FastForward: Heuristics For Planning

Chad Hogg

2007-02-27

Outline

Outline

State-Space Representation

- Back to classical planning for a moment ...

State-Space Representation

- Back to classical planning for a moment ...
- A *state* S is a set of ground literals.

State-Space Representation

- Back to classical planning for a moment ...
- A *state* S is a set of ground literals.
- An *action* $A = (h, p, e^+, e^-)$ consists of a head and 3 sets of literals – preconditions, positive effects, and negative effects.

State-Space Representation

- Back to classical planning for a moment ...
- A *state* S is a set of ground literals.
- An *action* $A = (h, p, e^+, e^-)$ consists of a head and 3 sets of literals – preconditions, positive effects, and negative effects.
- The application of an action A to a state S is another state $\gamma(S, A) = (S - e^-) \cup e^+$.

Forward Search in State Space

- Forward search applies actions to an initial state until a goal state is reached.

Forward Search in State Space

- Forward search applies actions to an initial state until a goal state is reached.
- Depth-first search expands until dead-end, then backtracks

Forward Search in State Space

- Forward search applies actions to an initial state until a goal state is reached.
- Depth-first search expands until dead-end, then backtracks
  - Sound, but not complete or admissible in general case

Forward Search in State Space

- Forward search applies actions to an initial state until a goal state is reached.
- Depth-first search expands until dead-end, then backtracks
  - Sound, but not complete or admissible in general case
- Breadth-first search checks all 1-action sequences, then all 2-action sequences, etc.

Forward Search in State Space

- Forward search applies actions to an initial state until a goal state is reached.
- Depth-first search expands until dead-end, then backtracks
  - Sound, but not complete or admissible in general case
- Breadth-first search checks all 1-action sequences, then all 2-action sequences, etc.
  - Sound, complete, and admissible

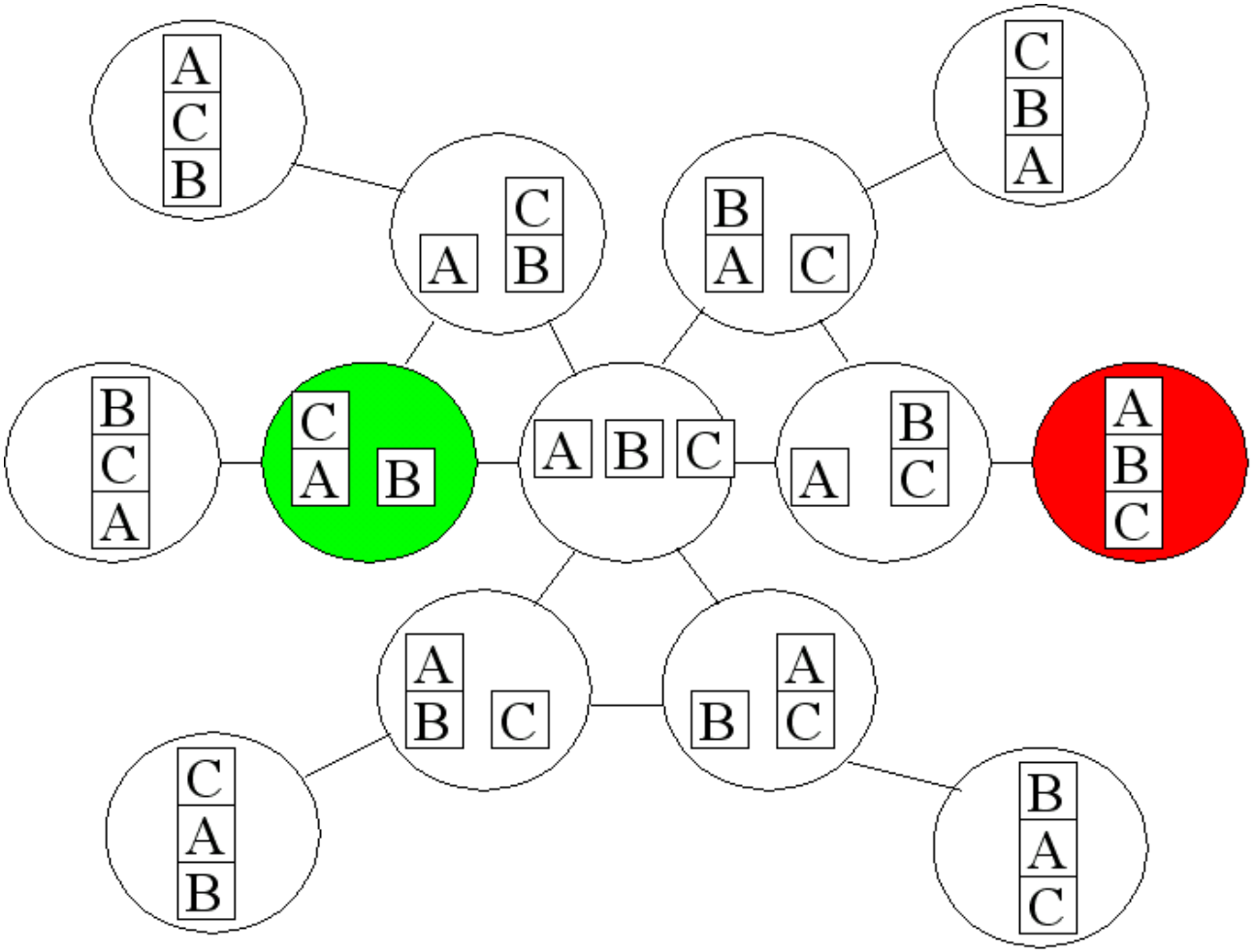
Forward Search in State Space

- Forward search applies actions to an initial state until a goal state is reached.
- Depth-first search expands until dead-end, then backtracks
  - Sound, but not complete or admissible in general case
- Breadth-first search checks all 1-action sequences, then all 2-action sequences, etc.
  - Sound, complete, and admissible
  - Very memory intensive

Forward Search in State Space

- Forward search applies actions to an initial state until a goal state is reached.
- Depth-first search expands until dead-end, then backtracks
  - Sound, but not complete or admissible in general case
- Breadth-first search checks all 1-action sequences, then all 2-action sequences, etc.
  - Sound, complete, and admissible
  - Very memory intensive
- Best-first, Iterative Deepening are specialized

Our Familiar Example



Outline

VanillaIce Properties

- Written by me last week, largely untested

VanillaIce Properties

- Written by me last week, largely untested
- A breadth-first, forward-searching state-space classical planner

VanillaIce Properties

- Written by me last week, largely untested
- A breadth-first, forward-searching state-space classical planner
- Sound, complete, and admissible (assuming no bugs)

VanillaIce Properties

- Written by me last week, largely untested
- A breadth-first, forward-searching state-space classical planner
- Sound, complete, and admissible (assuming no bugs)
- Much, much slower than state-of-the-art systems

VanillaIce Usage

- Binary is located at /home/cmh204/vanilla_ice/vanilla_ice1.0/vanilla_ice on vega.cc.lehigh.edu

VanillaIce Usage

- Binary is located at /home/cmh204/vanilla_ice/vanilla_ice1.0/vanilla_ice on vega.cc.lehigh.edu
- Source is available upon request

VanillaIce Usage

- Binary is located at /home/cmh204/vanilla_ice/vanilla_ice1.0/vanilla_ice on vega.cc.lehigh.edu
- Source is available upon request
- All relevant information and instructions are in README.txt file

VanillaIce Usage

- Binary is located at /home/cmh204/vanilla_ice/vanilla_ice1.0/vanilla_ice on vega.cc.lehigh.edu
- Source is available upon request
- All relevant information and instructions are in README.txt file
- Sample domain and problem input files in PDDL are provided

VanillaIce Usage

- Binary is located at /home/cmh204/vanilla_ice/vanilla_ice1.0/vanilla_ice on vega.cc.lehigh.edu
- Source is available upon request
- All relevant information and instructions are in README.txt file
- Sample domain and problem input files in PDDL are provided
- Demonstration ...

