Chapter S:III

III. Informed Search

- ❑ Best-First Search Basics
- ❑ Best-First Search Algorithms
- ❑ Cost Functions for State-Space Graphs
- ❑ Evaluation of State-Space Graphs
- ❑ Algorithm A*

- ❑ BF* Variants
- ❑ Hybrid Strategies

- ❑ Best-First Search for AND-OR Graphs
- ❑ Relation between GBF and BF
- ❑ Cost Functions for AND-OR Graphs
- ❑ Evaluation of AND-OR Graphs

Best-First Search Basics

"To enhance the performance of AI's programs, knowledge [about the problem domain, which enables us to guide search into promising directions] *is the power."*

[Feigenbaum 1980]

Best-First Search

"To enhance the performance of AI's programs, knowledge [about the problem domain, which enables us to guide search into promising directions] *is the power."*

[Feigenbaum 1980]

Best-First Search

"To enhance the performance of AI's programs, knowledge [about the problem domain, which enables us to guide search into promising directions] *is the power."*

[Feigenbaum 1980]

Examples for heuristic functions [S:I Examples for Search Problems] :

- 8-Queens problem. Maximize h_1, the number of unattacked cells.
- 8-Puzzle problem. Minimize h_1, the number of non-matching tiles.

Knowledge on how to achieve this (Maximize. . . , Minimize. . .) is beyond that which is built into the state and operator definitions.

Best-First Search Basics

"To enhance the performance of AI's programs, knowledge [about the problem domain, which enables us to guide search into promising directions] *is the power."*

[Feigenbaum 1980]

Examples for heuristic functions [S:I Examples for Search Problems] :

- 8-Queens problem. Maximize h_1, the number of unattacked cells.
- 8-Puzzle problem. Minimize h_1, the number of non-matching tiles.

Knowledge on how to achieve this (Maximize. . . , Minimize. . .) is beyond that which is built into the state and operator definitions.

Where is heuristic knowledge employed in the formalism of systematic search?

- Greedy Search. Move into the direction of a most promising successor n' of the current node.
- Best-First Search. Move into the direction of a most promising node n, where n is chosen among all nodes encountered so far.

Best-First Search Basics

"The promise of a node is estimated numerically by a heuristic evaluation function $f(n)$ *which, in general, may depend on the description of* n*, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain."*

[Pearl 1984]

Best-First Search Basics

"The promise of a node is estimated numerically by a heuristic evaluation function $f(n)$ which, in general, may depend on the description of n, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain."

[Pearl 1984]

The evaluation function f may depend on

1. evaluation of the state information given by n,
2. estimates of the complexity of the remaining problem at n in relation to Γ,
3. evaluations of the explored path to n in the search space graph G,
4. domain specific problem solving knowledge K about G.

$$f = f(n, \Gamma, G, K)$$

Objective is to quantify for a generated, but yet unexpanded node n its potential of guiding the search into a desired direction.

Remarks:

- Node n represents a solution base. Therefore, n gives access to information about a path from s to the state represented by n.

- The remaining problem is the problem of determining a solution path for the state in G given by node n. Using such path as a continuation of the solution base given by n, a solution path for s is given. The complexity estimation of the remaining problem is beyond the information encoded in nodes and edges.

- Knowing that G is Euclidean is an example for domain specific problem solving knowledge. Euclidean distances can be used for estimating remaining path length.

- Evaluation functions are domain dependent. Therefore, these functions (or parts of them) will be provided to search algorithms as parameters.

- We could think of doing even more: An evaluation of a solution base by f could also depend on the explored part of the search space graph G, e.g., the emphasis on specific knowledge in the computation of f could be changed thereby. However, such a dependence would require an update of computed f-values (highly inefficient) each time the explored part of G changes.

Best-First Search Basics

Generic Schema for Best-First Algorithms

. . . from a solution-base-oriented perspective:

1. Initialize a solution base storage.
2. Loop.
   (a) Select a most promising solution base using an evaluation function f.
   (b) Expand the only unexpanded node in the solution base.
   (c) Extend the solution base by one successor node at a time and save it as a new candidate.
   (d) Determine whether a solution path has been found.

Usage:

- Node expansion is used as basic step.
- Best-First Algorithms are informed versions of Basic_OR.

Remarks:

- The schema is further extended by termination tests for failure and success.
- The job of the evaluation function is to make two solution bases comparable and hence to provide an order on them.
- Best-first strategies differ in the evaluation functions they use. Placing restrictions on the computation of these functions will establish a taxonomy of best-first algorithms.
- Even when considering constraint satisfaction problems it makes sense to use best-first algorithms. The paradigm "Small is quick!" follows the idea that low cost values will be assigned to solutions with simple structure and that simple structures can be established in a few steps, i.e., in short time.

Chapter S:III

III. Informed Search

- ❑ Best-First Search Basics
- ❑ Best-First Search Algorithms
- ❑ Cost Functions for State-Space Graphs
- ❑ Evaluation of State-Space Graphs
- ❑ Algorithm A*

- ❑ BF* Variants
- ❑ Hybrid Strategies

- ❑ Best-First Search for AND-OR Graphs
- ❑ Relation between GBF and BF
- ❑ Cost Functions for AND-OR Graphs
- ❑ Evaluation of AND-OR Graphs

Best-First Search Algorithms

Notation for Evaluation Functions

- Evaluation functions f are specifically designed for a state-space G.

  This dependency is usually clear from the context. If not, we will use different names $f, f', \ldots$ to distinguish between evaluation functions for different state spaces.

- An evaluation function f for G uses information on a solution base P for some state s in G (a path in G from s to some other state s', the tip-node of the solution base) and knowledge about G.

$f(P)$ From a state-space graph perspective, function f must have a path parameters P that defines a solution base. As f is specific for G no further information is needed.

$f(n)$ From a back-pointer structure oriented perspective, it is enough to provide a node n as argument of f. The back-pointer path defines the solution base to consider.

- In property definitions for f we take a back-pointer perspective although f should be defined as function on paths of G.

  Nodes and back-pointer paths have to be seen as part of any back-pointer structure that is theoretically constructible and meaningful. These structures are NOT restricted to be back-pointer structures produced by some algorithm at some point in time.

  "For all nodes $n \ldots$" therefore has the same meaning as "For all solution bases P for $s \ldots$".

Best-First Search Algorithms

Notation for Evaluation Functions (continued)

Definition 20 (Evaluation Function f)

Let G be state-space graph.

An *evaluation function* f is a function that assigns values $f(n)$ in an ordered set to paths P in G, where paths P are given as back-pointer paths of nodes n.

