CSE 571 - FALL 99. HANDOUT 2. VERSION 0.2

Chitta Baral

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406 USA
chitta@asu.edu

http://www.public.asu.edu/~cbaral/cse571-f99/

September 16, 1999

SEARCH AND ITS ROLE IN PLANNING

Quick overview of search techniques

- Basic search procedure
  - 1. L := list of initial nodes
    (at any time L is list of nodes that have not been examined.)
  - 2. If L is empty FAIL, else, pick a node n from L.
  - 3. If n is a goal node STOP and return it and the path to it from an initial node.
  - 4. Otherwise, remove n from L, add n's children to L and keep their path information.

- Depth first search
  - In Step 4, n's children are added to the front of L (as in a stack).
  - In Step 2, the first node node in the list is picked.
  - Memory needed?
    Assume a constant branching factor b and a depth d. $\cong d(b-1)$
  - time: what if the depth is infinite?
- Breadth first search
  - In Step 4, n's children are added to the back of L (as in a queue).
  - In Step 2, the first node node in the list is picked.
  - Memory needed?
    Assume a constant branching factor b and a depth d. $\cong b^{d-1}$
  - time: is usually more than Depth first, but always reaches a solution if it exists.

- Iterative deepening
  - Do Depth First Search with an artificial depth cutoff of 1, 2 . . . until a solution is reached.
- Iterative broadening
  - Do depth first search with artificial breadth limits, increasing them until a solution is found.
- Searching graphs.
  - Open and Closed lists

    Closed list contains nodes that have been examined and removed from L.

    Open list is the list of nodes that still need to be examined.

- Heuristic search

  $h(s)$ – The heuristic estimate of how far s is from goal.

  $g(s)$ – The distance traveled to reach s from an initial node.

  - Hill Climbing: Keep walking uphill. (If you are at s now, and s is not a goal visit that child of s whose $h(s)$ is the minimum.)
  - Best-first search : At any point in the search pick the node with the minimum $h(s)$.

    Both may get stuck in local minimas.
  - Simulated annealing: Use hill climbing but occasionally (with a probability that is reduced over time) take a step which does not have the minimum $h(s)$.

    (the probability corresponds to temperature of metal during the cooling period in metal casting)

- A^*:
  - $f(n) = g(n) + h(n)$.
  - At any point in the search pick the node with the minimum $f(s)$.
  - Admissibility: Assuming h to be a non-negative function, and if for all n, $h(n) \leq h_{actual}(n)$, then A^* will always find an optimal goal node.
  - If $h(n) = 0$, then we have breadth first.
  - No admissible heursitics that mimics depth first.
  - Between two admissible heuristics, if one is consistently closer to the real value than the other, then it leads to lesser number of node expansions.
  - General problem of constructing admissible heuristics from non-admissible ones is open.
  - Memory is a problem in A^*. (Note: Breadth first is a special case of it.)

– Branch and Bound

If $g(n)$ just not depth but some cost, and if $h(n) = 0$, then we have 'branch-and-bound'.

- IDA^* (Iterative deepening A^*)
  * Like iterative deepening, but instead of pruning nodes that lie at a depth below some increasing cutoff, prune nodes for which the nearest goal node can be shown to lie below the cutoff depth.
  * 1. $c := 1$. (c is the current depth cutoff.)
  * 2. $L :=$ list of initial nodes in the problem.
  * 3. Let n be the first node on L. If L is empty, increment c and return to Step 2.
  * 4. If n is a goal node, stop and return it and a path from the initial node to n.
  * 5. Otherwise, remove n from L. Add to the front of L, every child n' of n, <u>for which $f(n') \leq c$</u>. Return to step 3.
  * Individual iteration of this algorithm uses simple depth first search. (not A^*)
    Heuristic function is used to prune nodes, not to determine the order in which they should be expanded.

- RTS (Real time search) [Korf AIJ 1990]
  - Perform limited amount of local search before committing and executing moves.
  - $c(a, s)$: cost (positive) of taking action a in state s.
  - s_a: state predicted to be reached after executing action a in state s.
  - $h(s)$: heuristic estimate of the distance of s from a goal.
  - The algorithm
    * 1. Evaluate each action a applicable in current state s as $Q(a, s) := c(a, s) + h(s_a)$
    * 2. Apply action $\mathbf{a}$ that minimizes $Q(\mathbf{a}, s)$, breaking ties randomly.
    * (3. Observe resulting state s'.)
    * 4. Exit if s' is a goal state, else set s to s' and go to Step 1.
  - If the action model is completely accurate, Step 3 can be omitted and s' can be replaced by the predicted state $s_{\mathbf{a}}$.
  - RTS may not yield optimal (i.e., minimum cost) solutions, even when the heuristic function h is admissible.

