Distributed Multiagent Systems

Boi Faltings

Laboratoire d'Intelligence Artificielle
boi.faltings@epfl.ch
http://moodle.epfl.ch/

Fall 2018

Degrees of interaction

- Social laws: no message exchange
- Coordination protocols: message passing
- Explicit cooperative planning: exchange of plan fragments

Social laws

- Common rules that all agents follow to avoid conflicts
- Example: Traffic laws
  - drive on the right
  - at crossings, traffic from left has right of way

$\Rightarrow$ no collisions, even though drivers do not explicitly consider each other's actions

- Similar examples in nature: flocks of birds, insect colonies, etc.

Generating social laws

- Laws must still allow agents to achieve any goal
- Formally: exists sequence of transitions to move between any pair of a set of *focal* states ($\subseteq$ all states)

$\Rightarrow$ finding useful social laws is NP-complete in number of states (not just focal states)

- Remember: number of states is already very large!
  $\Rightarrow$ not a paradigm for programming agent systems in general.

Multi-agent learning

Many situations repeat themselves:

- accessing resources, e.g. in wireless communication
- task allocation, e.g. distributing mail
- stock and commodity trading
- selling ice cream

Agents *learn* to coordinate their behavior.

Equilibrium

Optimal action also depends on other agents' strategies:

- A combination of strategies is in *equilibrium* if each strategy takes the optimal action given the other agents' strategies
- There can be multiple equilibria
- A learning algorithm is *no-regret* if the strategy it learns will eventually have performance equal to the best possible (deterministic) strategy
- Example: Q-learning with infinite samples is no-regret
- Theorem: under mild conditions (smoothness), agents using no-regret learning will converge to an equilibrium

Example

- 2 agents A and B want to repeatedly transmit data on frequencies 1 and 2
- Action space $= (1, 2)$, if both choose the same, they collide and fail
- 2 Equilibria:
  1. $(A, 1)$ and $(B, 2)$
  2. $(A, 2)$ and $(B, 1)$
- Both start out with the same channel 1 $\Rightarrow$ reward $= 0$
- If once they choose different channels, both will get higher rewards $\Rightarrow$ for each agent, this choice will have better Q-value and thus be chosen more and more often in the future

Confidence bounds

Other agents' strategies are unknown $\Rightarrow$ learning with uncertainty

- To learn optimal strategy, need to explore effects of all different actions
- Remaining uncertainty characterized by a *confidence bound* on the *expected* regret
- Generally interested in *upper* confidence bound (UCB) for each action (as in multi-armed bandits)
- However, poor convergence since opponent play is not stationary

Anti-coordination

- Agents have to learn *different* strategies (like in channel allocation)
- Very common scenario
- Difficult to learn because there are many different optimal strategies

Courteous learning

- Suppose that agents can also observe other agents' actions
- Courteous rule: agents that have converged on a strategy no longer change, and
- Agents that have not converged will not use conflicting actions

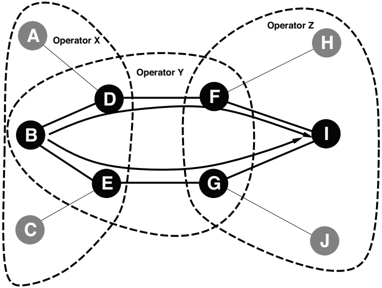
$\Rightarrow$ Fast convergence in $n \log n$ time (Cigler & Faltings 2011)

Distributed contract nets

- Explicit communication allows more complex protocols.
- Example: task allocation with contract nets.
- Each manager distributes tasks asynchronously and contacts agents directly.
- Market-based contract net does not work for such asynchronous settings.

Problems with distributed contract nets

- First come, first served
  ⇒ impossible to resolve conflicts.



- B → D → F → I
  may block A → D → F → H

⇒ idea: exchange tasks among agents to optimize allocation.