- We use the extended real numbers $\overline{\mathbf{R}} = \mathbf{R} \cup \{-\infty, +\infty\}$ and the $\leq$-relation as ordered set.
- Evaluation functions f used in algorithm BF is designed for a specific state-space G. f is, therefore, highly domain dependent.
- Algorithms will usually consider only paths starting in s and states on such paths that are available in its back-pointer structure at some point in time. These values will be denoted by $f(n)$ for some node n.

Best-First Search Algorithms

Basic Principles for an Algorithmization of Best-First Search

Prop$_1(G)$ Required Properties of G for Best-First Search

1. G has *Prop*$_0(G)$ properties.
2. Evaluation function f is defined for G and assigns cost values to paths in G.
3. f is computable.
4. When f evaluates a solution bases P_{s-n}, the computed value does not depend on the time of computation.
5. When f evaluates a solution bases P_{s-n}, f estimates optimum cost of solution paths that have P_{s-n} as initial part.
6. A most promising solution base has a minimum f-value in a candidate set.

Task

- Determine a solution paths for s in G.

Algorithmization:

- A most promising solution base is searched *among all solution bases* currently maintained by an algorithm.

Remarks:

- Best-first algorithms for state-space graphs are variants of algorithm Basic-OR. So, the solution bases P_{s-n} under consideration are defined by the states and back-pointers stored with the nodes in OPEN or CLOSED.

- If a dead-end recognition $\perp(n)$ is available, no solution base will be considered that contains an inner node labeled "unsolvable" using $\perp(n)$. A dead end recognition $\perp(.)$ can be integrated in f by setting $f(n) = \infty$ if $\perp(n)$ is true.

- Usually, the evaluation function $f(n)$ is based on a heuristic $h(n)$.

  $h(n)$ estimates the optimum cost of a solution path for the rest problem associated with a node n. Ideally, $h(n)$ should consider the probability of the solvability of the problem at node n.

Best-First Search Algorithms

Algorithm: Basic-BF (Compare BFS, BF, Basic-BF*)

Input: s. Start node representing the initial state (problem) in G.
successors(n). Returns *new instances of* nodes for the successor states in G.
$\star(n)$. Predicate that is *True* if n represents a goal state in G.
constraints(n). Predicate that is *True* if path repr. by n satisfies solution constraints.
$f(n)$. Evaluation function (cost) for the solution base in G represented by n.

Output: A node γ representing a solution path for s in G or the symbol *Fail*.

```
Basic-BF(s, successors, ⋆, constraints, f)   // A deterministic variant of Basic-OR.

1. s.parent = null;  add(s, OPEN, f(s));   // Store s on f-sorted OPEN.
2. LOOP
3.   IF (OPEN == ∅) THEN RETURN(Fail);
4.   n = min(OPEN, f);    // Find most promising (cheapest) solution base.
     remove(n, OPEN);  add(n, CLOSED);
5.   FOREACH n' IN successors(n) DO    // Expand n.
       n'.parent = n;
       IF ⋆(n') THEN
         IF constraints(n') THEN RETURN(n');
       add(n', OPEN, f(n'));    // Store n' on f-sorted OPEN.
     ENDDO
6. ENDLOOP
```


Remarks:

- Operationalization of best-first search:
  The function *add*$(n, \texttt{OPEN}, f(n))$ stores a node n according to $f(n)$ in the underlying data structure of the OPEN list. Using a sorted tree (a heap), a node with the minimum f-value is found in logarithmic (constant) time. [OPEN list in DFS] [OPEN list in BFS]

- Since f-values do not change over time, they can be stored with the nodes once computed.

- In all the following algorithms we can make use of dead-end functions $\perp(n)$.

- In addition, memory consumption can be reduced by using *cleanup_closed* in the case of nodes without successors. To save room, we will not include these parts in the pseudocode.

Best-First Search Algorithms

Uniform-Cost Search (UCS) as Variant of Basic-BF

Setting:

- The search space graph G contains several solution paths.
- f assigns cost values to solution bases that do not include future cost for extending a solution base to a solution path:

  $f(n)$ = cost of path P_{s-n}

Task:

- Determine a cheapest path from s to some goal $\gamma \in \Gamma$.

Necessary Prerequisite:

- The cost of a solution base is a lower bound for the cheapest solution cost that can be achieved by completing the solution base.

➜ UCS will search G in layers of (nearly) equal cost and
UCS is complete, if G with *Prop*$_0(G)$ is finite + cycle-free, and
UCS will – sometimes – find optimum solution paths in this case.

Remarks:

- Uniform-cost search is also called cheapest-first search.
- A specific cost concept is to assign cost values to edges in search space graphs. A path's cost can be calculated as the sum or as the maximum of the cost values of its edges.

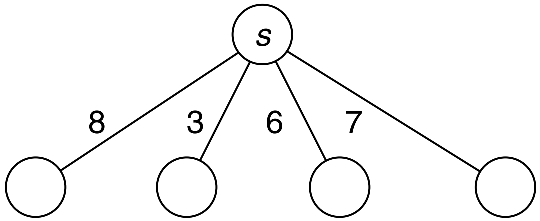
  If edge cost values are limited to non-negative numbers, the path cost of a solution base is an optimistic estimate of a cheapest solution path cost achievable by continuing that solution base.
- Depending on the state-space, the last step to a goal node could be quite expensive. Since delayed termination is not implemented, UCS immediately terminates when finding such a goal node, perhaps returning a suboptimal solution.
- If we have no means to calculate cost values for solution bases or if the cost of a solution base not guaranteed to be a lower bound for the cheapest solution cost that can be achieved by completing the solution base, the algorithm can surely know a minimum cost solution path, only if the set of solution bases in OPEN is exhausted.

Best-First Search Algorithms

Example: Uniform-Cost Search for Optimization

Determine the minimum column sum of a matrix:

8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6

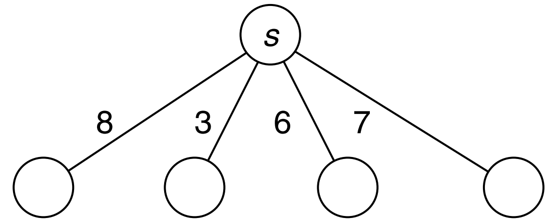


Best-First Search Algorithms

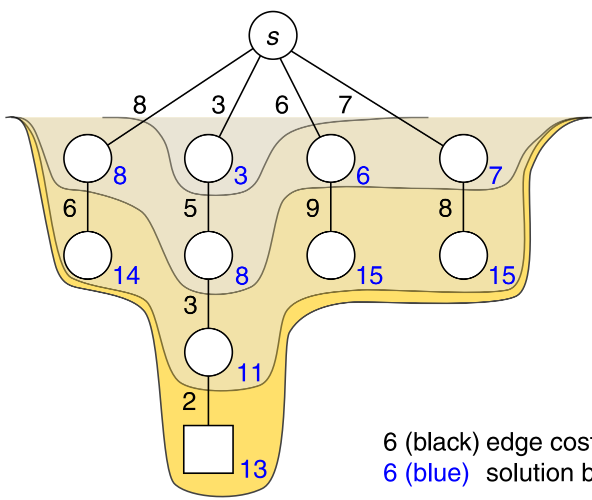
Example: Uniform-Cost Search for Optimization

