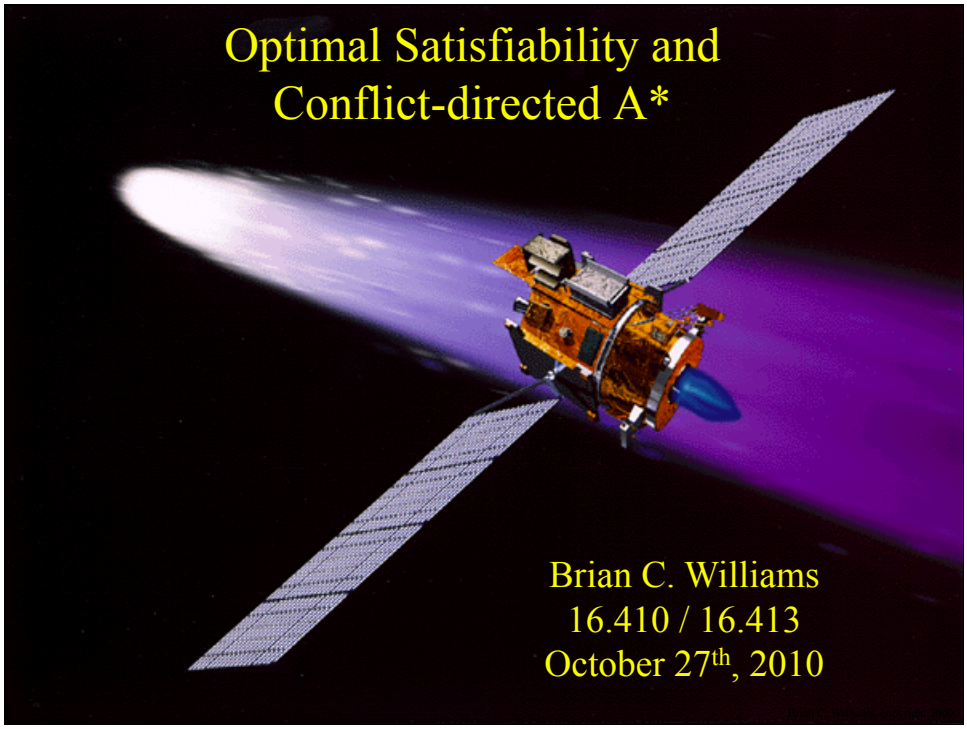


Image credit: NASA.

Assignment

- Remember:
  - Problem Set #6 Propositional Logic, due Today.
  - 16:413 Project Part 1: Sat-based Activity Planner, due Wednesday, November 3rd.
  - Problem Set #7 Diagnosis, Conflict-directed A* and RRTs, due Wednesday, November 10th.
- Reading
  - Today: Brian C. Williams, and Robert Ragno, "Conflict-directed A* and its Role in Model-based Embedded Systems," Special Issue on Theory and Applications of Satisfiability Testing, *Journal of Discrete Applied Math*, January 2003.

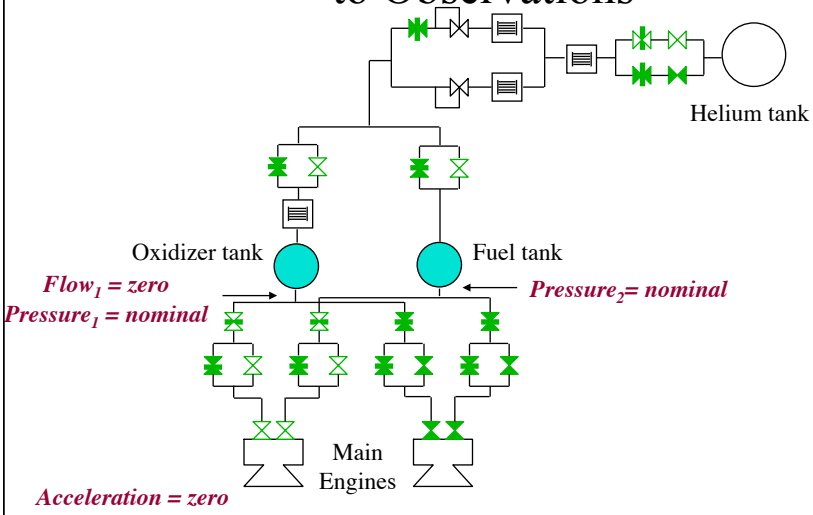
Model-based Diagnosis as Conflict-directed Best First Search

When you have eliminated the impossible, whatever remains, however improbable, must be the truth.

- Sherlock Holmes. The Sign of the Four.

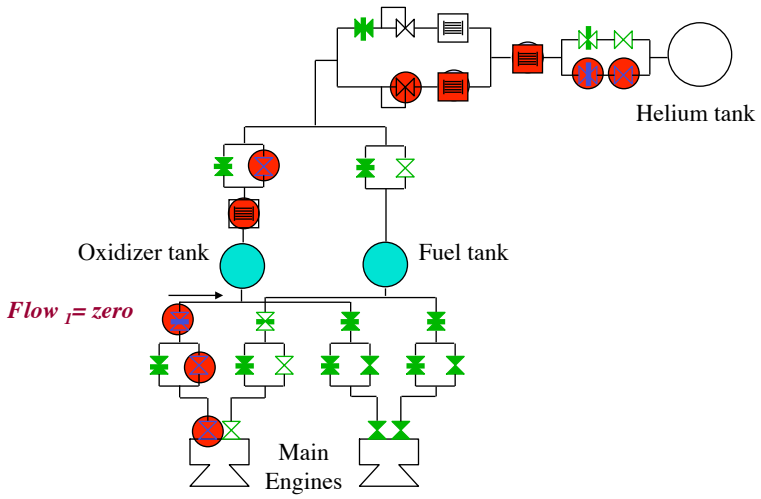
1. Generate most likely Hypothesis.
2. Test Hypothesis.
3. If Inconsistent, learn reason for inconsistency (a Conflict).
4. Use conflicts to leap over similarly infeasible options to next best hypothesis.

Compare Most Likely Hypothesis to Observations



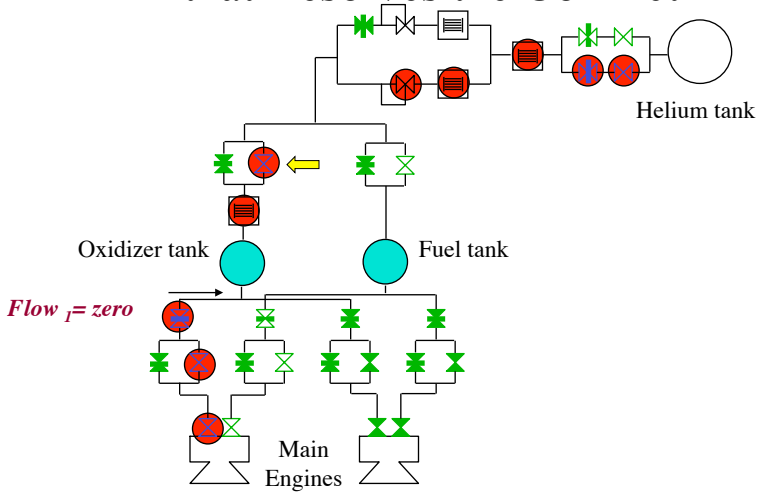
It is most likely that all components are okay.

Isolate Conflicting Information



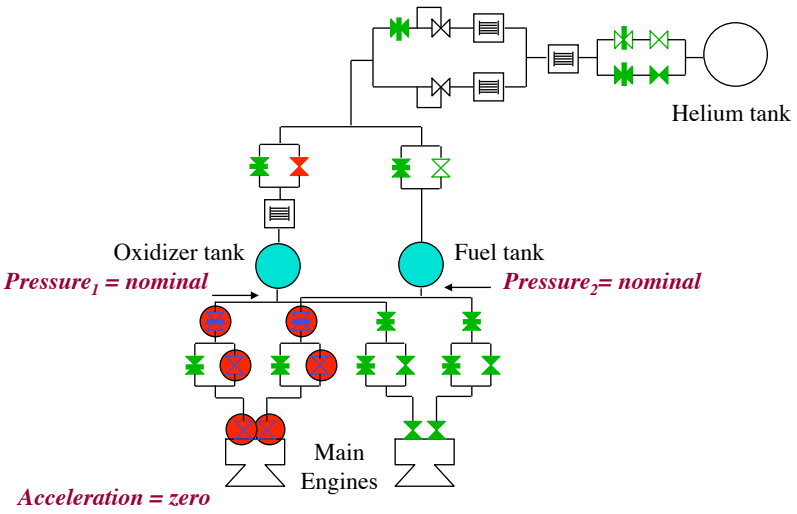
The red component modes *conflict* with the model and observations.