Marginal costs

Marginal cost to A_i of task t given a remaining set of tasks T:

$$c_{add}(A_i, t) = cost(A_i, T \cup t) - cost(A_i, T)$$

Principle:

- agent A_i *announces* t with limit $c < c_{add}(A_i, t)$
- agent A_j *bids* for t with bid $b > c_{add}(A_j, t)$
- t is reassigned to A_j if it is the lowest bid and $b < c$, agent A_i pays b to A_j

Implementation

- Agents look for tasks t that have particularly high marginal cost.
- Find other agents A_j that may have lower marginal cost.
- Announce the task to these agents.
- Agents A_j place bids for qualifying tasks and wait for decision.

Issues for announcers

- where to announce task?
- how long to wait until picking a winner?
- how to decide whether bid is still profitable?

$\Rightarrow$ requires knowledge of other agents' capabilities and expected costs

Issues for bidders

- how to bid considering outstanding bids and announced tasks?
- marginal cost depends on other tasks
- large risks while offers have not been answered

$\Rightarrow$ very difficult to even manage the messages, almost impossible to guarantee convergence
Need a more systematic way to solve such problems

General coordination

- Task allocation = for each task, decide what agent does it.
- Resource sharing = for each resource, decide which agent gets it at a certain time.
- Scheduling = deciding when agents do their tasks.
- All can be expressed as *constraint satisfaction*.
- Distributed coordination = distributed constraint satisfaction.
- Systematic approach with provable properties.

Constraint Satisfaction Problems (CSP)

Given $< X, D, C, R >$:

- variables $X = x_1, \ldots, x_n$
- domains $D = d_1, \ldots, d_n$
- constraints $C = c_1(x_{i,1}, x_{k,1}), \ldots, c_m(x_{i,m}, x_{k,m})$
- relations $R = (r_1 = \{(v_1, v_2), (v_3, v_4), \ldots\}, \ldots, r_m = \{(v_o, v_p), (v_q, v_r), \ldots\})$,

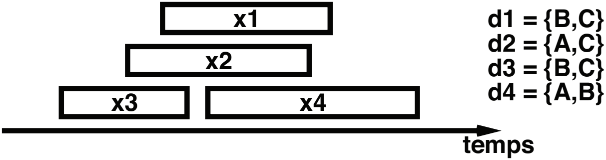
Find solution $= (x_1 = v_1 \in d_1, \ldots, x_n = v_n \in d_n)$ such that for all constraints, value combinations are allowed by relations

Can express most NP problems

Example of a CSP: Resource Allocation

Goal: assign ressources to tasks T1 - T4:

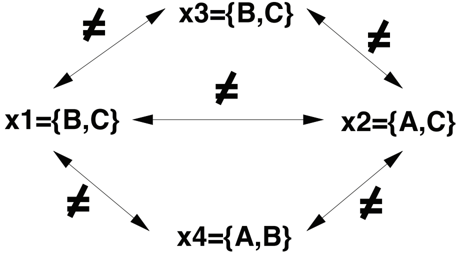
.



Resource Allocation (2)

CSP model:

- Variables = Tasks
- Domains = Resources that can carry out the task
- Constraints = between each pair of tasks that overlap in time
- Relations = inequality relations



Solving a CSP

Importance of CSP: large theory and tools for computing solutions.
Common methods:

- backtrack search: assign one variable at a time, backtrack when no assignment without satisfying constraints
- dynamic programming: eliminate variables and replace by constraints until a single one remains
- local (parallel) search: start with random assignment, make changes to reduce number of constraint violations

Distributed CSP (DCSP)

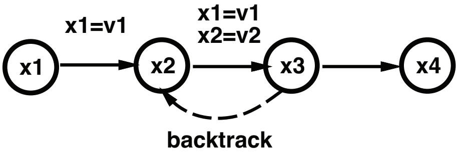
- Problem is distributed in a network of *agents*
- Each variable *belongs* to one agent
- Constraints are known to all agents with variables in it
- Distributed $\neq$ parallel: distribution of variables to agents cannot be chosen to optimize performance

Algorithms for solving DisCSP

1. Distributed backtracking:
   - synchronous
   - asynchronous
2. Dynamic programming
3. Local search

All algorithms require an *ordering* of agents.