- LRTA* (Learning Real Time A^*)
  - Slight modification of RTS where $h(s)$ is used to provide the *initial* cost estimates only.
  - Every time an action a is selected and applied in s, the estimate denoted by $V(s)$ are updates as
    $V(s) := c(a, s) + V(s_a)$
  - LRTA* algorithm
    * 0. For all s, $V(s) := h(s)$.
    * 1. Evaluate each action a applicable in current state s as
      $Q(a, s) := c(a, s) + V(s_a)$
    * 2. Apply action **a** that minimizes $Q(\mathbf{a}, s)$, breaking ties randomly.
    * 3. $V(s) := Q(\mathbf{a}, s)$
    * (4. Observe resulting state s'.)
    * 5. Exit if s' is a goal state, else set s to s' and go to Step 1.

- Result: If $h(s)$ is admisible, consecutive trials of LRTA* eventually yield an optimal behavior.

  (assuming the action model is correct, the state space is finite, and goals are reachable from every state.)

  * Note $h(s) = 0$ is admissible.
    So it will eventually find the optimal heuristic values and the optimal moves.
  * The rate at which LRTA* finds these moves depends in general on the *quality* of h.

Search and planning: earlier methods

- Forward search (in the state space) from initial state to goal states.
  - S: current state, G: goal, $Path$: path from the initial state to the current state.
  - $Progression(S, G, Path)$
  - If S satisfies G then $return(Path)$.
  - Else, pick an action which is executable in S.
  - If no such action exsists return with Failure.
  - Else Call $Progression(S_a, G, Path \circ a)$, where a is the action picked S_a is the state that will be reached after executing a in S, and $\circ$ is concatenation.
  - Intial call: $Progression(initialstate, Goal, [])$

- Backward search (in the state space) from goals to initial state.
  - I: initial state, G: current goals, $Path$: path computed so far.
  - $Regression(I, G, Path)$
  - If I satisfies G then $return(Path)$.
  - Else, pick an action a that can be regressed from G. (i.e., it has an effect that is one of the conjuncts of G.)
  - If no such action exists return Failure.
  - Else, let G_{-a} is the result of regressing a from G.
  - Call $Regression(I, G_{-a}, a \circ Path)$
  - Initial Call: $Regression(initial\_state, goal, [])$
- Partial order planning: search in the space of plans.

Partial order planning

- $POP(\langle A, O, L\rangle, Agenda, AList)$
- A: A set of instantiated actions;
- O: A set of orderings of the form $a_1 < a_2$ between instantiated actions where $a_1 < a_2$ means a_1 occurs before a_2.
- L: A set of causal links of the forms $a_1 \xrightarrow{f} a_2$ which means a_1 should occur before a_2 to satisfy one of the preconditions f of a_2.
- Agenda: A collection of pairs $\langle Q, a\rangle$, where $a \in A$, and Q is a precondition of a. Intuitively it means that the algorithm must try to achieve Q so that a can be executed.
- Alist: A list of actions.
- **Step 1: (Termination)** If Agenda is empty then return $\langle A, O, L\rangle$.

- **Step 2: (Goal Selection)** Select a pair $\langle Q, a\rangle$ from the Agenda. (Not a backtracking point.)
- **Step 3: (Action Selection)** Choose an action a_{add} that adds Q either by instantiating an action in $Alist$ or from A which can be consistently ordered before a.

  If no such action exists, then return failure.
  Otherwise, $L' = L \cup \{a_{add} \stackrel{Q}{\rightarrow} a\}$,
  $O' = O \cup \{a_{add} < a, a_0 < a_{add}, a_{add} < a_\infty\}$, and $A' = A \cup \{a_{add}\}$.
- **Step 4: (Updating Goal state)** $Agenda' = Agenda \setminus \{\langle Q, a\rangle\}$
  If a_{add} is newly instantiated, then for each conjunct Q_i of its precondition add $\langle Q_i, a_{add}\rangle$ to $Agenda'$.
- **Step 5: (Causal Link Protection)** For every action a_t that might threaten a causal link $a_p \stackrel{R}{\rightarrow} a_c$, add a consistent ordering constraint, either
  (a) Demotion: Add $a_t < a_p$ to O' or (b) Promotion: Add $a_c < a_t$ to O'.
  If neither constraint is consistent, then return failure.