Leap to the Next Most Likely Hypothesis that Resolves the Conflict



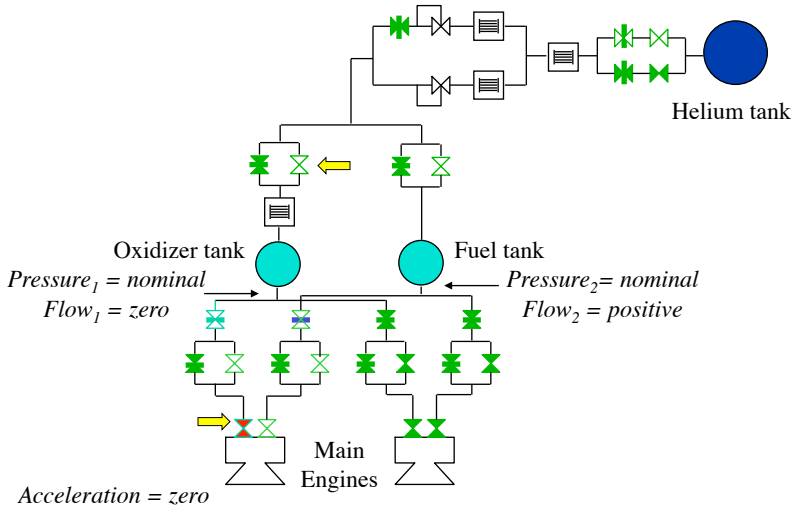
The next hypothesis must remove the conflict.

New Hypothesis Exposes Additional Conflicts



Another conflict, try removing both.

Final Hypothesis Resolves all Conflicts



Implementation: Conflict-directed A* search.

Outline

- Model-based Diagnosis
- Optimal CSPs
- Informed Search
- Conflict-directed A*

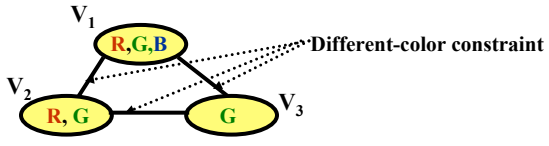
Constraint Satisfaction Problem

CSP = <X, D_X,C>

– variables X with domain D_X.

– Constraint C(X): $D_X \rightarrow$ {True, False}.

Problem: Find X in D_X s.t. C(X) is True.



Optimal CSP

OCSP= <Y, *g*, CSP>

- Decision variables Y with domain D_Y.
- Utility function g(Y): $D_Y \rightarrow \Re$.
- CSP over variables <X;Y>.

Find leading arg max *g*(Y)
$Y \in D_y$

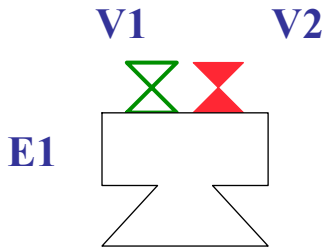
s.t. $\exists X \in D_X$ s.t. C(X,Y) is True.

➔ g: multi-attribute utility with preferential independence, value constraint, …

➔ CSP: propositional state logic, simple temporal problem, mixed logic-linear program, …

CSPs Are Frequently Encoded in Propositional State Logic

(mode(E1) = ok implies
(thrust(E1) = on if and only if flow(V1) = on and flow(V2) = on)) and
(mode(E1) = ok or mode(E1) = unknown) and
not (mode(E1) = ok and mode(E1) = unknown)



Multi Attribute Utility Functions

$g(Y) = G(g_1(y_1), g_2(y_2), \ldots)$

where

$G(u_1, u_2 \ldots u_n) = G(u_1, G(u_2 \ldots u_n))$

$G(u_1) = G(u_1, I_G)$

Example: Diagnosis

$g_i(y_i = mode_{ij}) = P(y_i = mode_{ij})$

$G(u_1, u_2) = u_1 \times u_2$

$I_G = 1$

Mutual Preferential Independence (MPI)

Assignment δ_1 is preferred over $\delta 2$ if $g(\delta_1) < g(\delta_2)$.

For any set of decision variables $W \subseteq Y$, our preference between two assignments to W is independent of the assignment to the remaining variables $W - Y$.

MPI Example: Diagnosis

If A1 = G is more likely than A1 = U,

then

{A1 = G, A2 = G, A3 = U, X1 = G, X2 = G}

is preferred to

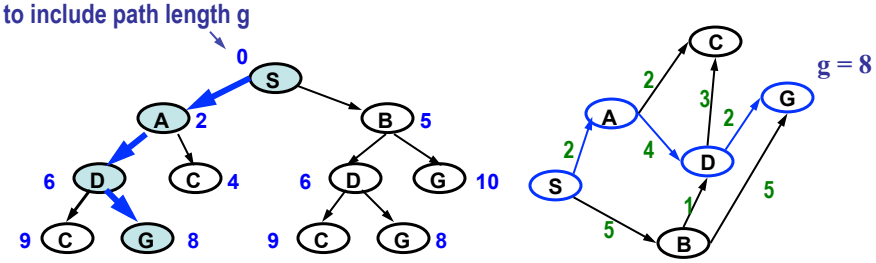
{A1 = U, A2 = G, A3 = U, X1 = G, X2 = G}.

Outline

- Model-based Diagnosis
- Optimal CSPs
- Informed Search
  - A*
  - Branch and Bound
- Conflict-directed A*

Informed Search

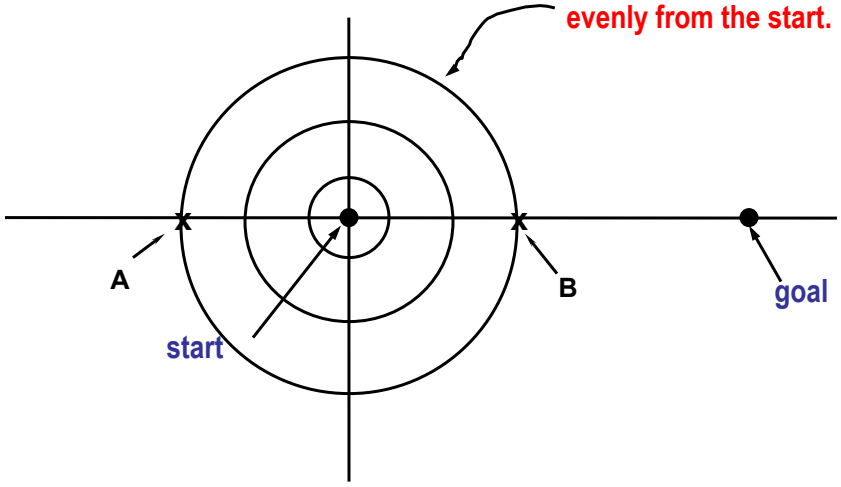
Extend search tree nodes
to include path length g

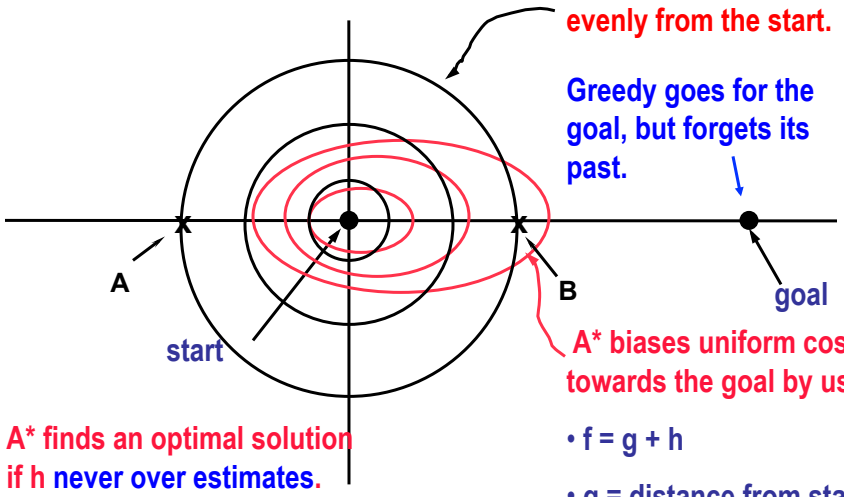


Problem:

- Given graph <V,E> with weight function w: $E \rightarrow \Re$, and vertices S, G in V,
- Find a path $S \rightarrow^p G$ with the shortest path length $\delta(S,G)$ where
  - $w(p) = \Sigma\, w(v_{i-1},v_i)$.
  - $\delta(u,v) = \min \{w(p) : u \rightarrow^p v\}$

Best-first Search with
Uniform Cost spreads
evenly from the start.





Best-first Search with Uniform Cost spreads evenly from the start.

Greedy goes for the goal, but forgets its past.

A* biases uniform cost towards the goal by using h.

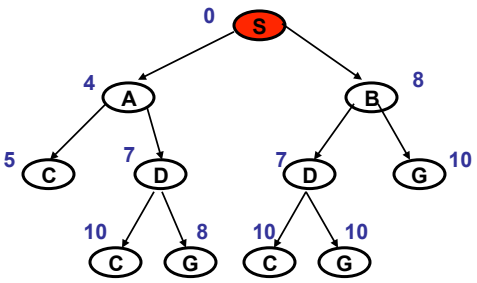
A* finds an optimal solution if h never over estimates.

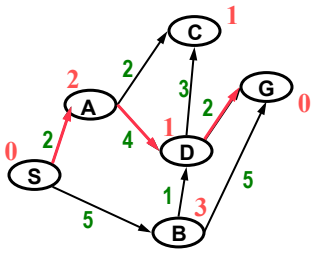
Then h is called "admissible"

- f = g + h
- g = distance from start.
- h = estimated distance to goal.