Synchronous Backtracking



1. first agent generates a partial solution for x1, k=2
2. k-th agent generates an extension to this partial solution
3. if solution cannot be extended, k=k-1
4. if solution can be extended, k=k+1
5. if $k < 1$, stop: unsolvable
6. unless k > n, goto 2
7. solution = current assignment

Improvements

Synchronous backtracking allows common CSP heuristics:

- forward checking: partial instantiations extended to future agents
- dynamic variable ordering: select next variable according to domain size

$\Rightarrow$ strong efficiency gains

Implementing CSP heuristics

Distributed forward checking:

- $A(x_k)$ sends $(x_1 = v_1, .., x_k = v_k)$ to all $A(x_j)$, $j > k$
- $A(x_j)$ initiates backtrack at x_k whenever domain becomes empty

Dynamic variable ordering:

- $A(x_j)$ sends back size of remaining domain for x_j
- $A(x_k)$ chooses smallest one to be x_{k+1}

Asynchronous Backtracking

- Agents work in parallel without synchronization
- Global priority ordering among variables (ex.: unique processor id); assume x_i has higher priority than x_j whenever $i < j$
- Asynchronous message delivery, but all messages arrive in order in which they were sent
- Performance similar to synchronous backtracking

Distributed Monte-Carlo search

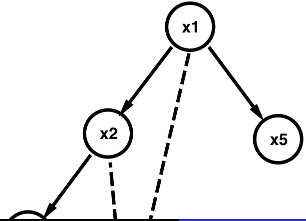
- Monte-Carlo search: search for an optimal solution by generating candidates randomly and observing their quality.
- Deliberative agent: search in 2 phases:
  1. cost estimation using random sampling
  2. value assignment picking the values that seem best
- Different branches of a tree are independent: sampling can run in parallel.
- Generalize to constraint graphs with cycles by using a pseudotree ordering.

Pseudotrees

Depth-first search traversal:

- move to neighbour not yet visited
- connect neighbours already in graph by *back edges*
- backtrack when no new neighbour

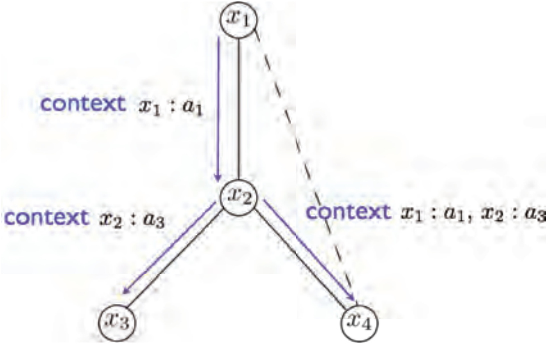
All edges connect to ancestors
$\Rightarrow$ no edges *between* nodes in different branches!

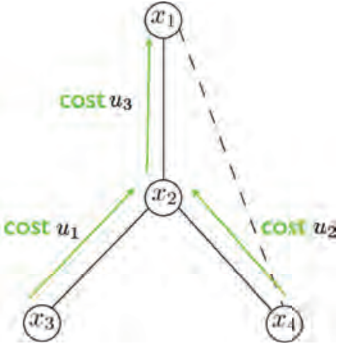


Cost Estimation

- Each variable receives a *context* from its ancestors.
- For each context, samples different values for its own variable and forwards to its descendants.
- Generalize from *conflicts* to *cost* of constraint (violations).
- Leaf nodes compute cost and send up to direct ancestor.
- Ancestor forms averages of samples and sends up to its own ancestor.

Cost Estimation(2)

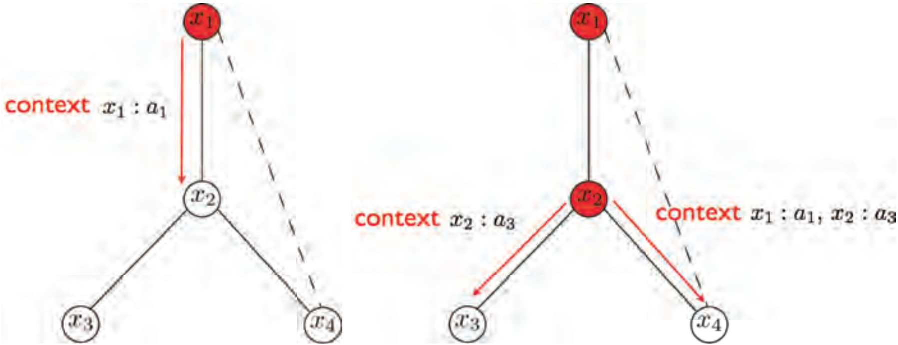




Value Assignment

- Root picks optimal value and sends to descendants as value contexts.
- Descendants pick optimal values depending on the context received from ancestors and results of Monte-Carlo sampling.