Outline

FastForward

- Written by Jörg Hoffmann in 2000

FastForward

- Written by Jörg Hoffmann in 2000
- Winner of many contemporary planning competitions

FastForward

- Written by Jörg Hoffmann in 2000
- Winner of many contemporary planning competitions
- Another breadth-first, forward-searching state-space classical planner

FastForward

- Written by Jörg Hoffmann in 2000
- Winner of many contemporary planning competitions
- Another breadth-first, forward-searching state-space classical planner
- Sound and complete but not admissible

FastForward

- Written by Jörg Hoffmann in 2000
- Winner of many contemporary planning competitions
- Another breadth-first, forward-searching state-space classical planner
- Sound and complete but not admissible
- Much faster than VanillaIce due to heuristics, hill-climbing, and pruning strategies

Heuristic Properties

- A *heuristic* in general is a procedure that often performs well but has no guarantees

Heuristic Properties

- A *heuristic* in general is a procedure that often performs well but has no guarantees
- Usually involves solving a relaxed version of problem

Heuristic Properties

- A *heuristic* in general is a procedure that often performs well but has no guarantees
- Usually involves solving a relaxed version of problem
- In search problems, a heuristic is an estimate of how likely a node is to quickly lead to a solution

Heuristic Properties

- A *heuristic* in general is a procedure that often performs well but has no guarantees
- Usually involves solving a relaxed version of problem
- In search problems, a heuristic is an estimate of how likely a node is to quickly lead to a solution
- Such a heuristic allows the most promising nodes to be explored first

Heuristic Properties

- A *heuristic* in general is a procedure that often performs well but has no guarantees
- Usually involves solving a relaxed version of problem
- In search problems, a heuristic is an estimate of how likely a node is to quickly lead to a solution
- Such a heuristic allows the most promising nodes to be explored first
- Must be reasonably accurate and significantly cheaper to compute than solution to problem

Outline

Goal Distance

- So what does this mean for forward-search state-space planning?

Goal Distance

- So what does this mean for forward-search state-space planning?
  - How many actions must be applied to a state to achieve the goals

Goal Distance

- So what does this mean for forward-search state-space planning?
  - How many actions must be applied to a state to achieve the goals
- Why not compute this exactly?

Goal Distance

- So what does this mean for forward-search state-space planning?
  - How many actions must be applied to a state to achieve the goals
- Why not compute this exactly?
  - Only known way would give you the plan, meaning you have to solve the problem.

Terminology

- We shall use $h^*(s)$ to mean the actual minimum number of actions between a state and the goals

Terminology

- We shall use $h^*(s)$ to mean the actual minimum number of actions between a state and the goals
- $\Delta(s, p)$ is an estimated minimum distance from state s to a state containing predicate p

Terminology

- We shall use $h^*(s)$ to mean the actual minimum number of actions between a state and the goals
- $\Delta(s, p)$ is an estimated minimum distance from state s to a state containing predicate p
- $\Delta(s, g)$ is an estimated distance from state s to set g of predicates

Terminology

- We shall use $h^*(s)$ to mean the actual minimum number of actions between a state and the goals
- $\Delta(s, p)$ is an estimated minimum distance from state s to a state containing predicate p
- $\Delta(s, g)$ is an estimated distance from state s to set g of predicates
- $h(s)$ is an estimated heuristic value for state s

Relaxing Goal Distance

- What makes this so computationally expensive?

Relaxing Goal Distance

- What makes this so computationally expensive?
  - Selecting actions is a combinatorial problem because of interactions and deletions among methods

Relaxing Goal Distance

- What makes this so computationally expensive?
  - Selecting actions is a combinatorial problem because of interactions and deletions among methods
- How could we relax the problem but still have an informative heuristic?

Relaxing Goal Distance

- What makes this so computationally expensive?
  - Selecting actions is a combinatorial problem because of interactions and deletions among methods
- How could we relax the problem but still have an informative heuristic?
  - Ignore negative effects of actions, so state is constantly expanding towards goals

Relaxing Goal Independence

- Another relaxation is assuming goals are achieved independently

Relaxing Goal Independence

- Another relaxation is assuming goals are achieved independently
- $\Delta_0(s,p) = \begin{array}{cc} 0, & p \in s \\ \min_a\{1 + \Delta_0(s, \text{prec}(a)) | p \in \text{eff}^+(a)\}, & p \notin s \end{array}$

Relaxing Goal Independence

- Another relaxation is assuming goals are achieved independently
- $\Delta_0(s,p) = \begin{array}{cc} 0, & p \in s \\ \min_a\{1 + \Delta_0(s, \text{prec}(a)) | p \in \text{eff}^+(a)\}, & p \notin s \end{array}$
  - Recursive backward count from predicate to state

Relaxing Goal Independence

- Another relaxation is assuming goals are achieved independently
- $\Delta_0(s,p) = \begin{array}{cc} 0, & p \in s \\ \min_a\{1 + \Delta_0(s, \text{prec}(a)) | p \in \text{eff}^+(a)\}, & p \notin s \end{array}$
  - Recursive backward count from predicate to state
- $\Delta_0(s,g) = \sum_{p \in g} \Delta_0(s,p)$

Relaxing Goal Independence

- Another relaxation is assuming goals are achieved independently
- $\Delta_0(s,p) = \begin{array}{cc} 0, & p \in s \\ \min_a\{1 + \Delta_0(s, \text{prec}(a)) | p \in \text{eff}^+(a)\}, & p \notin s \end{array}$
  - Recursive backward count from predicate to state
- $\Delta_0(s,g) = \sum_{p \in g} \Delta_0(s,p)$
  - Simple summation of distances to each goal

Relaxing Goal Independence

- Another relaxation is assuming goals are achieved independently
- $\Delta_0(s,p) = \begin{array}{cc} 0, & p \in s \\ \min_a\{1 + \Delta_0(s, \text{prec}(a)) | p \in \text{eff}^+(a)\}, & p \notin s \end{array}$
  - Recursive backward count from predicate to state
- $\Delta_0(s,g) = \sum_{p \in g} \Delta_0(s,p)$
  - Simple summation of distances to each goal
- $h_0(s) = \Delta_0(s,g)$

Related Heuristics

- What if goals are not independent?

Related Heuristics

- What if goals are not independent?
  - $\Delta_1(s, g) = \max_{p \in g} \Delta_0(s, p)$

Related Heuristics

- What if goals are not independent?
  - $\Delta_1(s, g) = \max_{p \in g} \Delta_0(s, p)$
  - $\Delta_2(s, g) =$ shortest path to most difficult pair of goals

Related Heuristics

- What if goals are not independent?
  - $\Delta_1(s, g) = \max_{p \in g} \Delta_0(s, p)$
  - $\Delta_2(s, g) =$ shortest path to most difficult pair of goals
  - $\Delta_k(s, g) =$ shortest path to most difficult k goals

Related Heuristics

- What if goals are not independent?
  - $\Delta_1(s, g) = \max_{p \in g} \Delta_0(s, p)$
  - $\Delta_2(s, g) =$ shortest path to most difficult pair of goals
  - $\Delta_k(s, g) =$ shortest path to most difficult k goals
- Higher-order heuristics are more informative but harder to compute

Outline

Graphplan As Heuristic

- Expanding into larger states disregarding negative effects should look familiar

Graphplan As Heuristic

- Expanding into larger states disregarding negative effects should look familiar
- Graphplan does nearly the same thing, including but marking negative effects

Graphplan As Heuristic

- Expanding into larger states disregarding negative effects should look familiar
- Graphplan does nearly the same thing, including but marking negative effects
- Building planning graph to search in works similarly to exploring with heuristics to guide search

