EECS 3401 — AI and Logic Prog. — Lecture 13

Adapted from slides of Yves Lesperance

Vitaliy Batusov
`vbatusov@cse.yorku.ca`

York University

November 4, 2020

- Today: **Search Algorithms and Constraint Satisfaction**
- Required reading: Russell & Norvig Chapters 3.6, 4.1, 6

Recap from last time

- Searching in a state space for a sequence of transitions taking us from initial state to goal
- Frontier: a list of nodes to expand, at start contains just the initial state
- Choice of which node to expand defines the search strategy
- Formalized as: sort frontier by some criterion and always choose the first node
- Blind (uninformed) searches: BFS, DFS, IDS, UC (pros and cons)
- Informed search: guided by a heuristic function (domain-specific)
- Heuristics: **admissible**, **monotone**

Recap: Heuristics and A*

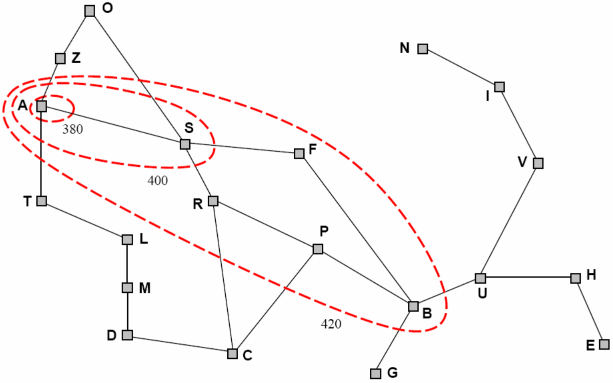
- **A* Search**: sort frontier by f-values (increasing)

  $$f(n) = g(n) + h(n)$$

  where $g(n)$ is the actual cost to get to n.
- h is **admissible** if $h(n) \leq h^*(n)$ for all n
- h is **monotone** if $h(n) \leq c(n \rightarrow n') + h(n')$ for all n, n', c
- Every monotone h is admissible
- Monotonicity makes A* amazing (relatively)

A* search with monotonicity

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Admissibility without monotonicity

What happens to the properties of A* when h is admissible but not monotonic?

- Time and space complexity remain the same
- Completeness still holds
- Without cycle checking, optimality still holds, but for a different reason
  Assume the goal path $\langle S, \ldots, G\rangle$ found by A* has cost $g(G)$ greater than the optimal cost C^*. Then, there must exist a node n in the optimal path that is still in the frontier. So:

  $$f(n) \quad = \quad g(n) + h(n) \quad \leq \quad g(n) + h^*(n) \quad = \quad C^* \quad < \quad f(G)$$

  If the f-value of n is smaller than that of the goal, then n would've been selected for expansion **before** G — a contradiction.

Admissibility without monotonicity

- No longer guaranteed to get an optimal path to a node **on first visit**.
- Cycle checking, as defined previously, will not preserve optimality
- To fix this, must remember cost of previous path. If new path is cheaper, must explore again.
- Monotonic contours no longer exist

Building Heuristics

How to come up with heuristic functions?

- A good approach: **simplify the problem**, let $h(n)$ be the cost of reaching the goal in the simplified version

- Example: 8-Puzzle
  In the original problem, can move a tile from square A to B if A is adjacent to B **and** B is empty.

  Can relax this in several ways:

  1. Ignore whether B is empty
  2. Ignore whether A is adjacent to B
  3. Ignore all preconditions on actions

Building Heuristics: Relaxing the Problem

- "Ignore all preconditions on actions" — known as the **misplaced tiles** heuristic
  - To solve the puzzle, need to move each tile in its final position
  - Number of required moves = number of misplaced tiles
  - Let $h(n) = \textit{(\# of misplaced tiles)}$. Clearly, this underestimates the actual cost

- "Ignore whether B is empty" — known as the **manhattan distance** heuristic
  - To solve the puzzle, need to slide each tile (in sequence of vertical or horizontal steps) to its proper place (different for each tile)
  - Number of required moves $= \sum_{t \in Tiles} manhattan\_distance(t)$
  - Let $h(n) = \sum_{t \in Tiles} manhattan\_distance(t)$. This also underestimates the actual cost