Value Assignment (2)



Distributed UCT

- DUCT algorithm implements distributed Monte-Carlo search.
- Uses random sampling controlled by multi-armed bandit model.
- Model = upper confidence bound in trees (as in game tree search)
- Orders of magnitude faster than systematic search.

Problems with backtrack search

- Every step in the search requires at least one message $\Rightarrow$ number of messages grows exponentially with variables
- Message delivery is much slower than computation $\Rightarrow$ process does not scale to large problems
- Better: fewer large messages

Dynamic Programming

- Principle: replace variables by constraints
- Consider variable x having constraint with y
- For each value of x, there may be a consistent value of y

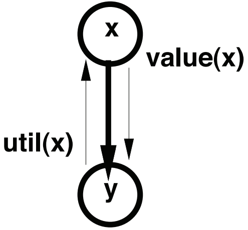
$\Rightarrow$ replace y by a constraint on x:

x=v is allowed if there is a consistent value of y

- Optimization version:

utility(x=v) = utility(x=v,y=w);
w = best possible value of y given x=v

Example



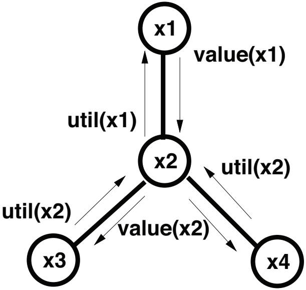
- y sends constraint in util(x) message

$\Rightarrow$ x can decide (best) value locally

- x informs y of value using value(x) message

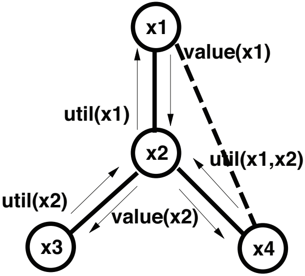
Dynamic programming in trees

- Rooted tree: every node has exactly one parent
- Nodes send util messages to their parents
- Best values of x3, x4 $\Rightarrow$ unary constraint on x2
- x2 sums up util messages + own constraint $\Rightarrow$ unary constraint on x1
- x1 picks best value v(x1); sends value(x1=v(x1)) $\rightarrow$ x2
- x2 picks best value given x1 and informs x3,x4



Dynamic Programming in Graphs

- Pseudo-trees: util messages are for all values in the context, not just the parent.
- Two messages per variable (util and value) ⇒ *number* of messages grows linearly with the size of the problem
- However, maximum message *size* grows exponentially with the tree-width of the induced graph (maximum number of backedges)



Dynamic Programming in Graphs

- Generalization to Pseudo-trees: UTIL messages are for all values in the context, not just the variable.
- Two messages per variable (Util and Value)

⇒ *number* of messages grows linearly with the size of the problem

- However, the maximum message *size* grows exponentially with the tree-width of the induced graph
(maximum number of backedges)
- In many distributed problems, the tree-width is relatively small

Distributed local search

Local search:

- initialize variables to arbitrary values
- iteratively make local improvements
- stop when no more improvements are found

Advantages: simple to implement, low complexity
Disadvantage: incomplete, usually only gets within 2-3% of the best solution

Min-conflicts

- Assign random value to each variable in parallel (this will conflict with some constraints)
- At each step, find the change in variable assignment which most reduces the number of conflicts
- Corresponds to search by "hill-climbing"

Distributed min-conflicts

- *Neighbourhood* of $N(x_i)$ = variables connected to x_i through constraints
- Change to x_i can happen asynchronously with others as long as there is no other change in the neighbourhood

⇒ two neighbouring agents are not allowed to change simultaneously:
  - highest improvement wins
  - ties broken by fixed ordering