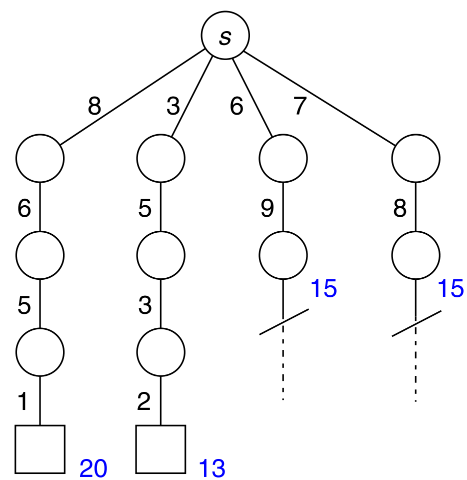
Determine the minimum column sum of a matrix:

8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6



Comparison of UCS (left) and DFS (right):





6 (black) edge cost
6 (blue) solution base cost

Best-First Search Algorithms

Uniform-Cost Search is an uninformed (systematic) search strategy.

Uniform-cost search characteristics:

- Node expansion happens in levels of equal costs:

  A node n with $f(n) = \textit{cost}(n)$ will not be expanded as long as a non-expanded node n' with $f(n') = \textit{cost}(n') < \textit{cost}(n) = f(n)$ resides on the OPEN list.

≈ UCS can be seen as application of the BFS strategy to solve optimization problems (using cost instead of depth).

≈ BFS can be seen as a UCS variant using $f(n) = \textit{depth}(n)$.
DFS can be seen as a UCS variant using $f(n) = -\textit{depth}(n)$.

- The optimistic cost estimation is crucial also for the correctness of the Uniform-Cost Search algorithm:

  If the cheapest solution cost that can be achieved by completing the solution base is overestimated we might miss an optimum cost solution path.

Best-First Search Algorithms

Delayed Termination: Basic-BF for Optimization

In general, the first solution found by algorithm Basic-BF may not be optimum with respect to the evaluation function f.

Important preconditions for (provably) finding optimum solution paths:

1. The cost estimate underlying f must be optimistic, i.e., underestimating costs or overestimating merits.

   In particular, the true cost $f_{P_{s-\gamma}}(\gamma)$ of a cheapest solution path $P_{s-\gamma}$ extending a solution base P_{s-n} exceeds its f-value: $f_{P_{s-\gamma}}(\gamma) \geq f_{P_{s-n}}(n)$ (➜ domain-dependent).

2. The termination in case of success ($\star(n) =$ *True*) must be delayed.

   In particular, there is no termination test when reaching a node, but each time when choosing a node from the OPEN list (➜ easily implemented).

➜ Algorithms using delayed termination are indicated by a star (*), Basic-BF becomes Basic-BF*.

Best-First Search Algorithms

Algorithm: Basic-BF* (Compare BF*.)

Input: s. Start node representing the initial state (problem) in G.
successors(n). Returns *new instances of* nodes for the successor states in G.
$\star(n)$. Predicate that is *True* if n represents a goal state in G.
$f(n)$. Evaluation function (cost) for the solution base in G represented by n.

Output: A node γ representing an (optimum) solution path for s in G or the symbol *Fail*.

Basic-BF*$(s, \textit{successors}, \star, f)$ // A delayed termination variant of Basic-BF.

s.*parent* $=$ *null*; *add*$(s, \texttt{OPEN}, f(s))$;
LOOP
  IF $(\texttt{OPEN} == \emptyset)$ THEN RETURN(*Fail*);
  $n = \textit{min}(\texttt{OPEN}, f)$;
  IF $\star(n)$ THEN RETURN(n); // Delayed termination.
  remove$(n, \texttt{OPEN})$; *add*$(n, \texttt{CLOSED})$;
  FOREACH n' IN *successors*(n) **DO** // Expand n.
    n'.*parent* $= n$;
    ~~IF $\star(n')$ THEN RETURN(n');~~ // Early termination removed.
    add$(n', \texttt{OPEN}, f(n'))$;
  ENDDO
ENDLOOP

Remarks:

- If the evaluation function f depends on the evaluations of the explored part G of the search space graph ONLY, f is uninformed and algorithm Basic-BF* performs a uniform-cost search with delayed termination.
- In the problem "minimum column sum of a matrix" the evaluation function $f(n)$ which returns the sum of column entries up to n is optimistic if matrix entries are nonnegative. In this case, algorithm Basic-BF* returns an optimum column.

Best-First Search Algorithms

Space Efficiency of Basic-BF and Basic-BF*

Approach:
Instead of storing all known paths to a node, only a most promising one is kept.

An implementation of this principle is called path discarding (aka parent discarding).

➜ Basic-BF with path discarding is called BF,
BF with delayed termination is called BF*.

Important preconditions for (provably) finding optimum solution paths in OR-graphs by best-first algorithms:

1. The cost estimate underlying f must be order-preserving, i.e., a solution base for a node n that is more promising than some other solution base for n will lead to a solution path which is not inferior to solution paths reached by extending the inferior solution base.

2. In particular, cyclic paths should not be considered.

3. When defining a tie breaking strategy for OPEN, goal nodes must be preferred.

Best-First Search Algorithms

Implementing Path Discarding in Basic-BF

```
BF(s, successors, ⋆, f)    // An path discarding variant of Basic-BF.

  1.  s.parent = null;  add(s, OPEN, f(s));
  2.  LOOP
  3.    IF (OPEN == ∅) THEN RETURN(Fail);
  4.    n = min(OPEN, f);    // Find most promising (cheapest) solution base.
        remove(n, OPEN);  add(n, CLOSED);
  5.    FOREACH n' IN successors(n) DO    // Expand n.
          n'.parent = n;
          IF ⋆(n') THEN RETURN(n');
          n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
          IF ( n'_old == null )
          THEN   // n' not in OPEN or CLOSED:  n' refers to a new state.
            add(n', OPEN, f(n'));
          ELSE   // n' refers to an already visited state.
            IF ( f(n') < f(n'_old) )    // Compare cost of solution bases.
            THEN   // Solution base of n' is cheaper:  path discarding.
              n'_old.parent = n'.parent;  f(n'_old) = f(n');
              IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED);  add(n'_old, OPEN, f(n'_old));  ENDIF
            ENDIF
          ENDIF
        ENDDO
  6.  ENDLOOP
```

Remarks:

- The function *retrieve*$(n', \texttt{OPEN} \cup \texttt{CLOSED})$ retrieves (without removing) a previously stored node instance from OPEN resp. CLOSED referring to the same state in G as n'.
- Due to space limitations the above algorithm does not mention that the new instance of a node n' that has a counterpart in OPEN or CLOSED has to be removed. BF always keeps of all node instances referring to the same state only that one that was generated first.
- Statement $f(n'_{old}) = f(n')$ in algorithm BF is to be understood in the sense that old f-values that have been stored (with the nodes) are overwritten. Not only the new parent reference, also the new f-value is kept.
- The updating of back-pointers performed by BF algorithms preserves the structure of the traversal tree (maintained by BF via nodes stored in OPEN and CLOSED and back-pointers) at any point in time t.
  At each point in time (i.e., each time that the algorithm is at the beginning of the main loop) BF has a traversal tree at hand which is a subtree of G rooted in s.