A*

- **Best-first Search with Q ordered by admissible f = g + h.**





Heuristic Value h in Red

Edge cost in Green

A* Search: State of Search

Problem:	State Space Search Problem.
• Θ	Initial State.
• Expand(*node*)	Children of Search Node = adjacent states.
• Goal-Test(node)	True if search node at a goal-state.
• Nodes	Search Nodes to be expanded.
• Expanded	Search Nodes already expanded.
• Initialize	Search starts at Θ, with no expanded nodes.
g(state)	Cost to state
h(state)	*Admissible Heuristic - Optimistic cost to go.*
Search Node:	Node in the search tree.
• State	State the search is at.
• Parent	Parent in search tree.
Nodes[Problem]:	
• Enqueue(*node, f*)	Adds node to those to be expanded.
• Remove-Best(f)	Removes best cost queued node according to f.

A* Search

Function A*(*problem, h*)
returns the best solution or failure. Problem pre-initialized.
***f*(x) ← *g*[*problem*](x) + *h*(x)**
loop do

Expand Best-first

***node* ← Remove-Best(Nodes[*problem*], f)**

new-nodes ← Expand(*node, problem*)
for each *new-node* in *new-nodes*

then Nodes[*problem*]← Enqueue(Nodes[*problem*], *new-node, f*)

end

A* Search

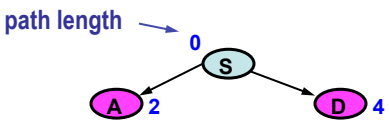
Function A*(*problem, h*)
returns the best solution or failure. Problem pre-initialized.
f(x) ← *g*[*problem*](x) + *h*(x)
loop do
if Nodes[*problem*] is empty then return failure
node ← Remove-Best(Nodes[*problem*], f)

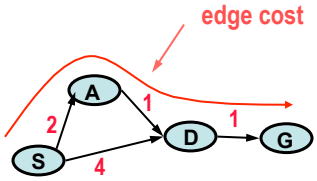
Terminate when . . .

new-nodes ← Expand(*node, problem*)
for each *new-node* in *new-nodes*

then Nodes[*problem*] ← Enqueue(Nodes[*problem*], *new-node, f*)
if Goal-Test[*problem*] applied to State(*node*) succeeds
then return *node*
end

Expand Vertices More Than Once

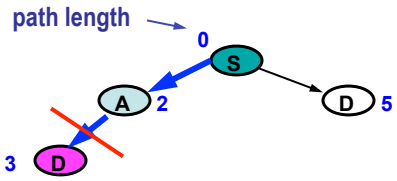


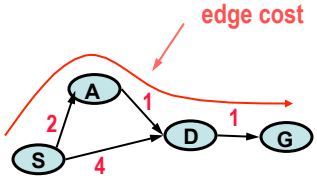


- The shortest path from S to G is (G D A S).
- D is reached first using path (D S).

Suppose we expanded only the first path that visits each vertex X?

Expand Vertices More Than Once

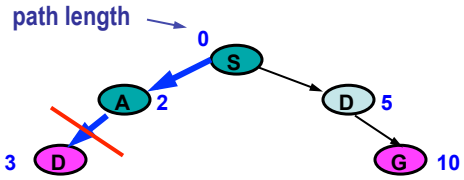


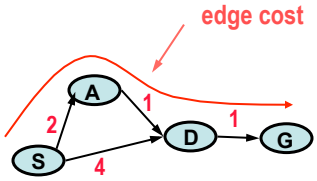


- The shortest path from S to G is (G D A S).
- D is reached first using path (D S).
- This prevents path (D A S) from being expanded.

Suppose we expanded only the first path that visits each vertex X?

Expand Vertices More Than Once

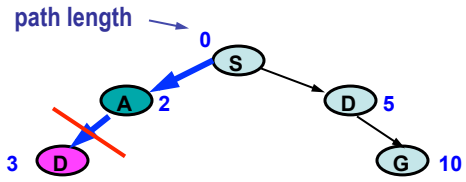


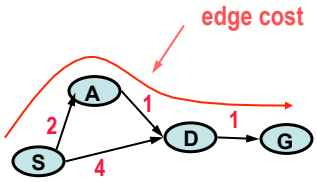


- The shortest path from S to G is (G D A S).
- D is reached first using path (D S).
- This prevents path (D A S) from being expanded.

Suppose we expanded only the first path that visits each vertex X?

Expand Vertices More Than Once





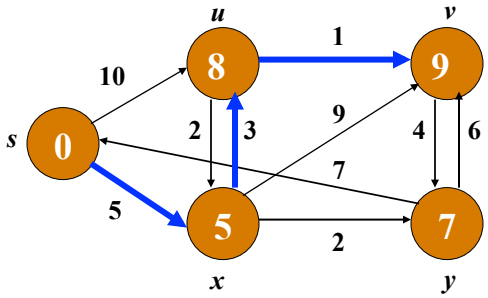
- The shortest path from S to G is (G D A S).
- D is reached first using path (D S).
- This prevents path (D A S) from being expanded.
- The suboptimal path (G D S) is returned.

Suppose we expanded only the first path that visits each vertex X?

⇨**Eliminate the Visited List.**

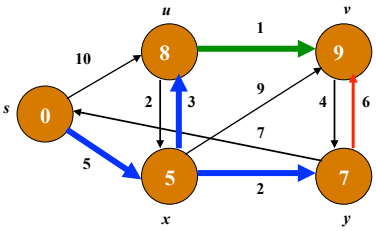
⇨**Return solution when taken off queue.**

Shortest Paths Contain Only Shortest Paths



- Subpaths of shortest paths are shortest paths.
  - $s \rightarrow^p v$ = <s, x, u, v> Shortest, then ….
  - $s \rightarrow^p u$ = <s, x, u> Shortest
  - $s \rightarrow^p x$ = <s, x> Shortest

Shortest Paths can be Grown From Shortest Paths



The length of shortest path $s \rightarrow^p u \rightarrow v$ is $\delta(s,v) = \delta(s,u) + w(u,v)$ such that $\forall <u,v> \in E \quad \delta(s,v) \leq \delta(s,u) + w(u,v)$.

Dynamic Programming Principle:

- Given the shortest path to U, don't extend other paths to U; delete them (expanded list).
- When A* dequeues the first partial path with head node U, this path is *guaranteed to be the shortest path* from S to U.

A* Search

Function A*(*problem, h*)
returns the best solution or failure. Problem pre-initialized.
f(x) ← *g*[*problem*](x) + *h*(x)
loop do
if Nodes[*problem*] is empty **then return** failure
node ← Remove-Best(Nodes[*problem*], f)
***state* ← State(*node*)**
remove any n from Nodes[*problem*] such that State(n) = *state*
Expanded[*problem*] ← Expanded[*problem*] ∪ {*state*}
new-nodes ← Expand(*node, problem*)
for each *new-node* in *new-nodes*
unless State(*new-node*) is in Expanded[*problem*]
then Nodes[*problem*] ← Enqueue(Nodes[*problem*], *new-node, f*)
if Goal-Test[*problem*] applied to State(*node*) succeeds
then return *node*
end

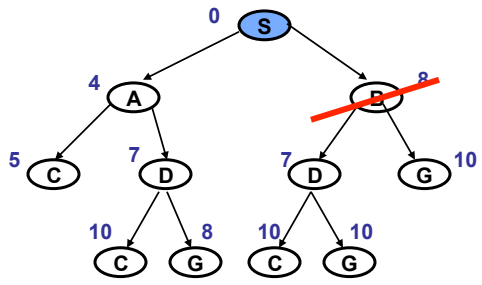
Dynamic Programming Principle . . .

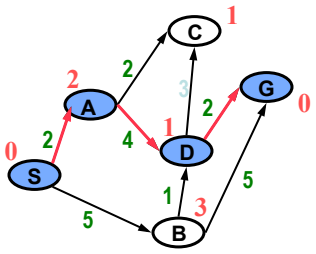
Outline

- Model-based Diagnosis
- Optimal CSPs
- Informed Search
  - A*
  - Branch and Bound
- Conflict-directed A*

Branch and Bound

- **Maintain the best solution found thus far (incumbent).**
- **Prune all subtrees worse than the incumbent.**



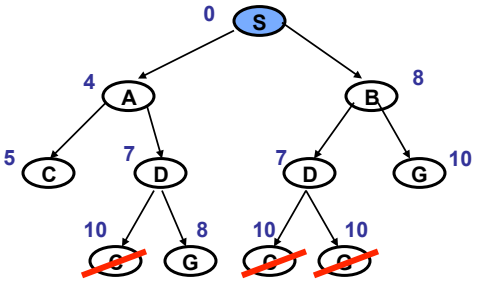


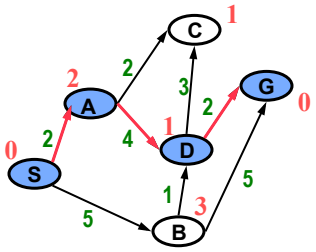
Incumbent:
cost U = ∞, 8
path P = (), (S A D G)

Heuristic Value h in Red
Edge cost in Green

Branch and Bound

- Maintain the best solution found thus far (incumbent).
- Prune all subtrees worse than the incumbent.
- Any search order allowed (DFS, Reverse-DFS, BFS, Hill w BT…).