Graphplan In FastForward

- FastForward calls a special version of Graphplan to compute a heuristic value

Graphplan In FastForward

- FastForward calls a special version of Graphplan to compute a heuristic value
- Domain given to Graphplan contains no negative effects

Graphplan In FastForward

- FastForward calls a special version of Graphplan to compute a heuristic value
- Domain given to Graphplan contains no negative effects
  - Thus, no mutual exclusion among actions or literals

Graphplan In FastForward

- FastForward calls a special version of Graphplan to compute a heuristic value
- Domain given to Graphplan contains no negative effects
  - Thus, no mutual exclusion among actions or literals
  - Thus, no need for backtracking

Graphplan In FastForward

- FastForward calls a special version of Graphplan to compute a heuristic value
- Domain given to Graphplan contains no negative effects
  - Thus, no mutual exclusion among actions or literals
  - Thus, no need for backtracking
- Graphplan always chooses no-ops first

Graphplan In FastForward

- FastForward calls a special version of Graphplan to compute a heuristic value
- Domain given to Graphplan contains no negative effects
  - Thus, no mutual exclusion among actions or literals
  - Thus, no need for backtracking
- Graphplan always chooses no-ops first
  - Thus, actions appear only once

Graphplan In FastForward

- FastForward calls a special version of Graphplan to compute a heuristic value
- Domain given to Graphplan contains no negative effects
  - Thus, no mutual exclusion among actions or literals
  - Thus, no need for backtracking
- Graphplan always chooses no-ops first
  - Thus, actions appear only once
- Length of Graphplan's solution to relaxed problem from a node is heuristic value for that node

Relaxed Graphplan Complexity

- On relaxed problems, Graphplan runs in polynomial time in the number of actions, size of largest positive effects set, and size of initial state.

Relaxed Graphplan Complexity

- On relaxed problems, Graphplan runs in polynomial time in the number of actions, size of largest positive effects set, and size of initial state.
- Refer to the paper for proof

Relaxed Graphplan Complexity

- On relaxed problems, Graphplan runs in polynomial time in the number of actions, size of largest positive effects set, and size of initial state.
- Refer to the paper for proof
- Building the planning graph sounds expensive, but if this polynomial-time algorithms restricts the exponential state space, it will be a win

Example Notation

- We take advantage of lack of mutual exclusion to use a compact notation for large planning graphs

Example Notation

- We take advantage of lack of mutual exclusion to use a compact notation for large planning graphs
- Literals are only listed in the first level where they are true

Example Notation

- We take advantage of lack of mutual exclusion to use a compact notation for large planning graphs
- Literals are only listed in the first level where they are true
- Actions are only listed in the first level where they are applicable

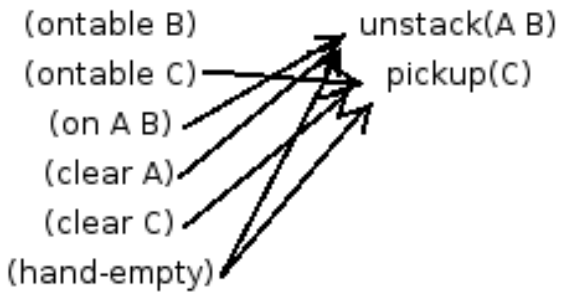
Example Notation

- We take advantage of lack of mutual exclusion to use a compact notation for large planning graphs
- Literals are only listed in the first level where they are true
- Actions are only listed in the first level where they are applicable
- Only new effects of actions are shown, in dashed lines

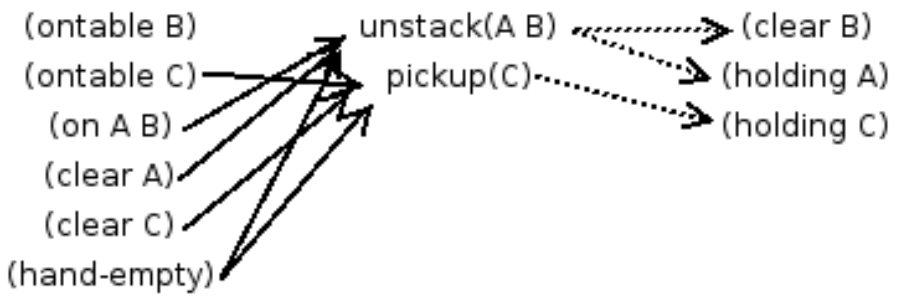
Example

(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)

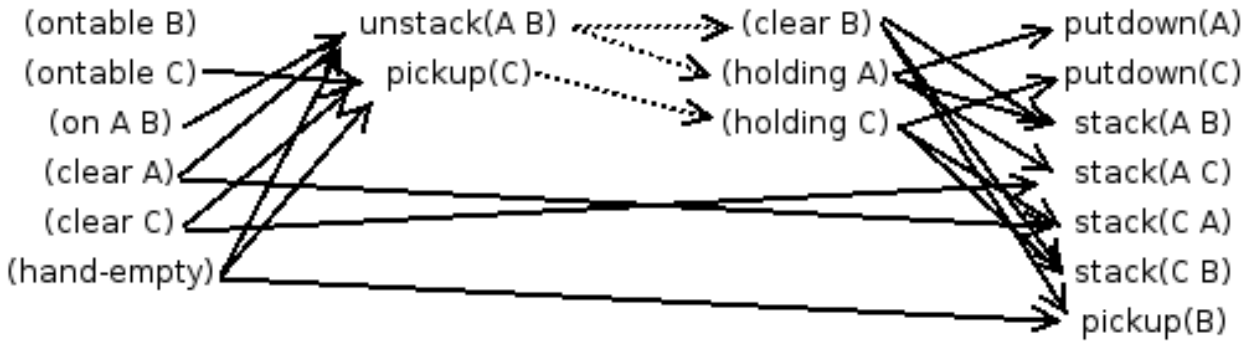
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)

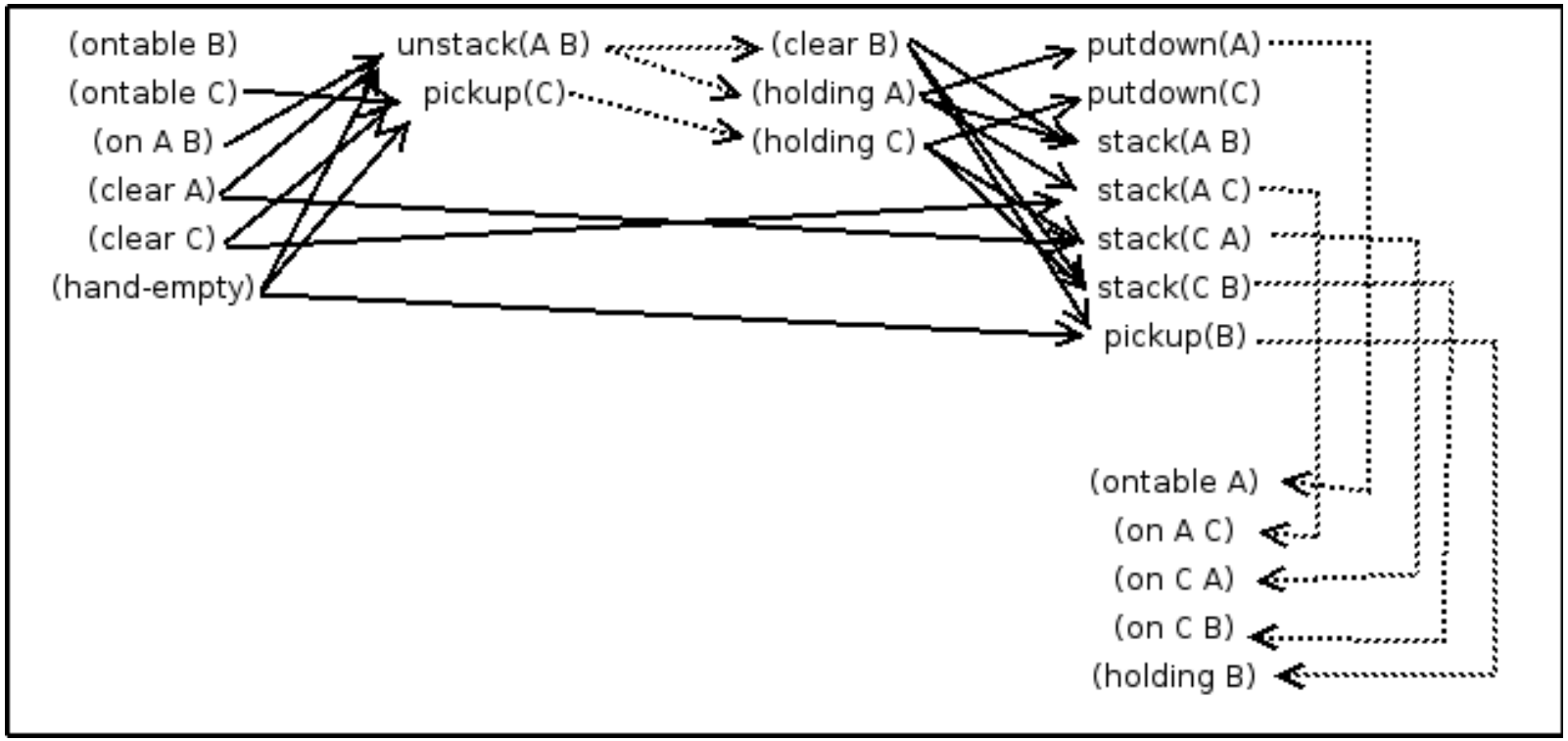
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)