Remarks:

- Path discarding entails the risk of not finding desired solutions. The risk can be eliminated by restricting to evaluation functions f that fulfill particular properties. Keyword: *Order preserving property* [Specialized Cost Measures]

- If cyclic paths have smaller f-values than corresponding cyclefree paths, the back-pointer structure will be corrupted when a cycle is found.

- As a consequence of path discarding *at most one solution base* for each state in G,

- As a consequence of path discarding, for two paths leading to the same node, the one with the higher f-value is discarded.

Best-First Search Algorithms

Path Discarding for a Node n'

```
5.    FOREACH n' IN successors(n) DO    // Expand n.
        ...
        n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
        IF ( n'_old == null )
        ...
        ELSE
          IF ( f(n') < f(n'_old) )    // Compare cost of solution bases.
          THEN    // Solution base of n' is cheaper:  path discarding.
            n'_old.parent = n'.parent;  f(n'_old) = f(n');
            IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED);  add(n'_old, OPEN, f(n'_old)); ENDIF
        ENDIF
```

- $f(n')$ is computed using the new node instance n' and the back-pointer path from s to n' via its parent n.
- $f(n'_{old})$ is computed using the old node instance n' and the back-pointer path from s to n'_{old}.
- n' and n'_{old} are referring to the same state in G.
- Path discarding is performed implicitly by maintaining at most one node instance referring to some state and, therefore, maintaining at most one back-pointer, i.e., at most one path.
- Algorithm BF cannot recover paths that were discarded, i.e., path discarding is irrevocable.
- f-values do not change over time. Once computed, f-values are stored with the nodes.

Best-First Search Algorithms

Re-evaluation of a Node n'

Case 1: n'_{old} is still on OPEN.

```
5.    FOREACH n' IN successors(n) DO    // Expand n.
        ...
        n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
        IF ( n'_old == null )
        ...
        ELSE
          IF ( f(n') < f(n'_old) )    // Compare cost of solution bases.
          THEN   // Solution base of n' is cheaper:  path discarding.
            n'_old.parent = n'.parent;  f(n'_old) = f(n');
            IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED);  add(n'_old, OPEN, f(n'_old)); ENDIF
        ENDIF
```

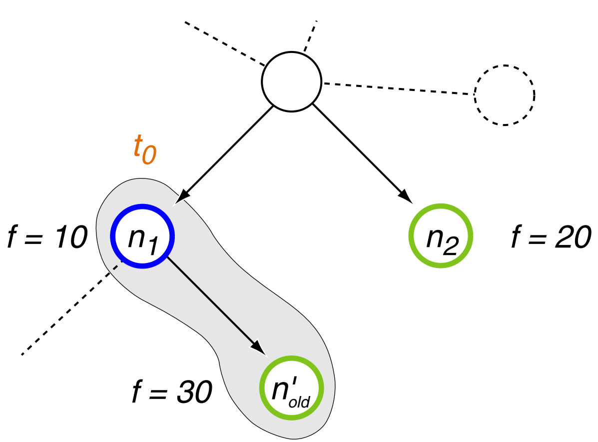
Best-First Search Algorithms

Re-evaluation of a Node n'

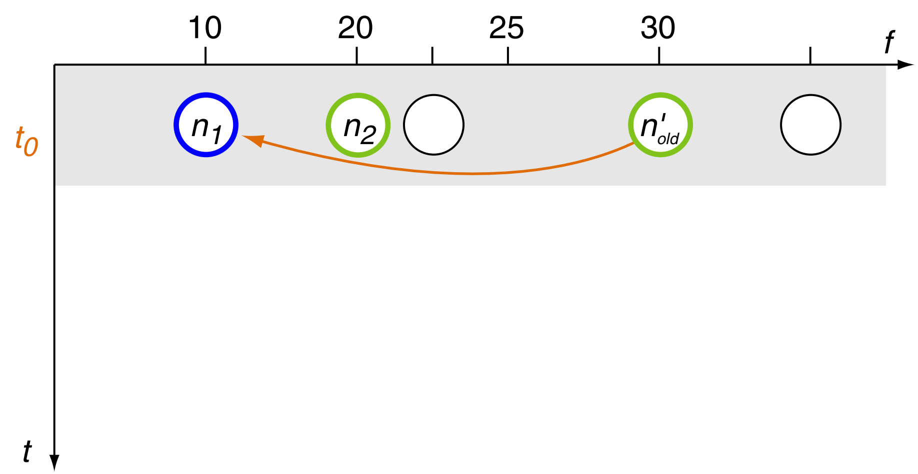
Case 1: n'_{old} is still on OPEN.

```
5.    FOREACH n' IN successors(n) DO        // Expand n.
        ...
        n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
        IF ( n'_old == null )
        ...
        ELSE
          IF ( f(n') < f(n'_old) )    // Compare cost of solution bases.
          THEN   // Solution base of n' is cheaper:  path discarding.
            n'_old.parent = n'.parent;  f(n'_old) = f(n');
            IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED); add(n'_old, OPEN, f(n'_old)); ENDIF
        ENDIF
```

State-space:



OPEN ∪ CLOSED list:



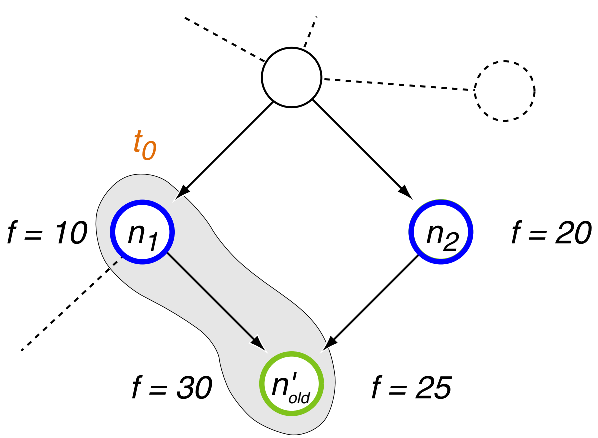
Best-First Search Algorithms

Re-evaluation of a Node n'