Building Heuristice: Relaxing the Problem

- The optimal cost to nodes in the relaxed problem is an **admissible heuristic** for the original problem
- Proof: The optimal solution in the original problem is also a solution for the relaxed problem. Therefore, it must be at least as expensive as the optimal solution in the relaxed problem.
- Real example: solving the 8-Puzzle using IDS, A*+misplaced, and A*+manhattan (average total nodes expanded)

Depth	**IDS**	**A*+mispl.**	**A*+manh.**
10	47127	93	39
14	3473941	539	113
24	—	39135	1641

Building Heuristice: Relaxing the Problem

Does *manhattan* always expand fewer nodes than *misplaced*?

- Yes. $h_{mispl}(n) \leq h_{manh}(n)$. We say that h_{manh} "dominates" h_{mispl}.
- **Among several admissible heuristics, the one with the highest value is the fastest**.

Building Heuristice: Pattern Databases

- Admissible heuristics can also be derived from solutions to **subproblems**
- *Each state is mapped into a partial specification*
- Example: in the 15-Puzzle, assume only the position of some specific tiles matters

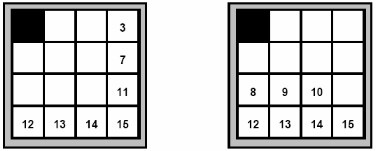


Fig. 2. The Fringe and Corner Target Patterns.

By searching backwards from these goal states, we can compute the distance of any configuration of these tiles to their goal locations. We are **ignoring** the identity of the other tiles, thus underestimating the effort in the general case.

Building Heuristice: Pattern Databases

- These configurations are stored in a database, along with the number of moves required to move the tiles into place
- The **maximum** number of moves taken over all of the databases can be used as a heuristic
- In the 15-Puzzle:
  - The “fringe” database yields about 300-fold decrease in the search tree size
  - The “corner” database yields about a 400-fold decrease
- Can also generate a database of **disjoint patterns** (mutually non-contradictory), so that the number of moves can be *added* rather than taking the maximum. This gives about a 10000-fold decrease compared to h_{manh}.

Assignment 2

To be posted by Friday

Local Search

- So far, we were concerned with findint a *path* to the goal
- For many problems, we don't care for the path—only want to find a goal state
- Examples: scheduling, IC layout, network optimization, etc.
- **Local search** algorithms operate using a single **current state** and generally move to neighbours of that state
- There is an **objective function** that tells the value of each state. The goal is defined as the state with the highest value (global maximum)
- Algorithms like *Hill Climbing* try to move to a neighbour with the highest value
- Danger of being stuck in local maximum. To deal with that, some degree of random exploration is added

Local Search Algorithms

- **Simulated Annealing**: instead of the best available move, take a random move and if it improves the situation then always accept, otherwise accept with a probability < 1. Progressively decrease the probability of accepting such moves.
- **Local Beam Search**: like a parallel version of Hill Climbing. Keeps K states and at each iteration chooses the K best neighbours (so information is shared between the parallel threads). Also stochastic version.
- **Genetic Algorithms**: similar to Stochastic Local Beam Search, but mainly use crossover operation to generate new nodes. This swaps feature values between the parent nodes to obtain children. This gives a hierarchical flavor to the search: chunks of solutions get combined. Choice of state representation becomes very important. Has had wide impact, but not clear if/when better than other approaches.

