}
= \{\{(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (q,10), (c,11), (r,13)\}\}
= \{\{(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (q,10), (c,11), (r,13), (f,18)\}\}
= \{\{(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (q,10), (c,11), (r,13), (f,18)\}\}
= \{\{(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (q,10), (c,11), (r,13), (f,18), (goal,23)\}\}
Analyzing DFS

• **Good news:**
  – Linear space complexity: $O(b^d)$
  – Easy to implement recursively (do not even need queue data structure).
  – More efficient than BFS if there are many paths leading to solution.

• **Bad news:**
  – Exponential time complexity: $O(b^d)$ [This is same as BFS]
  – Not optimal.
  – DFS may not complete!
  – NEVER use DFS if you suspect a big tree depth!
Depth-limited search

- **Algorithm**: search depth-first, but *terminate* a path either if a goal state is found, or *if the maximum depth allowed is reached*.

- **Always terminates:**
  - Avoids the problem of search never terminating by imposing a hard limit on the depth of any search path.

- However, it is still *not complete* (the goal depth may be greater than the limit allowed.)

---

Iterative Deepening Search (IDS)

- **Algorithm**: do depth-limited search, but with increasing depth.
- Expands nodes multiple times, but computation time has same complexity.
Analysis of IDS

• **Complete** (like BFS).
• Has **linear memory** requirements (like DFS).
• **Classical time-space tradeoff**.
  
  In AI, we are often concerned with *achieving rationality subject to resource constraints*. Will see similar trade-off often in this course.

• **Optimal** for problems with unit step costs.
• This is the **preferred method** for large state spaces, where the solution path length is unknown.
Questions

• Which method should you use if…. 
  
  – You need to find the optimal solution?
    • BFS, DFS or iterative deepening if unit cost.
    • Uniform-cost search if general cost.
  
  – The state space is VERY large?
    • Depth-first search if you know the maximum plan length.
    • Iterative deepening search otherwise.
  
  – You have limited memory?
    • Depth-first search / iterative deepening search.
  
  – You want to find quickly find the best solution within a cost budget?
    • Depth-limited search if unit cost.
    • Uniform-cost search if general cost.
Summary of uninformed search

• Assumes no knowledge about the problem.

• Main difference between methods is the order in which they consider states.
  – BFS
  – Uniform cost search
  – DFS
  – Fixed-depth DFS
  – Iterative deepening

• Very general, can be applied to any problem.

• Very expensive, since we assume no knowledge about the problem.

• Some algorithms are complete, i.e. they will find a solution if one exists.

• All uninformed search methods have exponential worst-case complexity.