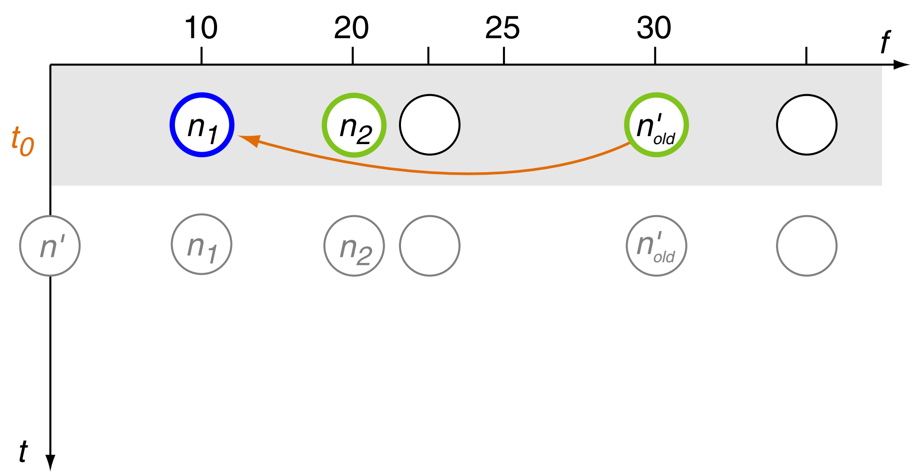
Case 1: n'_{old} is still on OPEN.

```
5.     FOREACH n' IN successors(n) DO       // Expand n.
         ...
         n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
         IF ( n'_old == null )
         ...
         ELSE
           IF ( f(n') < f(n'_old) )     // Compare cost of solution bases.
           THEN   // Solution base of n' is cheaper: path discarding.
             n'_old.parent = n'.parent;  f(n'_old) = f(n');
             IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED); add(n'_old, OPEN, f(n'_old)); ENDIF
         ENDIF
```

State-space:



OPEN ∪ CLOSED list:



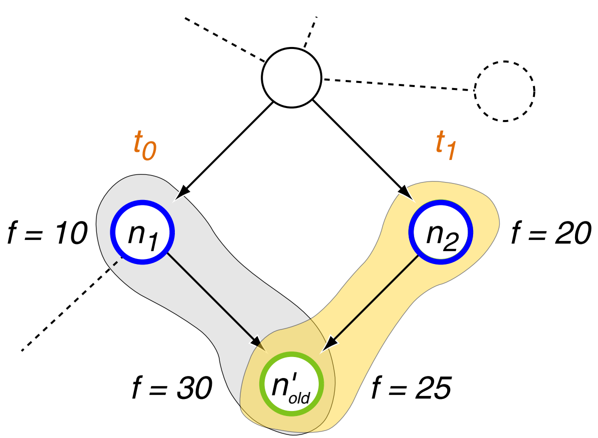
Best-First Search Algorithms

Re-evaluation of a Node n'

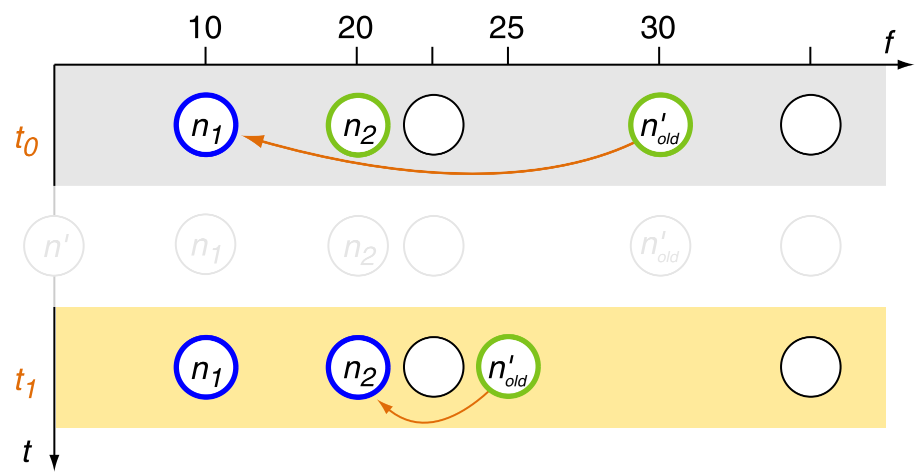
Case 1: n'_{old} is still on OPEN.

```
5.    FOREACH n' IN successors(n) DO     // Expand n.
        ...
        n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
        IF ( n'_old == null )
        ...
        ELSE
          IF ( f(n') < f(n'_old) )    // Compare cost of solution bases.
          THEN   // Solution base of n' is cheaper:  path discarding.
            n'_old.parent = n'.parent;  f(n'_old) = f(n');
            IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED);  add(n'_old, OPEN, f(n'_old)); ENDIF
        ENDIF
```

State-space:



OPEN ∪ CLOSED list:



Best-First Search Algorithms

Re-evaluation of a Node n' (continued)

Case 2: n'_{old} is already on CLOSED.

```
5.    FOREACH n' IN successors(n) DO      // Expand n.
        ...
        n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
        IF ( n'_old == null )
        ...
        ELSE
          IF ( f(n') < f(n'_old) )    // Compare cost of solution bases.
          THEN   // Solution base of n' is cheaper:  path discarding.
            n'_old.parent = n'.parent;  f(n'_old) = f(n');
            IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED);  add(n'_old, OPEN, f(n'_old));  ENDIF
        ENDIF
```

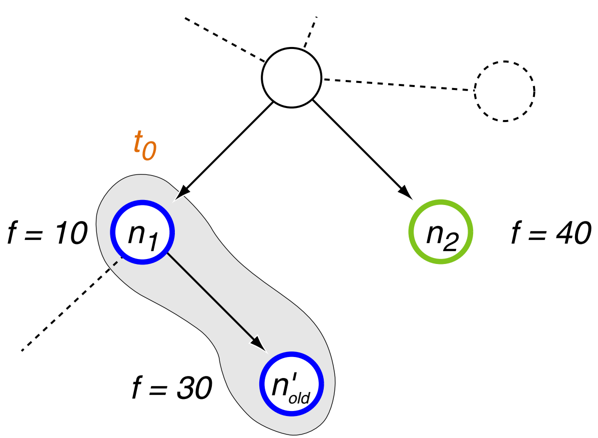
Best-First Search Algorithms

Re-evaluation of a Node n' (continued)