⇒ parallel, distributed execution

Example: resource allocation

Variables:

$$x_1 \in \{B, C\}$$
$$x_2 \in \{A, C\}$$
$$x_3 \in \{B, C\}$$
$$x_4 \in \{A, B\}$$

Constraints:

$$C(x_1, x_2) : \{(B, A), (B, C), (C, A)\}$$
$$C(x_1, x_3) : \{(B, C), (C, B)\}$$
$$C(x_1, x_4) : \{(B, A), (C, B), (C, A)\}$$
$$C(x_2, x_3) : \{(A, B), (A, C), (C, B)\}$$
$$C(x_2, x_4) : \{(A, B), (C, A), (C, B)\}$$

$\Rightarrow$ neighbourhoods:

$$N(x_1) = \{x_2, x_3, x_4\}$$
$$N(x_2) = \{x_1, x_3, x_4\}$$
$$N(x_3) = \{x_1, x_2\}$$
$$N(x_4) = \{x_1, x_2\}$$

Example (min-conflicts)

Initial assignment:
(x1 = B, x2 = A, x3 = B, x4 = A)
$\Rightarrow$ 2 conflicts: c(x1,x3) et c(x2,x4)
1st step:

change	conflicts	nconf
x1 $\rightarrow$ C	c(x2,x4)	1
x2 $\rightarrow$ C	c(x1,x3)	1
x3 $\rightarrow$ C	c(x2,x4)	1
x4 $\rightarrow$ B	c(x1,x3),c(x1,x4)	2

Accept $x_1 \rightarrow C$, changes to x_2, x_3 and x_4 blocked because of neighbourhood
(Possible simultaneous change: x_3 and x_4)

Example (min-conflicts)...

(x1 = C, x2 = A, x3 = B, x4 = A)
$\Rightarrow$ 1 conflict: c(x2,x4)
2nd step:

change	conflicts	nconf
x1 $\rightarrow$ B	c(x1,x3), c(x2,x4)	2
x2 $\rightarrow$ C	c(x1,x2)	1
x3 $\rightarrow$ C	c(x1,x3),c(x2,x4)	2
x4 $\rightarrow$ B	-	0

accept (x4 $\rightarrow$ B) $\Rightarrow$ solution:
(x1 = C, x2 = A, x3 = B, x4 = B)

Asynchronous assignments

Basic procedure for assigning values:

1. select value $x_i = v_j$
2. send $OK?(x_i = v_j)$ message to each neighbour
3. receive $OK(x_k = ..)$ message from each neighbour x_k

$\Rightarrow$ each agent knows the values of its neighbours

Asynchronous changes

If conflicts:

1. Agent view $\Rightarrow$ find best possible *improvement* by changing own value
2. broadcast *improvement* to neighbours
3. receive improvements from neighbours

evaluate if:

- own improvement $>$ every neighbour x_j's, or
- own improvement $\geq$ every neighbour x_j's and x_i has higher priority than every x_j with equal improvement

$\Rightarrow$ assign different value if condition is satisfied

Example 2 (min-conflicts)

Initial assignment:

$(x1 = B, x2 = A, x3 = B, x4 = A)$

$\Rightarrow$ 2 conflicts: c(x1,x3) et c(x2,x4)

1st step:

change	conflicts	nconf
$x1 \rightarrow C$	c(x2,x4)	1
$x2 \rightarrow C$	c(x1,x3)	1
$x3 \rightarrow C$	c(x2,x4)	1
$x4 \rightarrow B$	c(x1,x3),c(x1,x4)	2

accept $(x2 \rightarrow C)$

Example 2 (min-conflicts)...

$(x1 = B, x2 = C, x3 = B, x4 = A)$
$\Rightarrow$ 1 conflict: c(x1,x3)
2nd step:

change	conflicts	nconf
$x1 \rightarrow C$	c(x1,x2)	1
$x2 \rightarrow A$	c(x1,x3),c(x2,x4)	2
$x3 \rightarrow C$	c(x2,x3)	1
$x4 \rightarrow B$	c(x1,x3),c(x1,x4)	2

no improvement possible: local minimum!