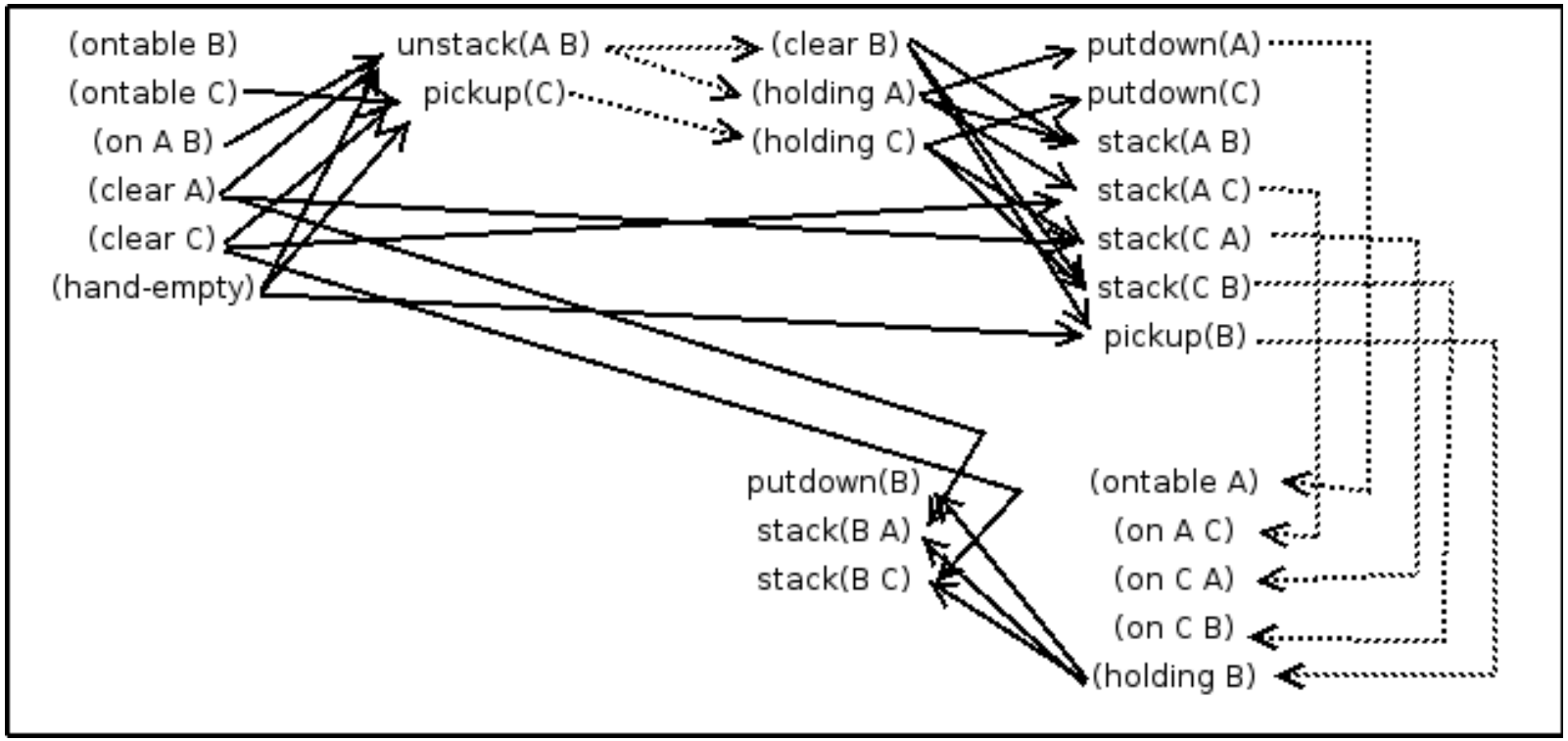
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)

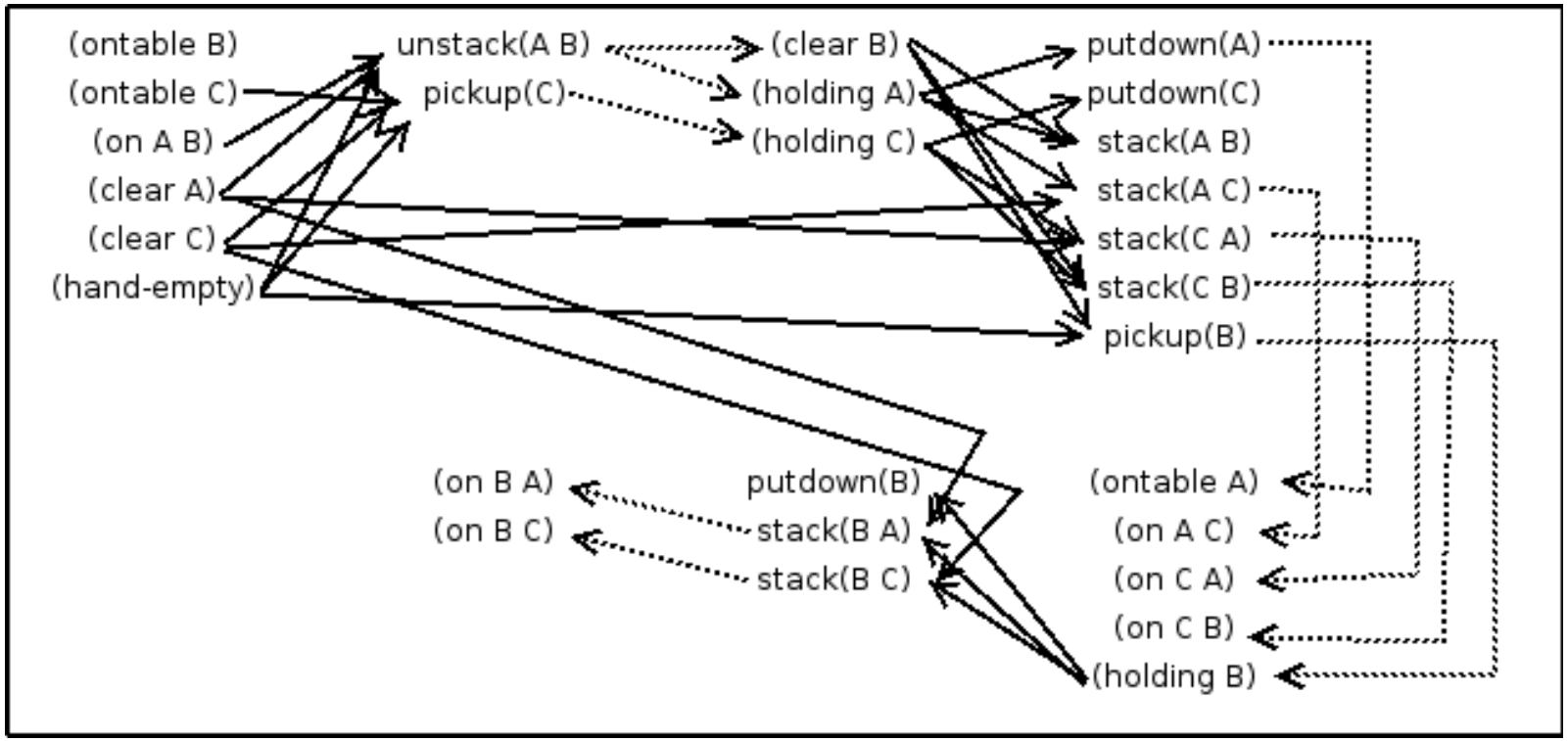
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