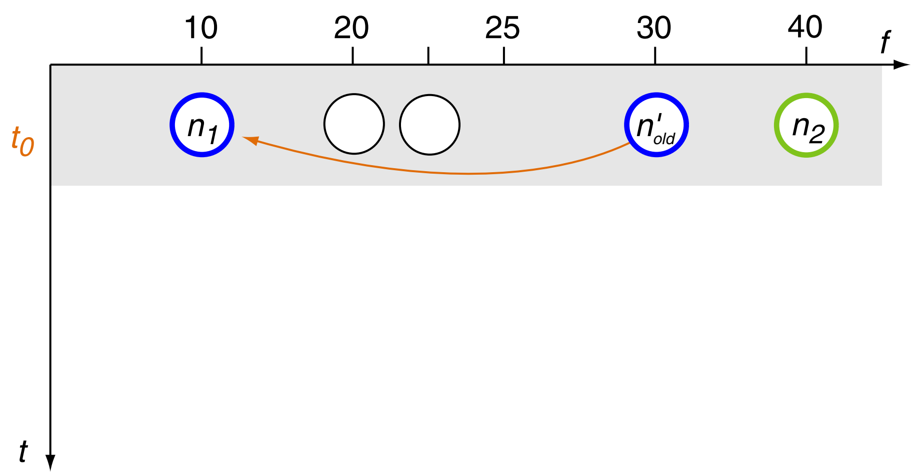
Case 2: n'_{old} is already on CLOSED.

```
5.    FOREACH n' IN successors(n) DO      // Expand n.
        ...
        n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
        IF ( n'_old == null )
        ...
        ELSE
          IF ( f(n') < f(n'_old) )    // Compare cost of solution bases.
          THEN   // Solution base of n' is cheaper:  path discarding.
            n'_old.parent = n'.parent;  f(n'_old) = f(n');
            IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED);  add(n'_old, OPEN, f(n'_old)); ENDIF
        ENDIF
```

State-space:



OPEN ∪ CLOSED list:



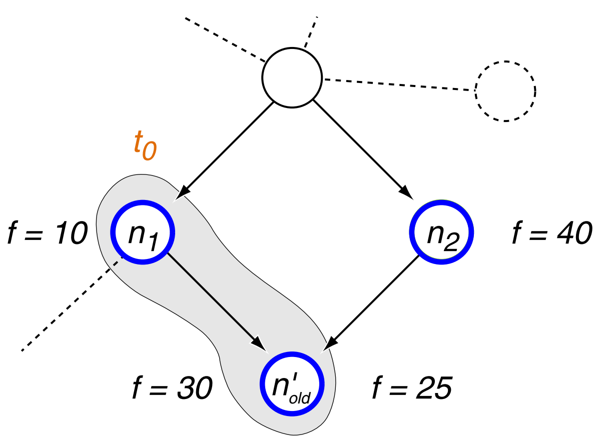
Best-First Search Algorithms

Re-evaluation of a Node n' (continued)

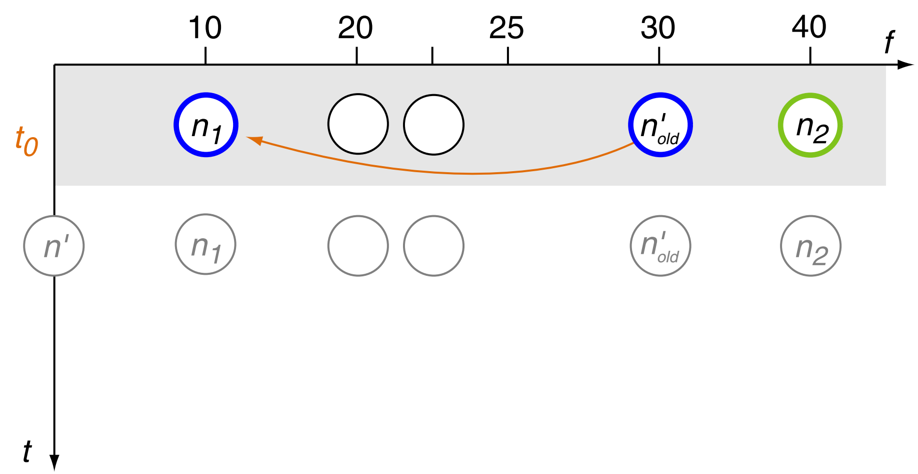
Case 2: n'_{old} is already on CLOSED.

```
5.   FOREACH n' IN successors(n) DO    // Expand n.
       ...
       n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
       IF ( n'_old == null )
       ...
       ELSE
         IF ( f(n') < f(n'_old) )    // Compare cost of solution bases.
         THEN   // Solution base of n' is cheaper:  path discarding.
           n'_old.parent = n'.parent;  f(n'_old) = f(n');
           IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED);  add(n'_old, OPEN, f(n'_old)); ENDIF
       ENDIF
```

State-space:



OPEN ∪ CLOSED list:



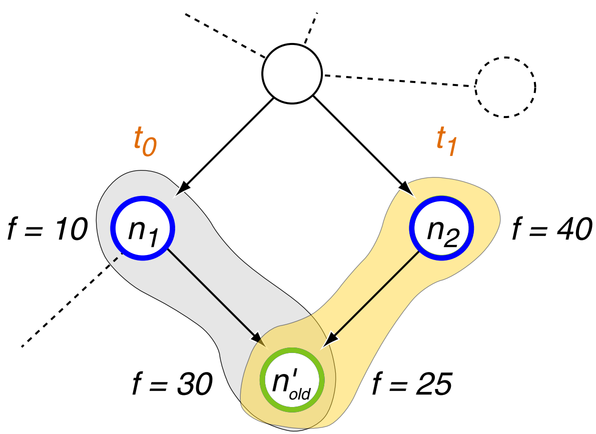
Best-First Search Algorithms

Re-evaluation of a Node n' (continued)