Incumbent:

cost U = ∞, 10, 8

path P = (), (S B G) (S A D G)

Heuristic Values of g in Red

Edge cost in Green

Simple Optimal Search Using Branch and Bound

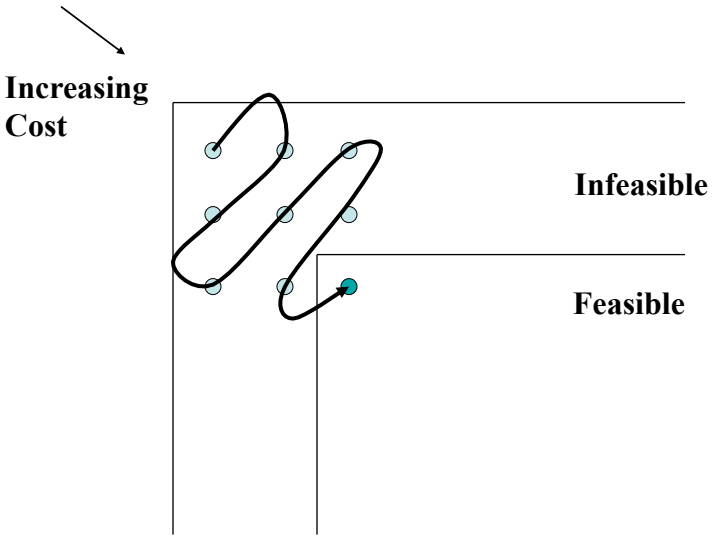
Let <V,E> be a Graph Let Q be a list of simple partial paths in <V,E>
Let S be the start vertex in <V,E> and Let G be a Goal vertex in <V,E>.
Let f = g + h be an admissible heuristic function.
U and P are the cost and path of the best solution thus far (Incumbent).

1. Initialize Q with partial path (S); Incumbent U = ∞, P = ();
2. If Q is empty, return Incumbent U and P,
   Else, remove a partial path N from Q;
3. If f(N) >= U, Go to Step 2.
4. If head(N) = G, then U = f(N) and P = N (a better path to the goal)
5. (Else) Find all children of head(N) (its neighbors in <V,E>) and create one-step extensions from N to each child.
6. Add extended paths to Q.
7. Go to Step 2.