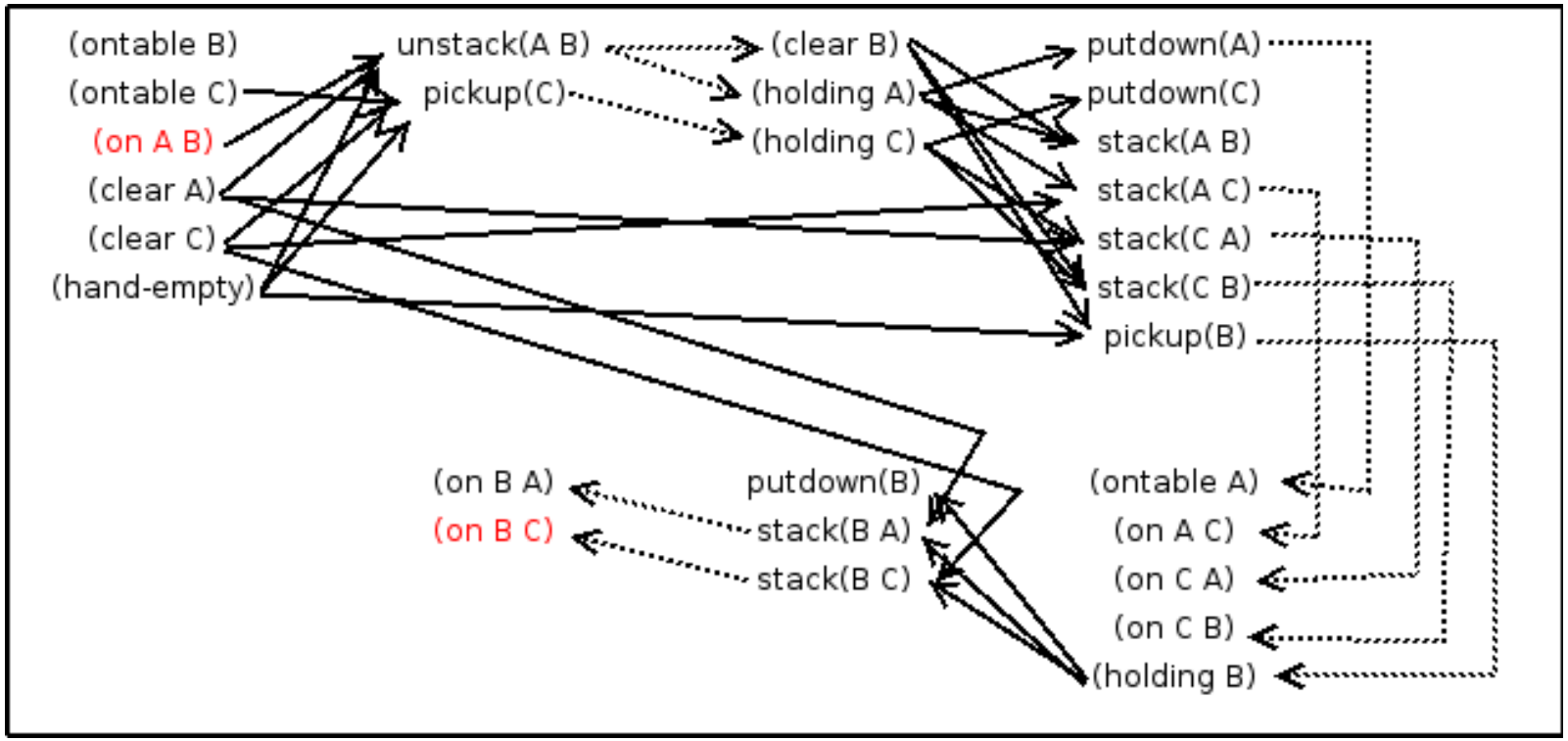
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
putdown(B)
stack(B A)
stack(B C)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

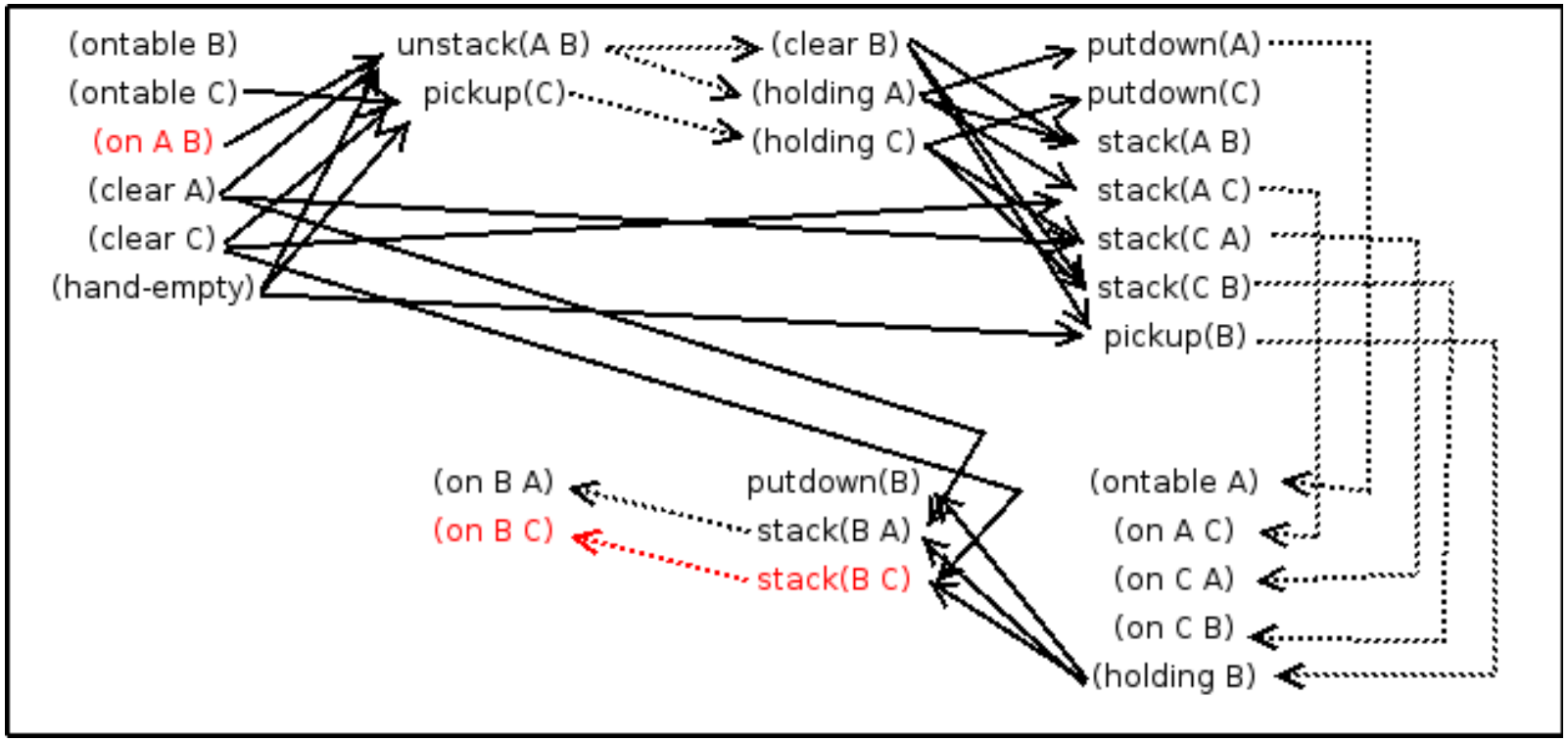
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
(on B A)
(on B C)
putdown(B)
stack(B A)
stack(B C)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