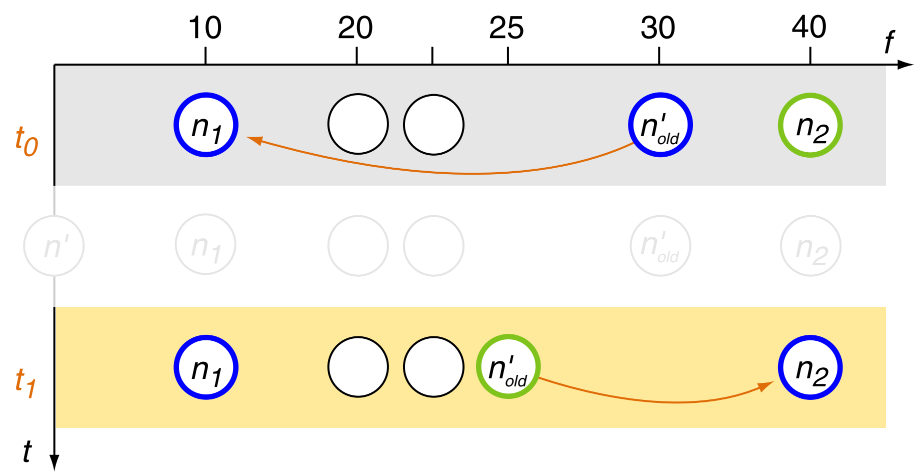
Case 2: n'_{old} is already on CLOSED.

```
5.    FOREACH n' IN successors(n) DO      // Expand n.
        ...
        n'_old = retrieve(n', OPEN ∪ CLOSED);     // State of n' already visited?
        IF ( n'_old == null )
        ...
        ELSE
          IF ( f(n') < f(n'_old) )    // Compare cost of solution bases.
          THEN   // Solution base of n' is cheaper:  path discarding.
            n'_old.parent = n'.parent;  f(n'_old) = f(n');
            IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED); add(n'_old, OPEN, f(n'_old)); ENDIF
        ENDIF
```

State-space:



OPEN ∪ CLOSED list:



Remarks:

- Given an occurrence of Case 2, it follows that f is not a monotonically increasing function in the solution base size (path length): $f(n') < f(n_2)$.

- Q. Given Case 2, and given the additional information that n_2 is a descendant of n'. What does this mean?

- Case 1 and Case 2 illustrate the path discarding behavior of algorithm BF, it follows that f is not a monotonically increasing function in the solution base size (path length): $f(n') < f(n_2)$.

- Implementation / efficiency issue: Instead of reopening a node n' (i.e., instead of moving n' from CLOSED to OPEN), a recursive update of the f-values and the back-pointers of its successors can be done. This is highly efficient but should only be done with care as it can easily lead to inconsistent traversal trees (wrong back-pointers).

  After reopening a node n', all the nodes n'' from which n' is reachable using only back-pointers are still available. Since the f-values stored with such nodes n'' are not updated, subsequent node expansions may use f-values not matching back-pointer paths. This can cause additional search efforts. Performing node expansion for nodes with invalid f-values can be avoided by using order-preserving functions f. Reopening nodes can be avoided by using monotonically increasing functions f (i.e., $f(n) \leq f(n')$ for successors n' of n).

Best-First Search Algorithms

Re-evaluation of a Node n' (continued)

Case 3: n'_{old} has been on OPEN but is not found on OPEN or CLOSED.

```
5.    FOREACH n' IN successors(n) DO     // Expand n.
        ...
        n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
        IF ( n'_old == null )
➜       THEN   // n' not in OPEN or CLOSED:  n' is a new state.
          add(n', OPEN, f(n'));
        ELSE
          ...
        ENDIF
```

Possible reasons:

1. There is no occurrence check. (State-space graph G is modeled as a tree.)
2. The occurrence check does not work properly. Note that state recognition can be a very hard (even undecidable) problem.
3. Explored parts of the state-space graph that seemed to be no longer required have been deleted by *cleanup_closed*.

Best-First Search Algorithms

Re-evaluation of a Node n' (continued)

Case 3: n'_{old} has been on OPEN but is not found on OPEN or CLOSED.

```
5.    FOREACH n' IN successors(n) DO      // Expand n.
        ...
        n'_old = retrieve(n', OPEN ∪ CLOSED);    // State of n' already visited?
        IF ( n'_old == null )
➜       THEN   // n' not in OPEN or CLOSED:  n' is a new state.
          add(n', OPEN, f(n'));
        ELSE
          ...
        ENDIF
```

Possible reasons:

1. There is no occurrence check. (State-space graph G is modeled as a tree.)
2. The occurrence check does not work properly. Note that state recognition can be a very hard (even undecidable) problem.
3. Explored parts of the state-space graph that seemed to be no longer required have been deleted by *cleanup_closed*.

Remarks:

- Q. What is the effect of the occurrence check in Case 1 and Case 2?
- Q. Should each visited node be stored in order to recognize the fact that its associated problem is encountered again?
- Q. Does a missing occurrence check affect the correctness of Algorithm BF?
- The shown version of the Algorithm BF has no call to *cleanup_closed*. However, such a call can be easily integrated, similar to the algorithms DFS or BFS.

Best-First Search Algorithms

```
BF*(s, successors, ⋆, f)   // A delayed termination variant of BF.
  1. s.parent = null;  add(s, OPEN, f(s));   // Store s on f-sorted OPEN.
  2. LOOP
  3.   IF (OPEN == ∅) THEN RETURN(Fail);
  4.   n = min(OPEN, f);   // Find most promising (cheapest) solution base.
 ➜     IF ⋆(n) THEN RETURN(n);   // Delayed termination.
       remove(n, OPEN);  add(n, CLOSED);
  5.   FOREACH n' IN successors(n) DO   // Expand n.
         n'.parent = n;
 ➜       ~~IF ⋆(n') THEN RETURN(n');~~   // Early termination removed.
         n'_old = retrieve(n', OPEN ∪ CLOSED);   // State of n' already visited?
         IF ( n'_old == null )
         THEN
           add(n', OPEN, f(n'));
         ELSE
           IF ( f(n') < f(n'_old) )
           THEN   // Solution base of n' is cheaper:  path discarding.
             n'_old.parent = n'.parent;  f(n'_old) = f(n');
             IF n'_old ∈ CLOSED THEN remove(n'_old, CLOSED);  add(n'_old, OPEN, f(n'_old)); ENDIF
           ENDIF
         ENDIF
       ENDDO
  6. ENDLOOP
```

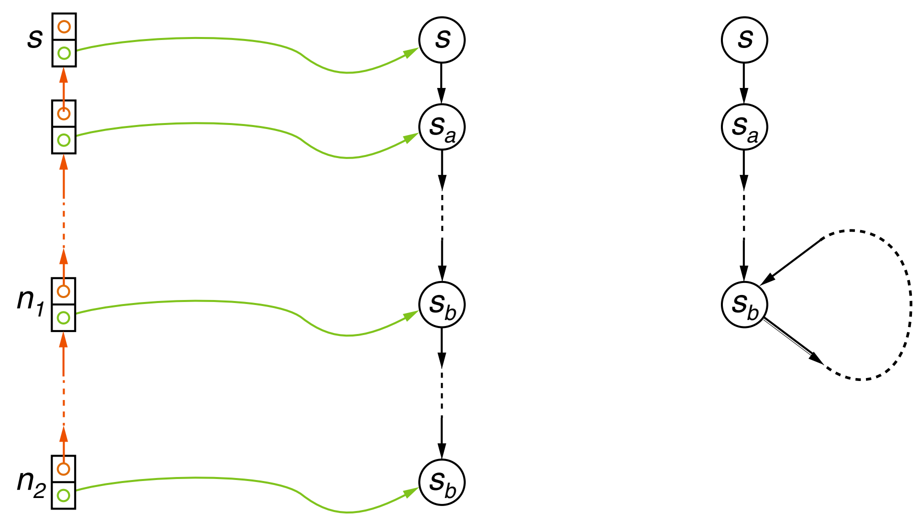

Best-First Search Algorithms

Definition 21 (Cycle-Averse Evaluation Function)