Breakout Algorithm

- Similar to min-conflict, but assign dynamic priority to every conflict (constraint), initially $=1$
- Modify variable which reduces the most the sum of the priority values of all conflicts.
- When local minimum:

  increase weight of every existing conflict

  Eventually, new conflicts will have lower weight than existing ones $\Rightarrow$ breakout

Local minima

If all improvements $= 0$:

1. increase weight of all constraint violations
2. restart asynchronous changes

Termination detection

- If constraint violation: $t - count \leftarrow 0$
- If no constraint violation: $t - count \leftarrow t - count + 1$
- Send $t - count$ to neighbours
- When receiving $t - count_j$ from another agent:
  $t - count \leftarrow min(t - count, t - count_j)$
- Termination when $t - count > d$, $d =$ max. distance of any agent
- Requires synchronous communication with time bounds

Example (Distributed Breakout)

- Assume initial choice = local minimum:
  (x1 = B, x2 = C, x3 = B, x4 = A)
  1 conflict $c(x1, x3)$
- A1: $x1 \rightarrow C : c(x1, x2); \textit{improvement} = 0$
  A2: $t - count \leftarrow 1$
  A3: $x3 \rightarrow C : c(x2, x3); \textit{improvement} = 0$
  A4: $t - count \leftarrow 1$

$\Rightarrow$ local minimum for $A1, A3$

$\Rightarrow$ increase weight of existing conflict $c(x1, x3)$

Example (Distributed Breakout)...

Increased weight $\rightarrow$ conflict weight $= 2$
A1: $x1 \rightarrow C : c(x1, x2); improvement = 1$
A2: $t - count \leftarrow min(1, 0) = 0$
A3: $x3 \rightarrow C : c(x2, x3); improvement = 1$
A4: $t - count \leftarrow min(1, 0) = 0$
$\Rightarrow$ A1 higher in priority order
$\Rightarrow$ accept change x1 $\Rightarrow$ C

Example (Distributed Breakout)...

- (x1 = C, x2 = C, x3 = B, x4 = A)
  1 conflict $c(x1, x2)$
- A1: $x1 \rightarrow B : c(x1, x3); improvement = -1$
  A2: $x2 \rightarrow A : c(x2, x4); improvement = 0$
  A3: $t - count \leftarrow 1$
  A4: $t - count \leftarrow 1$
- local minimum for A1,A2
- increase weight of existing conflict $c(x1, x2)$

Example (Distributed Breakout)...

- Increased weight $\rightarrow$ conflict weight $= 2$
- A1: $x1 \rightarrow B : c(x1, x3); improvement = 0$
  A2: $x2 \rightarrow A : c(x2, x4); improvement = 1$
  A3: $t - count \leftarrow min(1, 0) = 0$
  A4: $t - count \leftarrow min(1, 0) = 0$

$\Rightarrow$ A2 higher improvement

$\Rightarrow$ accept change x2 $\Rightarrow$ A

Example (Distributed Breakout)...

- (x1 = C, x2 = A, x3 = B, x4 = A)
  1 conflict $c(x2, x4)$
- A1: $t - count \leftarrow 1$
  A2: $x2 \rightarrow C : c(x1, x2); improvement = -1$
  A3: $t - count \leftarrow 1$
  A4: $x4 \rightarrow B : consistent; improvement = 1$

$\Rightarrow$ change $x4 \rightarrow B$

Detecting Termination

A1: $t - count \leftarrow 1 \leftarrow 2 > d$
A2: $t - count \leftarrow 1 \leftarrow 2 > d$
A3: $t - count \leftarrow 1 \leftarrow 2 \leftarrow 3 > d$
A4: $t - count \leftarrow 1 \leftarrow 2 \leftarrow 3 > d$
$\Rightarrow$ solution: (x1 = C, x2 = A, x3 = B, x4 = B)

Summary

- Distributed coordination = no central coordinator.
- Social Laws rarely feasible.
- Distributed Contract Nets: problems with convergence
- Distributed Constraint Satisfaction
  - Backtrack search algorithms
  - Dynamic Programming
  - Local search