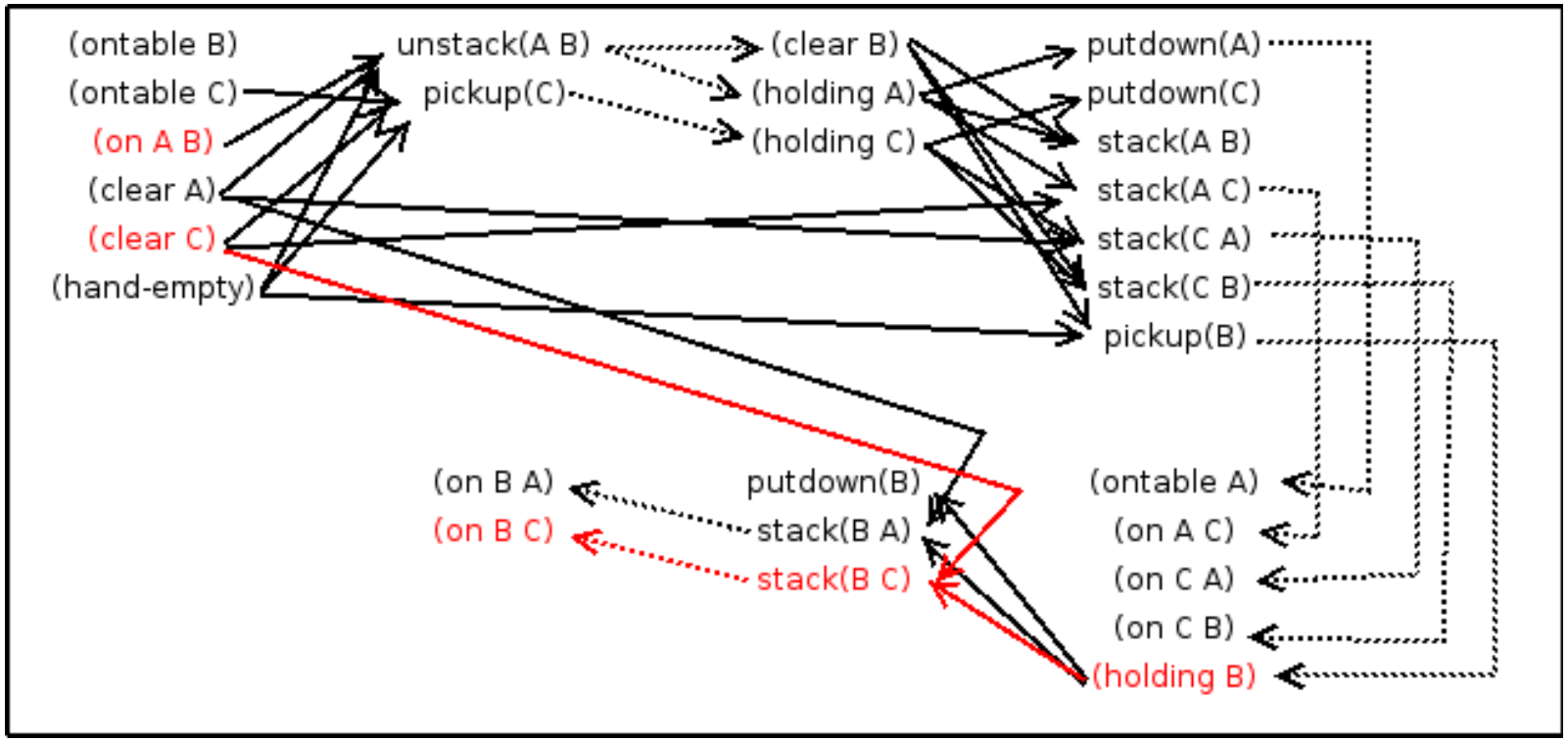
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
(on B A)
(on B C)
putdown(B)
stack(B A)
stack(B C)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

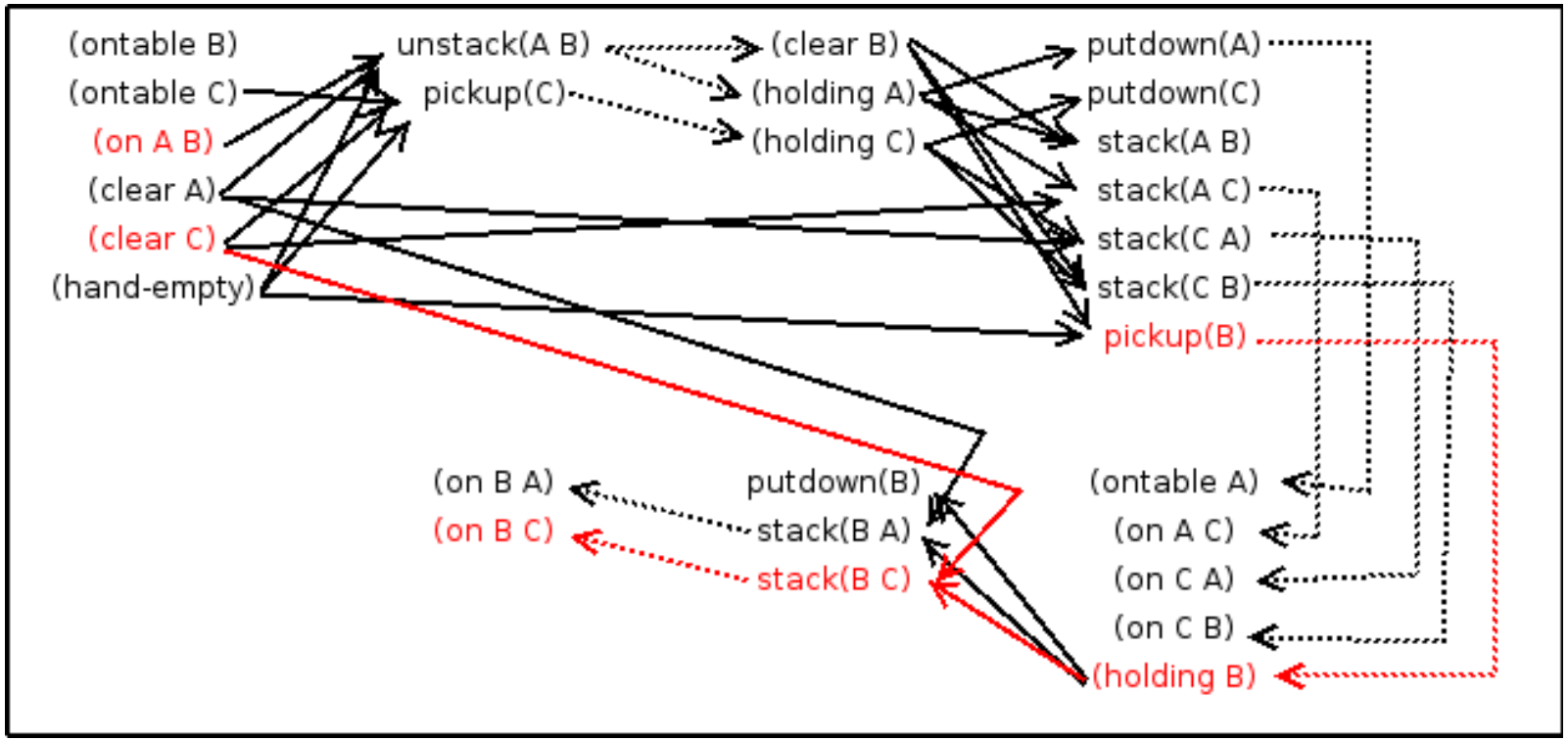
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
(on B A)
(on B C)
putdown(B)
stack(B A)
stack(B C)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