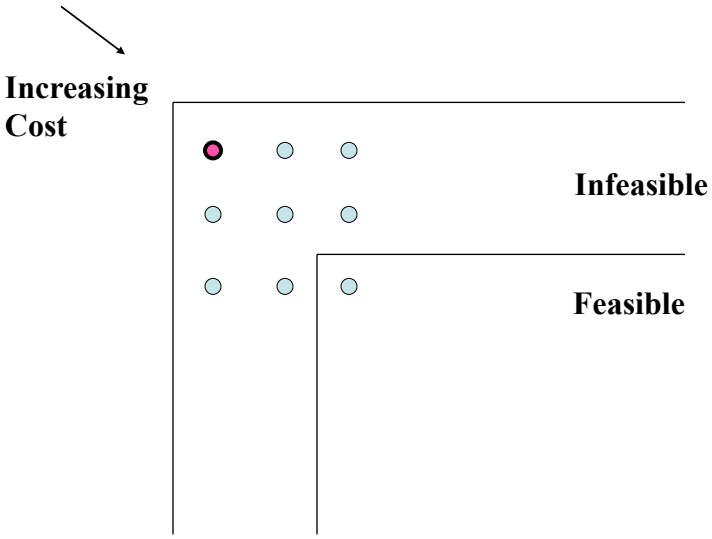
Outline

- Model-based Diagnosis
- Optimal CSPs
- Informed Search
- Conflict-directed A*

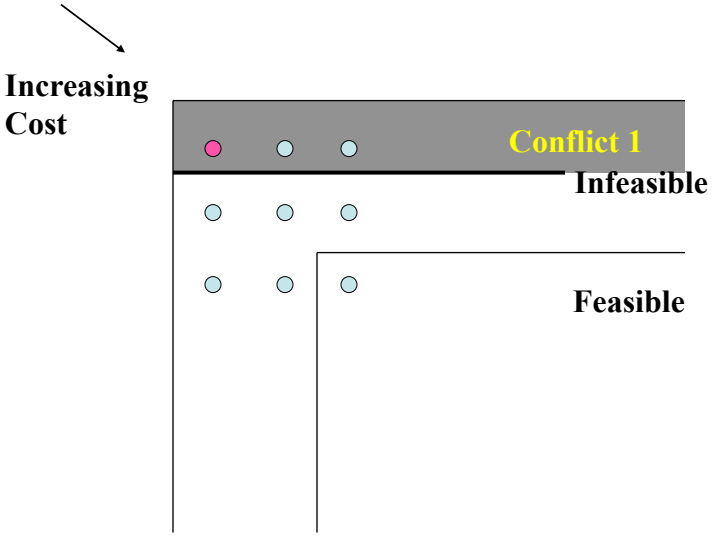
A*



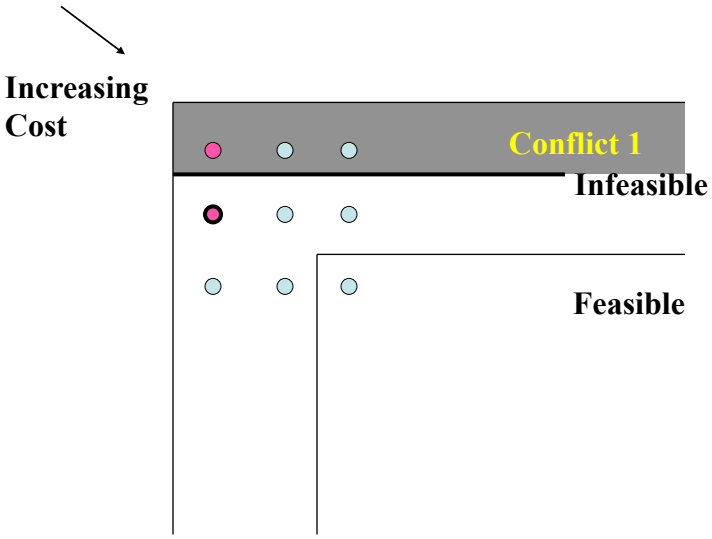
Conflict-directed A*



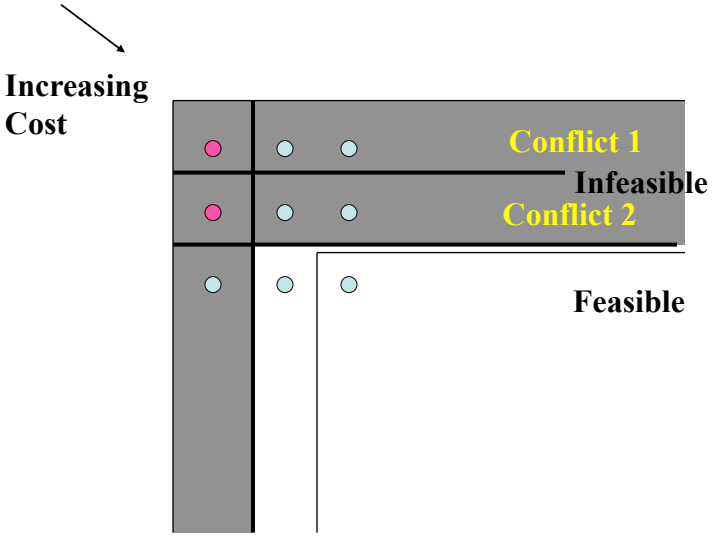
Conflict-directed A*



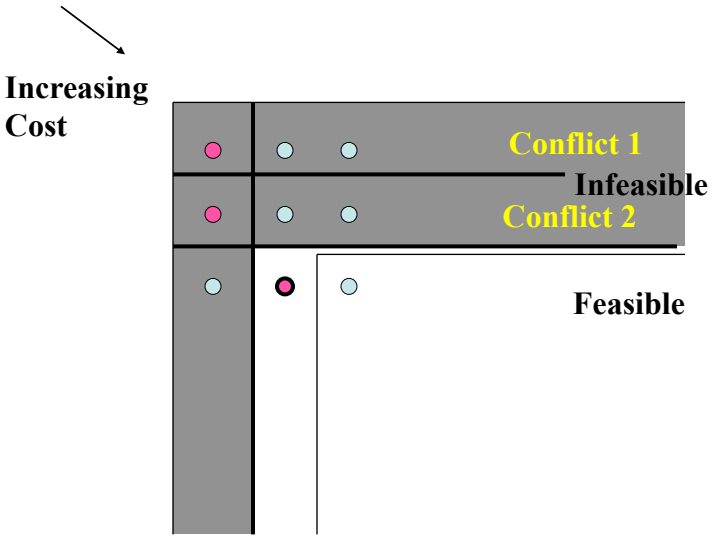
Conflict-directed A*



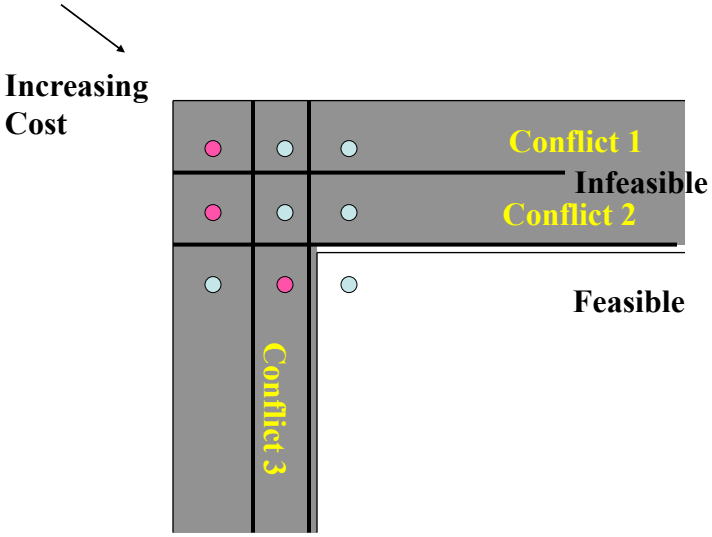
Conflict-directed A*



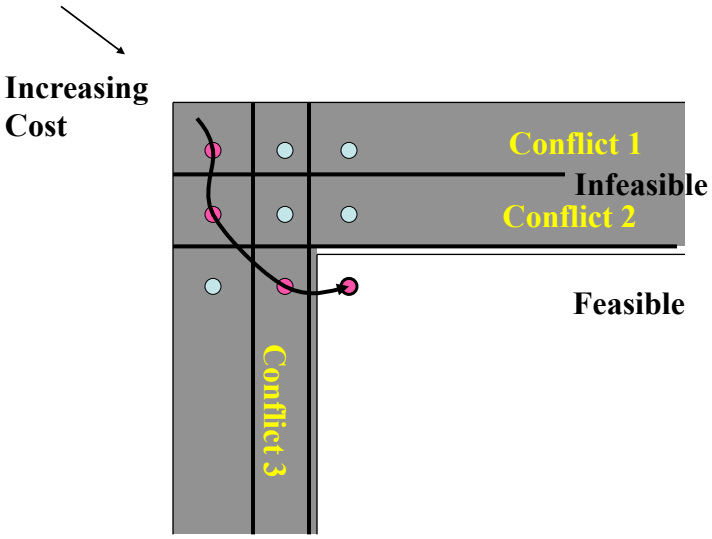
Conflict-directed A*



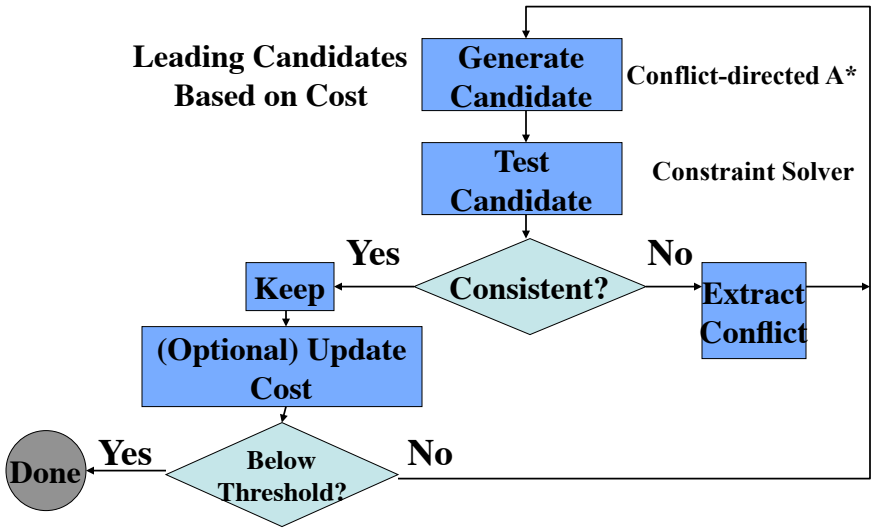
Conflict-directed A*



Conflict-directed A*



Solving Optimal CSPs Through Generate and Test



Conflict-directed A*

Function Conflict-directed-A*(*OCSP*)
returns the leading minimal cost solutions.
Conflicts[*OCSP*] ← {}
OCSP ← Initialize-Best-Kernels(*OCSP*)
Solutions[*OCSP*] ← {}
loop do
⇨ ***decision-state*** **← Next-Best-State-Resolving-Conflicts(OCSP)**

Conflict-guided Expansion

⇨ ***new-conflicts*** **← Extract-Conflicts(CSP[OCSP],** ***decision-state*****)**
⇨ **Conflicts[OCSP]**
← Eliminate-Redundant-Conflicts(Conflicts[OCSP] ∪ new-conflicts)
end

Conflict-directed A*

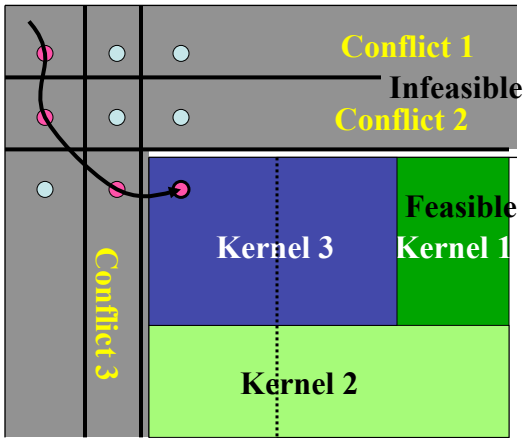
Function Conflict-directed-A*(*OCSP*)
returns the leading minimal cost solutions.
Conflicts[*OCSP*] ← {}
OCSP ← Initialize-Best-Kernels(*OCSP*)
Solutions[*OCSP*] ← {}
loop do
decision-state ← Next-Best-State-Resolving-Conflicts(OCSP)
⇨ **if no decision-state returned or**
Terminate?(*OCSP*)
then return Solutions[*OCSP*]
⇨ **if Consistent?(CSP[*OCSP*],** ***decision-state)***
then add *decision-state* to Solutions[OCSP]
new-conflicts ← Extract-Conflicts(CSP[OCSP], *decision-state*)
Conflicts[OCSP]
← Eliminate-Redundant-Conflicts(Conflicts[OCSP] ∪ new-conflicts)
end

Conflict-directed A*

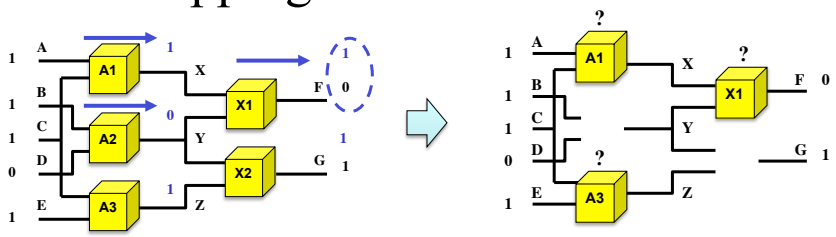
- Each feasible subregion described by a kernel assignment.

⇨ Approach: Use conflicts to search for kernel assignment containing the best cost candidate.

Increasing Cost



Recall: Mapping Conflicts to Kernels



Conflict C_i: A set of decision variable assignments that are inconsistent with constraints Φ.

C_i ^ Φ is inconsistent ➔ Φ entails ¬C_i

Constituent Kernel: An assignment a that resolves a conflict C_i.

a entails ¬ C_i

Kernel: A minimal set of decision variable assignments that resolves all known conflicts C.

A entails ¬ C_i for all C_i in C

Extracting a kernel's best state

- **Select best utility value for unassigned variables (Why?).**

{X1=U}

A1=? ∧ A2=U ∧ A3=? ∧ X1=? ∧ X2=?

A1=G ∧ A2=U ∧ A3=G ∧ X1=G ∧ X2=G

Next Best State Resolving Conflicts

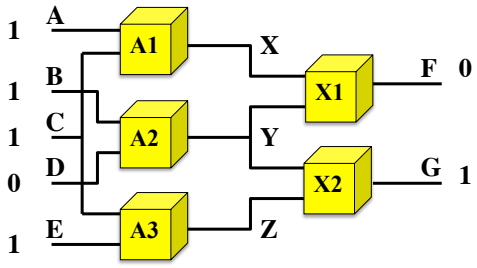
function Next-Best-State-Resolving-Conflicts(OCSP)
  best-kernel ← **Next-Best-Kernel(*OCSP*)**
  if *best-kernel = failure*
    then return failure
    else return kernel-Best-State[problem](*best-kernel)*
end

function Kernel-Best-State(kernel)
  unassigned ← **all variables not assigned in *kernel***
  return *kernel* ∪ {Best-Assignment(*v*) | v ∈ *unassigned*}
End

function Terminate?(*OCSP*)
  return True iff Solutions[*OCSP*] is non-empty

Algorithm for only finding the first solution, multiple later.