Constraint Satisfaction Problems

- The search algorithms we discussed so far had no knowledge of the state representation (black box)
- Couldn't take advantage of domain-specific information
- CSP are a special class of search problems with a **uniform and simple** state representation
- This allows to design more efficient algorithms

Constraint Satisfaction Problems

- Many problems can be represented as a search for a vector of feature values
    - k-features: variables
    - Each feature has a value from some domain
    - Example: $height = \{short, average, tall\}$, $weight = \{light, average, heavy\}$
- In such problems, the task is to search for a set of values for the features (variables) so that the values satisfy some given conditions (constraints)

Constraint Satisfaction Problems

Sudoku

- 81 variables: 9 by 9 grid, each cell needs a value
- Values: a fixed value for those cells that are already filled in, *some* value from the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for each of the empty cells
- Solution: a value for each cell satisfying the constraints:
  - No cell in the same column can have the same value
  - No cell in the same row can have the same value
  - No cell in the same sub-square can have the same value

Constraint Satisfaction Problems

Scheduling

Want to schedule a time and a space for each final exam, so that

- no student is scheduled to take more than one final exam at any one time
- The space allocated has to be available at the time chosen
- The space has to be large enough to accommodate all of the students taking the exam

Variables:

- $T_1, \ldots, T_m$: each T_i represents the scheduled time for the i-th final
- (Assume domains are fixed to something like $\{MonAM, MonPM, \ldots, FriPM\}$)
- $S_1, \ldots, S_m$: each S_i is the space variable for the i-th final
- (Domain of S_i is the set of all rooms big enough to hold the i-th final)

Constraint Satisfaction Problems

Want to find an assignment of values to each variable, subject to the constraints:

- For all pairs of finals i, j, such that there is a student taking both, want $T_i \neq T_j$
- For all pairs of finals i, j, want to have either $T_i \neq T_j$ or $S_i \neq S_j$.

CSP, Formally

A **Constraint Satisfaction Problem** (CSP) consists of:

- A set of variables $V_1, \ldots, V_n$
- For each variable V_i, a domain of possible values $Dom[V_i]$
- A set of constraints $C_1, \ldots, C_m$

- Each variable V_i can be assigned any value from $Dom[V_i]$
- Each constraint C has
  - A **scope**: a subset of the problem's variables which it concerns e.g., V_1, V_2, V_4
  - A boolean function that maps assignments to these variables to True/False, e.g.,
    $C(V_1 = a, V_2 = b, V_4 = c) = True$
    $C(V_1 = b, V_2 = c, V_4 = c) = False$

A **solution** to a CSP is an assignment of a value to all of the variables such that every constraint is satisfied.

Example: Sudoku

- Variables: $V_{1,1}, V_{1,2}, \ldots, V_{9,9}$
- Domains:
  $Dom[V_{i,j}] = \{1..9\}$ for empty cells,
  $Dom[V_{i,j}] = \{k\}$ for each cell pre-filled with some k
- Row constraints:
  $CR_1(V_{1,1}, V_{1,2}, \ldots, V_{1,9})$
  $CR_2(V_{2,1}, V_{2,2}, \ldots, V_{2,9})$
  ...
  $CR_9(V_{9,1}, V_{9,2}, \ldots, V_{9,9})$
- Column constraints:
  $CC_1(V_{1,1}, V_{2,1}, \ldots, V_{9,1})$
  etc.
- Sub-square constraints:
  $CSS_1(V_{1,1}, V_{1,2}, V_{1,3}, V_{2,1}, V_{2,2}, V_{2,3}, V_{3,1}, V_{3,2}, V_{3,3})$
  etc.

Example: Sudoku

- Each of these constraints is over 9 variables, and they are all the same constraint: all values must be unique
- Such constraints are often called the `ALL-DIFF` constraints
- Thus, Sudoku has 3 x 9 `ALL-DIFF` constraints, one over each set of variables in the same column, one over each set of variables in the same row, and one over each set of variables in the same sub-square
- Note: An `ALL-DIFF` constraint over k variables can be equivalently expressed by k-choose-2 not-equal constraints over each pair of these variables.
  Let NEQ be a not-equal constraint; then
  $CSS_1(V_{1,1}, V_{1,2}, V_{1,3}, V_{2,1}, V_{2,2}, V_{2,3}, V_{3,1}, V_{3,2}, V_{3,3}) =$
  $NEQ(V_{1,1}, V_{1,2}), NEQ(V_{1,1}, V_{1,3}), \dots, NEQ(V_{3,2}, V_{3,3})$

End of Lecture

- Next time: **Solving CSPs**