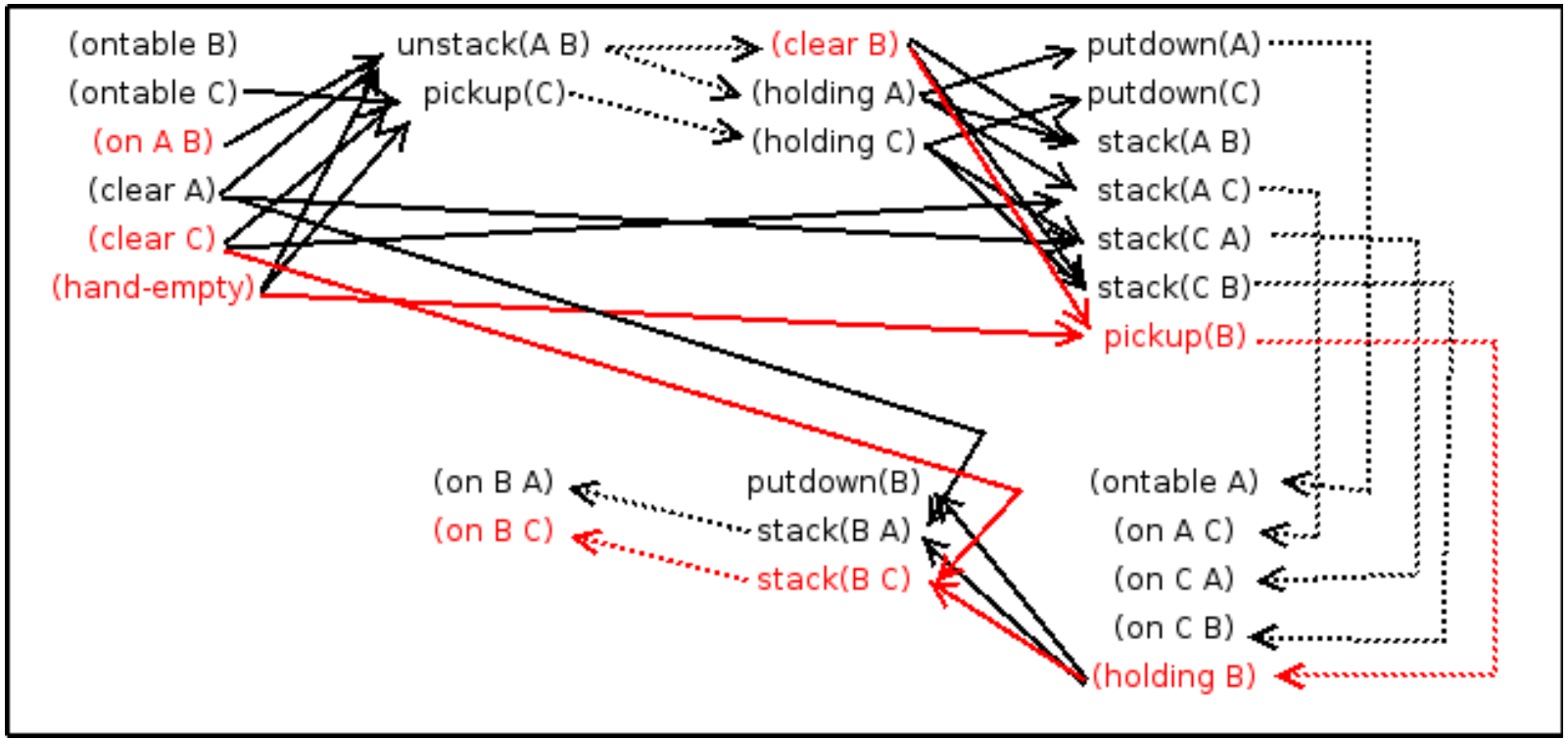
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
(on B A)
(on B C)
putdown(B)
stack(B A)
stack(B C)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

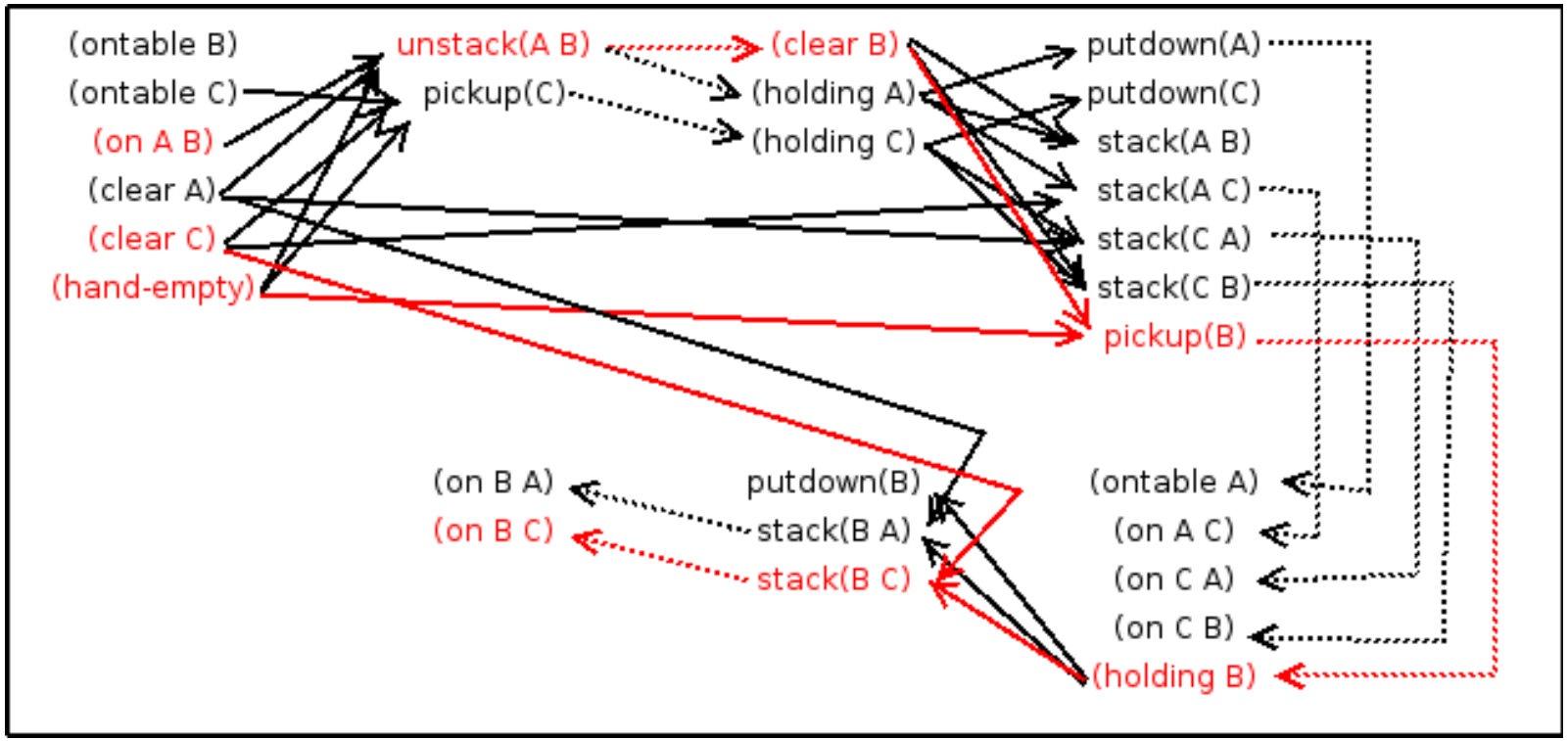
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
(on B A)
(on B C)
putdown(B)
stack(B A)
stack(B C)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