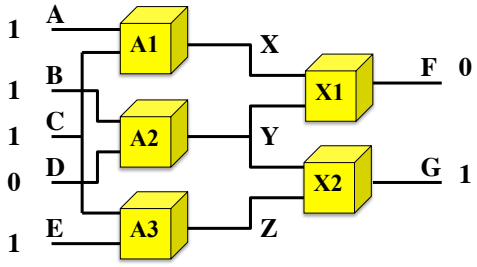
Example: Diagnosis



Assume Independent Failures:

- $P_{G(mi)} >> P_{U(mi)}$
- $P_{single} >> P_{double}$
- $P_{U(A2)} > P_{U(A1)} > P_{U(A3)} > P_{U(X1)} > P_{U(X2)}$

First Iteration



- Conflicts / Constituent Kernels
  - none
- Best Kernel:
  - {}
- Best Candidate:
  - ?

Extracting the kernel's best state

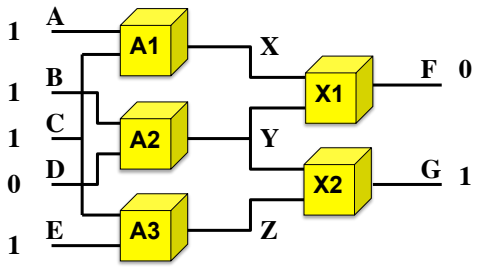
- **Select best value for unassigned variables**

{ }

A1=? ∧ A2=? ∧ A3=? ∧ X1=? ∧ X2=?

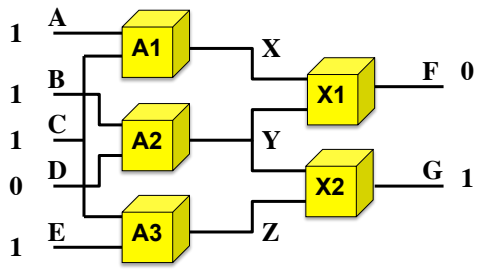
A1=G ∧ A2=G ∧ A3=G ∧ X1=G ∧ 2=G

First Iteration

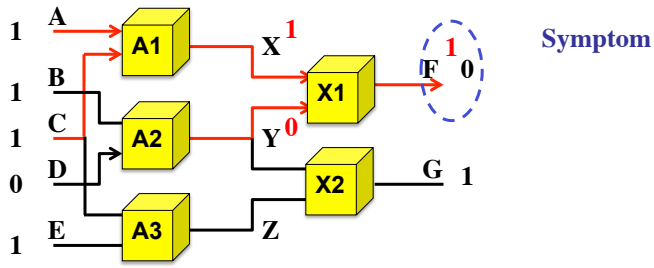


- Conflicts / Constituent Kernels
  - none
- Best Kernel:
  - {}
- Best Candidate:
  - A1=G ∧ A2=G ∧ A3=G ∧ X1=G ∧ X2=G
  - ?

Test: A1=G ∧ A2=G ∧ A3=G ∧ X1=G ∧ X2=G



Test: A1=G ∧ A2=G ∧ A3=G ∧ X1=G ∧ X2=G



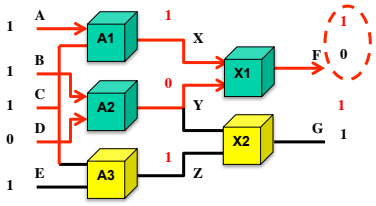
- Extract Conflict and Constituent Kernels:

  ¬ [A1=G ∧ A2=G ∧ X1=G]

  A1=U ∨ A2=U ∨ X1=U

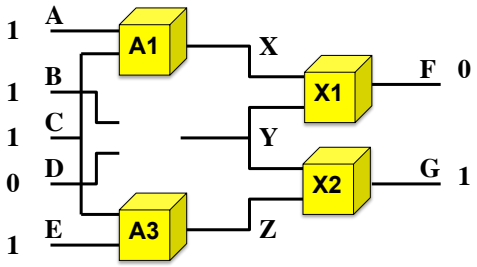
Second Iteration

- $P_{G(mi)} >> P_{U(mi)}$
- $P_{single} >> P_{double}$
- $P_{U(A2)} > P_{U(A1)} > P_{U(A3)} > P_{U(X1)} > P_{U(X2)}$

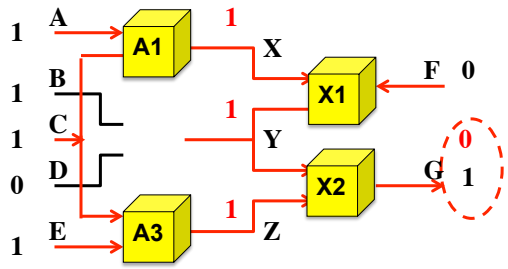


- Conflicts ⇨ Constituent Kernels
  - A1=U ∨ A2=U ∨ X1=U
- Best Kernel:
  - A2=U (why?)
- Best Candidate:
  - A1=G ∧ A2=U ∧ A3=G ∧ X1=G ∧ X2=G

Test: A1=G ∧ A2=U ∧ A3=G ∧ X1=G ∧ X2=G



Test: A1=G ∧ A2=U ∧ A3=G ∧ X1=G ∧ X2=G



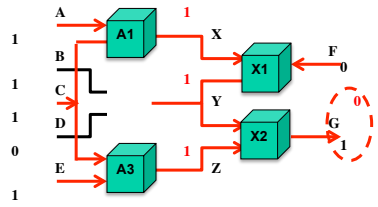
- **Extract Conflict:**

  ¬ [A1=G ∧ A3=G ∧ X1=G ∧ X2=G]

  A1=U ∨ A3=U ∨ X1=U ∨ X2=U

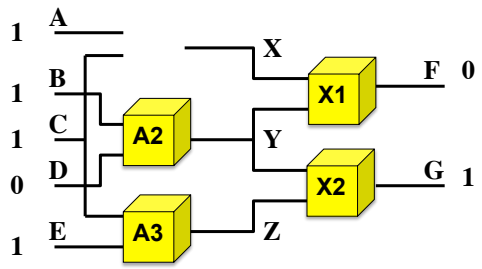
Third Iteration

- $P_{G(mi)} >> P_{U(mi)}$
- $P_{single} >> P_{double}$
- $P_{U(A2)} > P_{U(A1)} > P_{U(A3)} > P_{U(X1)} > P_{U(X2)}$

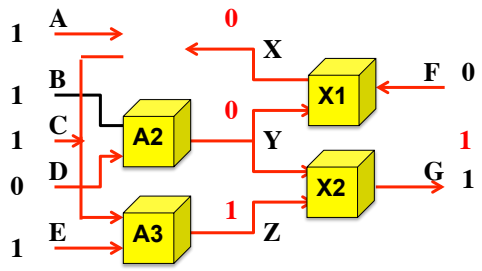


- Conflicts ⇨ Constituent Kernels
  - A1=U ∨ A2=U ∨ X1=U
  - A1=U ∨ A3=U ∨ X1=U ∨ X2=U
- Best Kernel:
  - A1=U
- Best Candidate:
  - A1=U ∧ A2=G ∧ A3=G ∧ X1=G ∧ X2=G

Test: A1=U ∧ A2=U ∧ A3=G ∧ X1=G ∧ X2=G



Test: A1=U ∧ A2=U ∧ A3=G ∧ X1=G ∧ X2=G

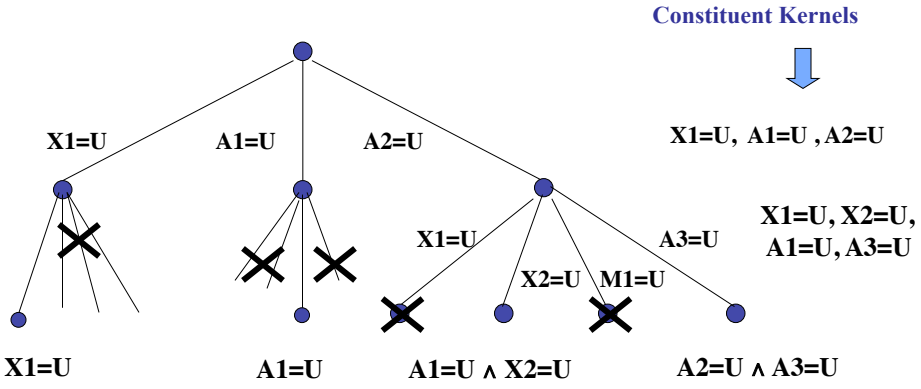


- **Consistent!**

Outline

- Model-based Diagnosis
- Optimal CSPs
- Conflict-directed A*
  - Generating the Best Kernel
  - Performance Comparison

Generating The Best Kernel of The Known Conflicts

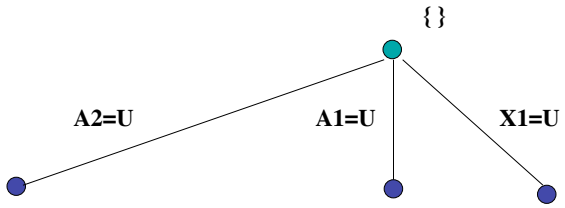


Insight:
- **Kernels found by minimal set covering**
- **Minimal set covering is an instance of breadth first search.**