- **Step 6: (Recursive invocation)**
  $POP(\langle A', O', L' \rangle, Agenda', Alist)$.

Partial order planning with conditional effects

- $POP_c(\langle A, O, L \rangle, Agenda, AList)$
- **Step 1** and **Step 2** are as before.
- **Step 3: (Action Selection)** Choose an action a_{add} that adds or *(conditionally) adds* Q either by instantiating an action in $Alist$ or from A which can be consistently ordered before a. If no such action exists, then return failure. Otherwise, $L' = L \cup \{a_{add} \stackrel{Q}{\rightarrow} a\}$, $O' = O \cup \{a_{add} < a, a_0 < a_{add}, a_{add} < a_\infty\}$, and $A' = A \cup \{a_{add}\}$.
- **Step 4: (Updating Goal state)** $Agenda' = Agenda \setminus \{\langle Q, a \rangle\}$ If Q is a conditional effect and it has not been already used to establish a link of the form $a_{add} \stackrel{Q}{\rightarrow} A$, for some A, then for each conjunct Q_i in the condition add $\langle Q_i, a_{add} \rangle$ to $Agenda'$. If a_{add} is newly instantiated, then for each conjunct Q_i of its precondition add $\langle Q_i, a_{add} \rangle$ to $Agenda'$.

- **Step 5: (Causal Link Protection)** For every action a_t that might threaten a causal link $a_p \stackrel{R}{\rightarrow} a_c$, add one of the following consistent constraint
(a) Demotion: Add $a_t < a_p$ to O' or (b) Promotion: Add $a_c < a_t$ to O', or.
(c) Confrontation: If a_t's threatening effect is conditional on antecedent S, then non-deterministically choose some conjunct g from S and add $(\neg g, a_t)$ to $Agenda'$.
If none of the constraints are consistent, then return failure.
- **Step 6: (Recursive invocation)**
$POP_c(\langle A', O', L' \rangle, Agenda', Alist)$.

Planning using heuristic search

- Using domain dependent heuristics
- Using domain independent heuristics.

A quick recall of STRIPS

- Syntax: $\langle \mathcal{L}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$
  - State Language ($\mathcal{L}$): constants (fixed symbols) and relational symbols (fluent symbols).
    Fluent atoms are of the form $p(t_1, \ldots, t_k)$, where p is a relational symbol and t_is are terms.
    Example: on(a,b)
  - Operators ($\mathcal{O}$): Each operator *op* has a precondition, an add and a delete list given by a set of atoms.
  - Initial situation ($\mathcal{I}$): A set of fluent atoms.
  - Goal situation ($\mathcal{G}$): A set of fluent atoms.

- Semantics: From syntax to a precise interpretation (a model).

  Any STRIPS planning problem $P = \langle \mathcal{L}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ is given a precise meaning by mapping it into the interpretation $S(P) = \langle S, s_0, S_G, A, f \rangle$ where

  – the states S are sets of atoms from $\mathcal{L}$

  – s_0 is $\mathcal{I}$

  – The goals states are states s such that $\mathcal{G} \subseteq s$

  – $A(s)$ is the subset of operators $op \in \mathcal{O}$ such that $Prec(op) \subseteq s$.

  – The transition function f (which we earlier called $next$) is defined as follows:

    $f(a, s) = s \cup Add(a) \setminus Del(a)$, for $a \in A(s)$.

HSP: Heuristic search planner

- $h(s)$ – estimate of the distance from the state s to the goal.
- $g_s(p)$ – the cost estimate of making the fluent p true starting from s.
- $g_s(C)$ – the cost estimate for achieving the conjunction of atoms C from s.
- $g_s(C) = \Sigma_{r \in c}\, g_s(r)$
  (we are assuming here that subgoals are independent)
- $h(s) = g_s(Goal)$
- Computing $g_s(p)$: (we now refer $g_s(p)$ simply as $g(p)$)
  - For $p \in s$, $g(p) := 0$ and for all other p's $g(p) := \infty$
  - $s^* := s$

REPEAT

- For all op that are applicable in s^*
  - If $p \in Add(op)$ add p to s^* and
    if $C = Prec(op)$ then
    $g(p) := min[g(p), 1 + g(C)]$

UNTIL $g(p)$ values do not change.