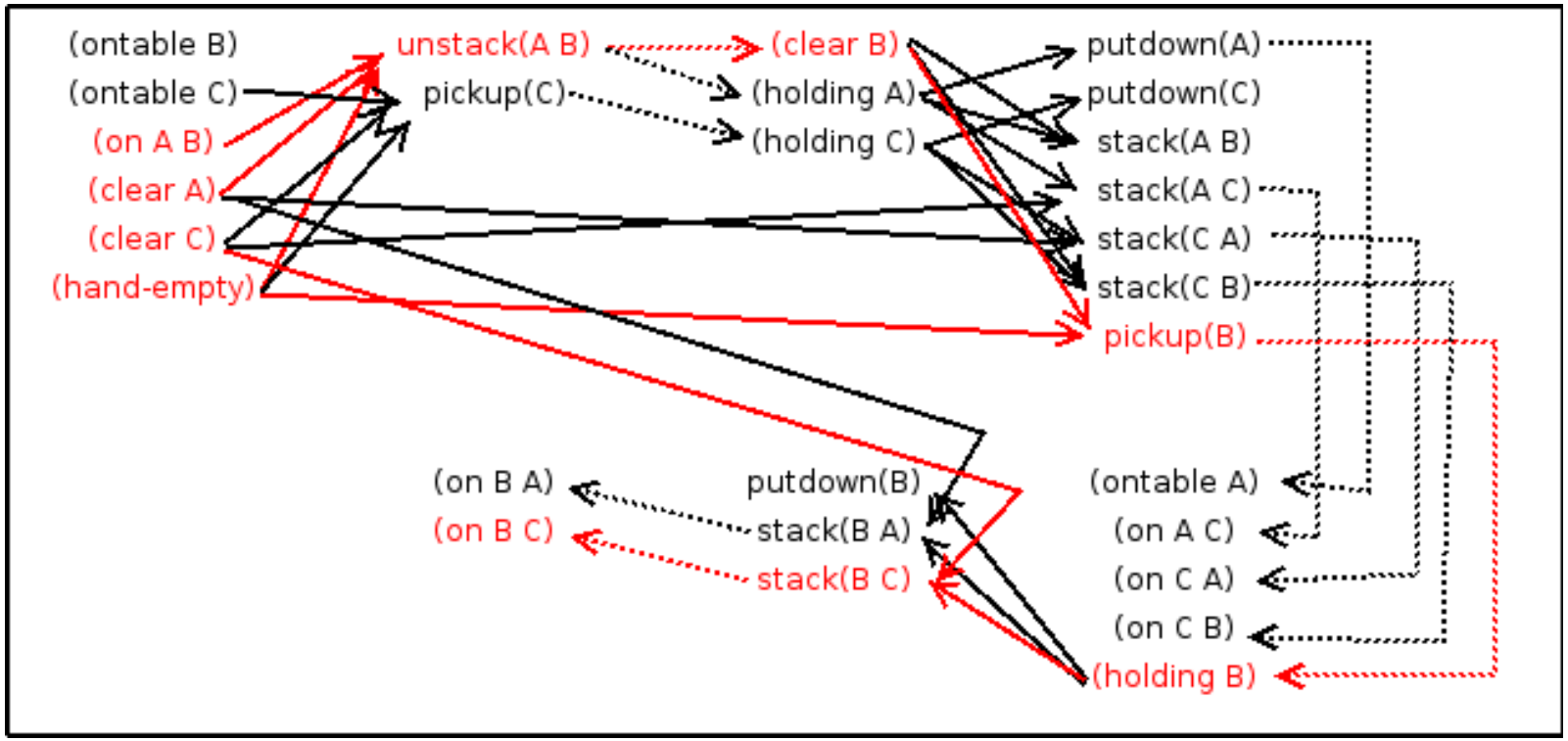
Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
(on B A)
(on B C)
putdown(B)
stack(B A)
stack(B C)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
(on B A)
(on B C)
putdown(B)
stack(B A)
stack(B C)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

Example


(ontable B)
(ontable C)
(on A B)
(clear A)
(clear C)
(hand-empty)
unstack(A B)
pickup(C)
(clear B)
(holding A)
(holding C)
putdown(A)
putdown(C)
stack(A B)
stack(A C)
stack(C A)
stack(C B)
pickup(B)
(on B A)
(on B C)
putdown(B)
stack(B A)
stack(B C)
(ontable A)
(on A C)
(on C A)
(on C B)
(holding B)

Example

- Graphplan's solution has three action levels

Example

- Graphplan's solution has three action levels
- Each level has one action: *unstack(A B)*, *pickup(B)*, *stack(B C)*

Example

- Graphplan's solution has three action levels
- Each level has one action: *unstack(A B)*, *pickup(B)*, *stack(B C)*
- Thus, $h(s_0) = 3$

Example

- Graphplan's solution has three action levels
- Each level has one action: *unstack(A B)*, *pickup(B)*, *stack(B C)*
- Thus, $h(s_0) = 3$
- This "plan" makes little sense, but it contains a sense of how much needs to be done

Break!

- When we return, we will discuss how FastForward uses this heuristic.

Hill-Climbing Defined

- A *hill-climbing* algorithm is a greedy search strategy that moves to the neighbor with highest value

Hill-Climbing Defined

- A *hill-climbing* algorithm is a greedy search strategy that moves to the neighbor with highest value
- Often finds a good local maxima, but not the optimal solution

Hill-Climbing Defined

- A *hill-climbing* algorithm is a greedy search strategy that moves to the neighbor with highest value
- Often finds a good local maxima, but not the optimal solution
- We want to minimize a heuristic, so perhaps “valley descending” would be more appropriate

Outline

Hill-Climbing in State Space

- *HSP* uses standard hill-climbing and a heuristic based on the relaxations previously discussed

Hill-Climbing in State Space

- *HSP* uses standard hill-climbing and a heuristic based on the relaxations previously discussed
- When no successor is a better state, and goals are not met, make arbitrary choice

Hill-Climbing in State Space

- *HSP* uses standard hill-climbing and a heuristic based on the relaxations previously discussed
- When no successor is a better state, and goals are not met, make arbitrary choice
- No backtracking, so a bad choice can make the problem unsolvable

Hill-Climbing in State Space

- *HSP* uses standard hill-climbing and a heuristic based on the relaxations previously discussed
- When no successor is a better state, and goals are not met, make arbitrary choice
- No backtracking, so a bad choice can make the problem unsolvable
- Sound, but not complete or admissible

Outline

Enforced Hill-Climbing

- FF uses a slightly modified hill-climbing algorithm

Enforced Hill-Climbing

- FF uses a slightly modified hill-climbing algorithm
- Instead of choosing the best successor, perform breadth-first search for the first strictly better descendent

Enforced Hill-Climbing

- FF uses a slightly modified hill-climbing algorithm
- Instead of choosing the best successor, perform breadth-first search for the first strictly better descendent
- Less likely to randomly wander around plateaus

Enforced Hill-Climbing

- FF uses a slightly modified hill-climbing algorithm
- Instead of choosing the best successor, perform breadth-first search for the first strictly better descendent
- Less likely to randomly wander around plateaus
- Also sound but neither complete nor admissible

Outline

Outline