Expanding a Node to Resolve a Conflict

Constituent kernels

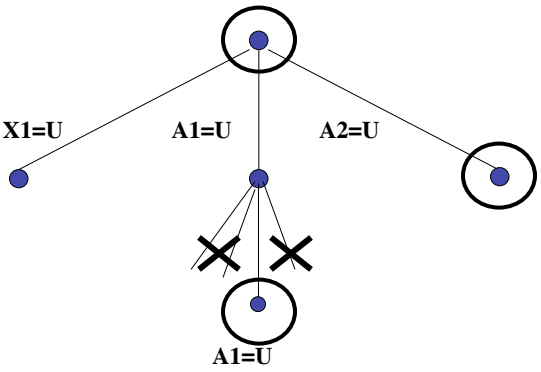
A2=U v A1=U v X1=U



To Expand a Node:

- **Select an unresolved Conflict.**
- **Each child adds a constituent kernel of Conflict.**
- **Prune any node that is**
  - **Inconsistent, or**
  - **A superset of a known kernel.**

Generating The Best Kernel of The Known Conflicts



Constituent Kernels

X1=U, A1=U, A2=U

X1=U, X2=U, A1=U, A3=U

Insight:

- **Kernels found by minimal set covering**
- **Minimal set covering is an instance of breadth first search.**

➔ **To find the best kernel, expand tree in best first order.**

Admissible h(α): Cost of best state that extends partial assignment α

f = g + h

A2=U ∧ A1=? ∧ A3=? ∧ X1=? ∧ X2=?

$P_{A2=u}$ **x** $P_{A1=G}$ **x** $P_{A3=G}$ **x** $P_{X1=G}$ **x** $P_{X2=G}$

- **Select best value of unassigned variables.**

Admissible Heuristic h

- Let g = <G,g_i,Y> describe a multi-attribute utility fn
- Assume the preference for one attribute x_i is independent of another x_k
  - *Called Mutual Preferential Independence*:

    For all u, v ∈Y

    If $g_i(u) \geq g_i(v)$ then for all w

    $G(g_i(u),g_k(w)) \geq G(g_i(v),g_k(w))$

An Admissible h:

- Given a partial assignment, to X ⊆ Y
- h selects the best value of each unassigned variable Z = X – Y

$$h(Y) = G(\{g_{zi\_max} | z_i \in Z, \max_{v_{ij} \in D_{zi}} g_{zi}(v_{ij})\})$$

- A candidate always exists satisfying h(Y).

Terminate when all conflicts resolved

```
Function Goal-Test-Kernel (node, problem)
  returns True IFF node is a complete decision state.
  if forall K in Constituent-Kernels(Conflicts[problem]),
      State[node] contains a kernel in K
    then return True
 else return False
```


Next Best Kernel of Known Conflicts

```
Function Next-Best-Kernel (OCSP)
  returns the next best cost kernel of Conflicts[OCSP].
  f(x) ← G[OCSP] (g[OCSP](x), h[OCSP](x))
  loop do
    if Nodes[OCSP] is empty  then return failure
    node ← Remove-Best(Nodes[OCSP], f)
    add State[node] to Visited[OCSP]
    new-nodes  ← Expand-Conflict(node, OCSP)
    for each new-node ∈ new-nodes
      unless ∃ n ∈ Nodes[OCSP] such that State[new-node] = State[n]
             OR State[new-node] ∈ Visited[problem]
        then Nodes[OCSP] ← Enqueue(Nodes[OCSP], new-node, f )
    if Goal-Test-Kernel[OCSP] applied to State[node] succeeds
      Best-Kernels[OCSP]
        ← Add-To-Minimal-Sets(Best-Kernels[OCSP], best-kernel)
      if best-kernel ∈ Best-Kernels[OCSP]
        then return State[node]
  end
```

An instance of A*

Outline

- Model-based Diagnosis
- Optimal CSPs
- Conflict-directed A*
  - Generating the Best Kernel
  - Performance Comparison

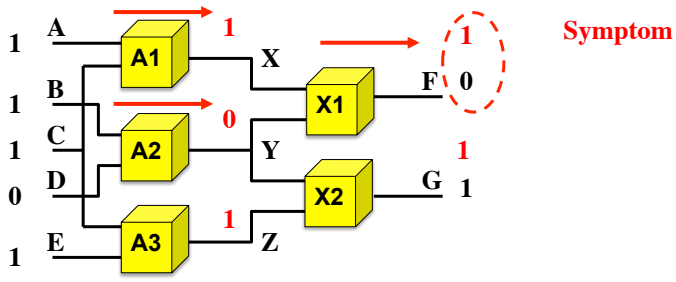
Performance: With and Without Conflicts

Problem Parameters				Constraint-based A* (no conflicts)		Conflict-directed A*			Mean CD-CB Ratio	
Dom Size	Dec Vars	Clau-ses	Clau-se lngth	Nodes Expanded	Queue Size	Nodes Expand	Queue Size	Conflicts used	Nodes Expanded	Queue Size
5	10	10	5	683	1,230	3.3	6.3	1.2	4.5%	5.6%
5	10	30	5	2,360	3,490	8.1	17.9	3.2	2.4%	3.5%
5	10	50	5	4,270	6,260	12.0	41.3	2.6	0.83%	1.1%
10	10	10	6	3,790	13,400	5.7	16.0	1.6	2.0%	1.0%
10	10	30	6	1,430	5,130	9.7	94.4	4.2	4.6%	5.8%
10	10	50	6	929	4,060	6.0	27.3	2.3	3.5%	3.9%
5	20	10	5	109	149	4.2	7.2	1.6	13.0%	13.0%
5	20	30	5	333	434	6.4	9.2	2.2	6.0%	5.4%
5	20	50	5	149	197	5.4	7.2	2.0	12.0%	11.0%

Multiple Fault Diagnosis of Systems with Novel Failures

Consistency-based Diagnosis: Given symptoms, find diagnoses that are consistent with symptoms.

Suspending Constraints: Make no presumption about a component's faulty behavior.



Model-based Diagnosis as Conflict-directed Best First Search

When you have eliminated the impossible, whatever remains, however improbable, must be the truth.

- Sherlock Holmes. The Sign of the Four.

1. Generate most likely Hypothesis.
2. Test Hypothesis.
3. If Inconsistent, learn reason for inconsistency (a Conflict).
4. Use conflicts to leap over similarly infeasible options to next best hypothesis.

Outline

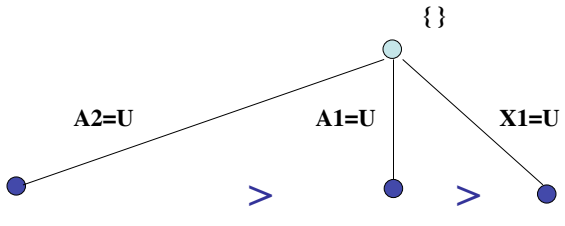
- Model-based Diagnosis
- Optimal CSPs
- Conflict-directed A*
  - Generating the Best Kernel
  - Performance Comparison
  - Appendix:
    - Intelligent Tree Expansion
    - Extending to Multiple Solutions
    - Review of A*

Expand Only Best Child & Sibling

Constituent kernels

A2=U ∨ A1=U ∨ X1=U

Order constituents by decreasing utility.



- **Traditionally all children expanded.**
- **Only need child containing best candidate.**

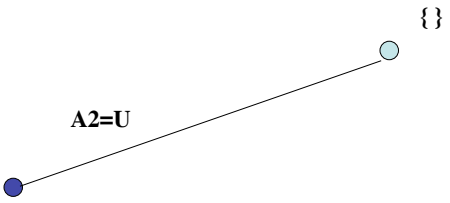
⇨ **Child with best estimated cost f = g+h.**

Expand Only Best Child & Sibling

Constituent kernels

A2=U v A1=U v X1=U

Order constituents by decreasing utility



- Traditionally all children expanded.
- Only need child with best candidate.

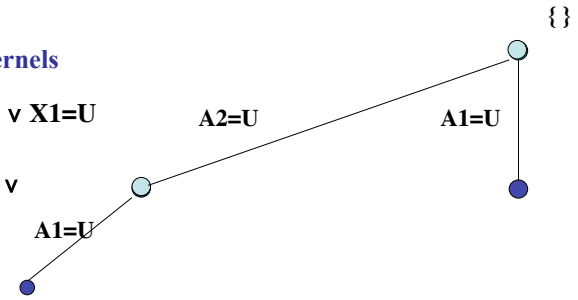
⇨ Child with best estimated cost f = g+h.

When Do We Expand The Child's Next Best Sibling?

Constituent kernels

A2=U v A1=U v X1=U

A1=U v A3=U v X1=U v X2=U



- When a best child has a subtree or leaf pruned, it may have lost its best candidate.
- One of the child's siblings might now contain the best candidate.

⇨ Expand child's next best sibling:

– when expanding children to resolve another conflict.