- Intuitive meaning of $g_s(p)$:

  Assuming no delete list (or considering a relaxed problem) what is the minimum number of actions necessary to get to a state (starting from s) where p is true.

- Analyzing $h(s)$
  - $h(s)$ is not admissible. It may overestimate the cost.
  - An admissible (but less informative, based on experiments) heuristic is based on
    $g_s(C) = max_{r \in c} g_s(r)$.

(A refined version of this seem to be equivalent to Graphplan)

- Inefficiency: For each s, $h(s)$ needs to be computed separately. HSP-R tackles this inefficiency.

- HSP algorithm
  - Uses a variation of hill climbing
    - Since computing $h(s)$ is expensive HSP tries to get to the goal with as few node evaluations as possible.
  - plus limited number of 'plateau' moves, restarts, and
  - memory to avoid past states.

HSP-R : Heuristic regression planning

- Search backwards from the goal to the initial state.
- To decide which backward step to take use the heuristics
  $h(s) = \Sigma_{p \in s}\, g(p)$
  to estimate how close s is to the initial state.
- where, $g(p)$ is the cost estimate of making p true starting from an initial state.
- Note: Once $g(p)$ is computed for different fluents using the procedure described earlier, $h(s)$ can be easily computed.
- Regression space: state-space for the backward search:
  - The initial states and goal states switch their role.
  - For an operation op in the original problem, its reverse operation op^{-1} is said to be applicable in a state s if:

    * Its add list has atoms that belong to s. (i.e., it is *relevant*.) $Add(op) \cap s \neq \emptyset$
    * Its delete list does not remove atoms that are in s. (i.e., it is *consistent*.) $Del(op) \cap s = \emptyset$.
  - The transition function $f(op^{-1}, s)$ is defined as $s \cup Prec(op) \setminus Add(op)$
  - There may be states in the regression space that are not reachable from the original initial state.
  - A plan is a sequence of actions $a_0^{-1}, \ldots, a_n^{-1}$ such that it is a solution of the regression space and the reverse action sequence $a_n, \ldots, a_0$ is a solution of the original problem.

- Mutexes
  - Two atoms p and q are said to form a *mutex* $\langle p, q \rangle$ (i.e, are mutually exclusive) when they both can not be true in a state

reachable from s_0.

- Example: $\langle on(a,b), on(a,c)\rangle$
- Defining Mutex set

  A set M is a *mutex set* given a set of operators O and an initial state s_0 iff for all $\langle p,q\rangle$ in M

  * p and q are not both true in s_0.
  * For every $op \in O$ such that $p \in Add(op)$, we have $q \in Del(op)$, or $q \notin Add(op)$ and for some $r \in Prec(op)$, $\langle r,q\rangle \in M$.
  * For every $op \in O$ such that $q \in Add(op)$, we have $p \in Del(op)$, or $p \notin Add(op)$ and for some $r \in Prec(op)$, $\langle r,p\rangle \in M$.
- Exercise: Show that if $\langle p,q\rangle \in M$ then there does not exist a reachable state from s_0 that contains both p and q.
- Exercise: Show that if M_1 and M_2 are mutex sets then $M_1 \cup M_2$ is a mutex set.
- Exercise: Find the complexity of computing the Maximal mutex set.
- Computing an approximation of the *Maximal* mutex set.

  - $M_0 := \{\langle p, q \rangle :$ there exists some $op \in O, (p \in Add(op)$ and $q \in Del(op))$ or $(q \in Add(op)$ and $p \in Del(op))\}$
  - Iteratively remove all bad pairs from M_0.

- HSP-R algorithm
  - Greedy Best First Search:
    Uses cost function $f(n) = g(n) + W.h(n)$, $W \geq 1$.
  - They use $W = 5$.
  - GBFS vs BFS
    - Before selecting a node n with minimum f-cost from OPEN, check whether a minimum f-cost child n'' of the last node n' expanded is 'good enough'.
    - If so, GBFS selects any such child, else it selects n as in BFS.
    - n'' is good enough if $h(n'') < h(n')$

HSP and HSP-R vs Graphplan

MDPs and POMDPs

Solving MDPs using Real time dynamic programming (RTDP)

Solving POMDPs using RTDP