Expand Node to Resolve Conflict

function Expand-Conflict(*node, OCSP*)
 return Expand-Conflict-Best-Child*(node, OCSP)* ∪
 Expand-Next-Best-Sibling (node, OCSP)

function Expand-Conflict-Best-Child(*node, OCSP)*
 if for all K_γ in Constituent-Kernels(Γ[OCSP])
 State[*node*] contains a kernel ∈ K_γ
 then return {}
 else return Expand-Constituent-Kernel*(node,OCSP)*

function Expand-Constituent-Kernel(*node, OCSP)*
 K_γ ← = smallest uncovered set ∈ Constituent-Kernels(Γ[*OCSP*])
 C ← {$y_i = v_{ij}$ | {$y_i = v_{ij}$} in K_γ, $y_i = v_{ij}$ is consistent with State[*node*]}
 Sort *C* such that for all *i* from 1 to |*C*| - 1,
 Better-Kernel?(C[*i*],C[*i+1*], *OCSP*) is True
 Child-Assignments[*node]* ← *C*
 $y_i = v_{ij}$ ← *C*[1], *which is the best kernel in K_γ consistent with State[*node*]*
 return {Make-Node({$y_i = v_{ij}$}, node)}

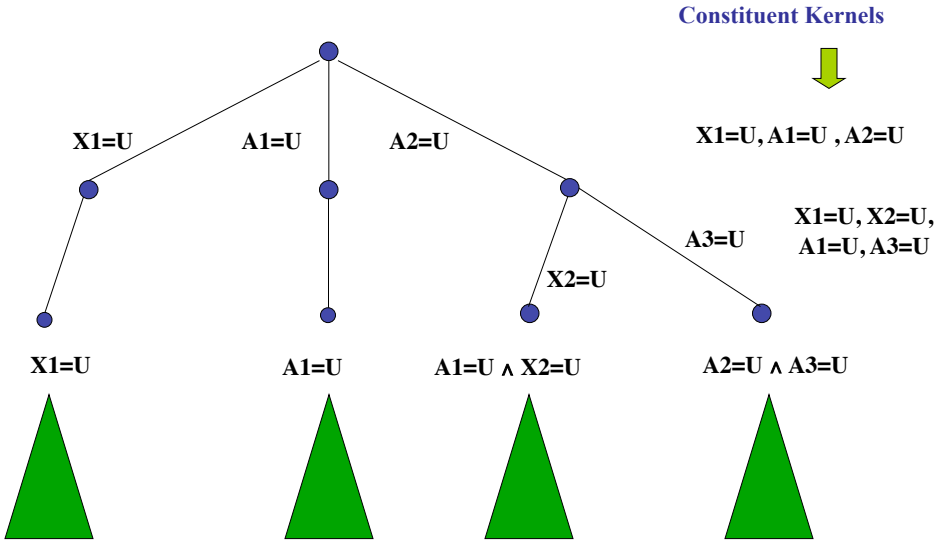
Expand Node to Resolve Conflict

function Expand-Next-Best-Sibling(*node, OCSP)*
 if Root*?*[*node*]
 then return {}
 else {$y_i = v_{ij}$} ← Assignment[*node*]
 {$y_k = v_{kl}$} ← next best assignment in consistent
 child-assignments[Parent[*node*]] after {$y_i = v_{ij}$}
 if no next assignment {$y_k = v_{kl}$}
 or **Parent**[*node*] already has a child with {$y_k = v_{kl}$}
 then return {}
 else return {Make-Node({$y_k = v_{kl}$}, Parent[*node*])}

Outline

- Model-based Diagnosis
- Optimal CSPs
- Conflict-directed A*
  - Generating the Best Kernel
  - Performance Comparison
  - Appendix:
    - Intelligent Tree Expansion
    - Extending to Multiple Solutions
    - Review of A*

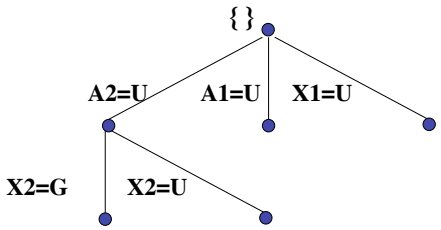
Multiple Solutions: Systematically Exploring Kernels



Child Expansion For Finding Multiple Solutions

Conflict

¬ (A2=G ∧ A1=G ∧ X1=G)



If Unresolved Conflicts:

- **Select unresolved conflict.**
- **Each child adds a constituent kernel.**

If All Conflicts Resolved:

- **Select unassigned variable y_i.**
- **Each child adds an assignment from D_i.**

Intelligent Expansion Below a Kernel

Select Unassigned Variable.

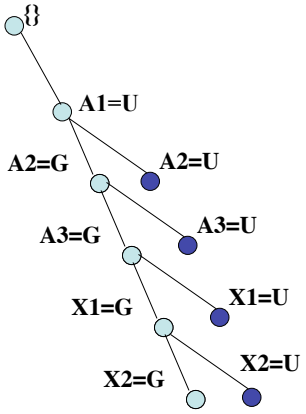
A2=G ∨ A2=U

Order assignments by decreasing utility.

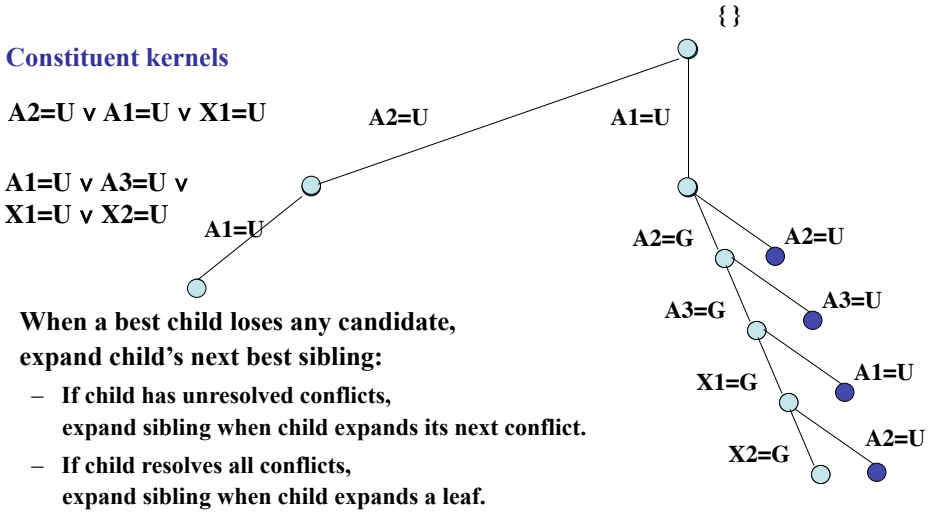
Expand best child.

Continue expanding best descendents.

When leaf visited,, expand all next best ancestors. (why?)



Putting It Together: Expansion Of Any Search Node



- When a best child loses any candidate, expand child's next best sibling:
  - If child has unresolved conflicts, expand sibling when child expands its next conflict.
  - If child resolves all conflicts, expand sibling when child expands a leaf.

Conflict-directed A*

When you have eliminated the impossible, whatever remains, however improbable, must be the truth.

- Sherlock Holmes. The Sign of the Four.

1. Generate most likely hypothesis.
2. Test hypothesis.
3. If inconsistent, learn reason for inconsistency (a Conflict).
4. Use conflicts to leap over similarly infeasible options to next best hypothesis.

Outline

- Using conflicts in backtrack search
  - Dependency-directed backtracking
  - Conflict learning
  - Conflict-directed backjumping

Using Conflicts to Guide Search: Dependency-directed Search [Stallman & Sussman, 1978]

Input: Constraint satisfaction problem.
Output: Satisfying assignment.

Repeat while a next candidate assignment exists.
- Generate candidate assignment c.
- Check candidate c against conflicts.
  - If c is a superset of a conflict,
    Then loop to the next candidate.
- Check consistency of c.
  - If inconsistent,
  - Then extract and record a conflict from c.
  - Else return c as a solution.

⇒ Like a Graphplan memo, but generalizes an inconsistent solution.

Procedure Dependency_directed_Backtracking (<X,D,C>)

Input: A constraint network R = <X, D, C>
Output: A solution, or notification that the network is inconsistent.

```
i ← 1; a_i = {}; conflicts = {}        Initialize variable counter, assignments,
D'_i ← D_i;                            Copy domain of first variable.
while 1 ≤ i ≤ n
  instantiate x_i ← Select-DDB-Value(); Add to assignments a_i
  if x_i is null                       No value was returned,
    i ← i - 1;                         then backtrack
  else
    i ← i + 1;                         else step forward and
    D'_i ← D_i;                              copy domain of next variable
 end while
if i = 0
  return "inconsistent"
else
  return a_i, the instantiated values of {x_i, …, x_n}
end procedure
```


Procedure Select-DDB-Value()

Output: A value in D'_i consistent with $\vec{a}_{i-1}$, or null, if none.

```
while D'_i is not empty
  select an arbitrary element a ∈ D'_i and remove a from D'_i;
  a_i ← a_{i-1},∪ {x_i = a};
  if for every c in conflicts, not (a_i superset c)
    if consistent(a_{i-1}, x_i = a )
      return a;
    else conflicts ← conflicts ∪
                     minimal inconsistent subset of a_{i-1};
end while
return null
end procedure
```

16.410 / 16.413 Principles of Autonomy and Decision Making

Fall 2010