Let f be an evaluation function defined for state-space graph G.

f is called *cycle-averse*, if for each node n_2 with a cyclic back-pointer path, i.e., containing another node n_1 referring to the same state (n_1 is first occurrence, nearer to the start node s, and n_2 is some later occurrence), such that n_1 is reachable from n_2 via back-pointers, we have

$$f(n_1) \leq f(n_2) \qquad \text{i.e.,} \qquad f_{P_{s-n_1}}(n_1) \leq f_{P_{s-n_1-n2}}(n_2)$$



Back-pointer Structure Traversal Tree Path in G

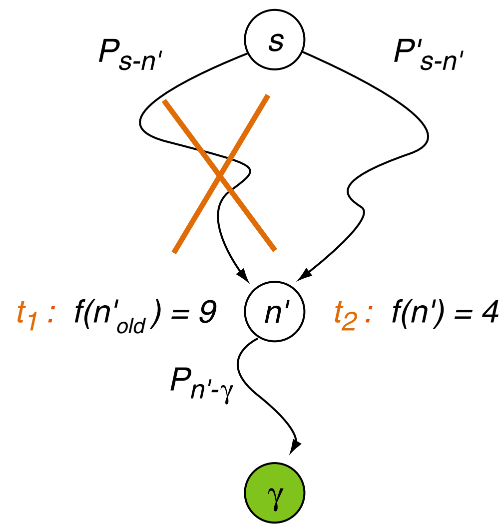
Remarks:

- If the task is to find a cheapest solution path that satisfies some constraints, we might not be successful when f is cycle-averse, even if path from start to goal nodes exist.

  As an example we can consider a minimum-path-length constraint, i.e., a solution path is required to have at least a path length of B for some B in $\mathbf{N}$. If a solution path exists, it might be necessary to "blow up" the path by adding cycles in order to meet the length constraint.

Best-First Search Algorithms

Irrevocable Path Discarding in BF



Path discarding is based on f-values computed for node instances.

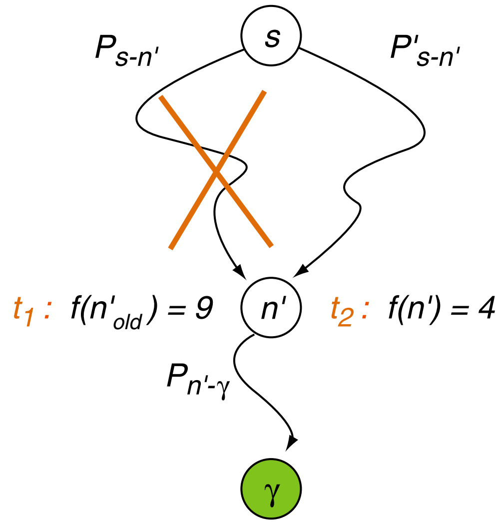
Irrevocability may not be allowable (solutions missed) if constraints on solution paths take into account *global properties* of the path.

Examples:

1. "Determine the shortest path (cheapest solution) that has two edges (operators) of equal costs."
2. "Determine a path (a solution) that minimizes the maximum edge cost difference (operator cost difference)."

Best-First Search Algorithms

Irrevocable Path Discarding in BF (continued)



Irrevocability is reasonable:

1. For constraint satisfaction problems, if the following equivalence holds:

   "Solution base $P_{s-n'}$ can be completed by $P_{n'-\gamma}$ to a solution path."

   $\Leftrightarrow$

   "Solution base $P'_{s-n'}$ can be completed by $P_{n'-\gamma}$ to a solution path."

2. For optimization problems, if for alternative solution bases the order w.r.t. cost estimations is preserved when using $P_{n'-\gamma}$ as their shared continuation.

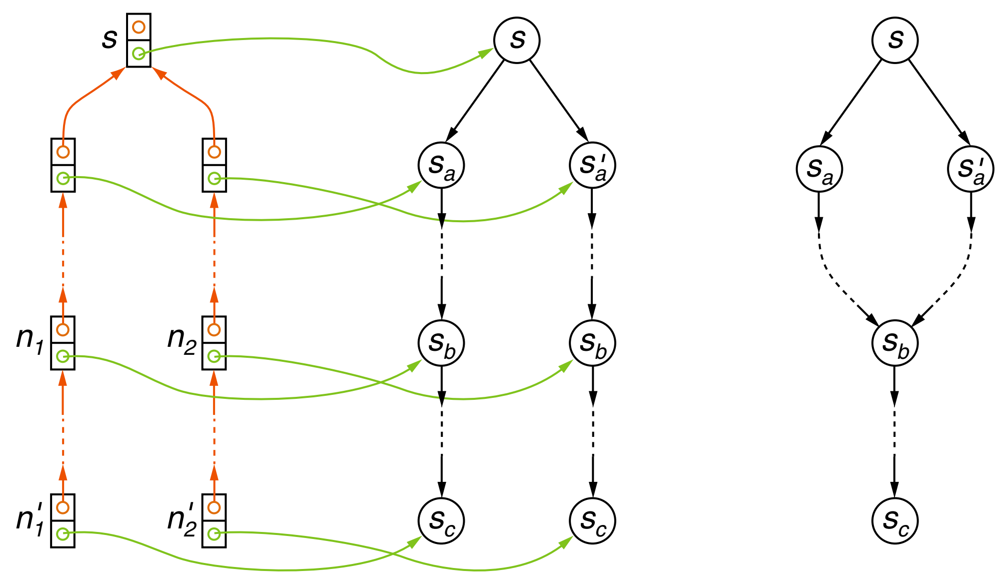
Best-First Search Algorithms

Definition 22 (Order-preserving Evaluation Function)

Let f be an evaluation function defined for state-space graph G.

f is called *order-preserving*, if for each pair of nodes n'_1 and n'_2 with predecessors n_1 and n_2 via back-pointers respectively, such that the back-pointer paths of n'_1 and n'_2 coincide from n_1 resp. n_2 on, then we have

$$f(n_1) \leq f(n_2) \;\Rightarrow f(n'_1) \leq f(n'_2)$$



Back-pointer Structure Traversal Tree Paths in G

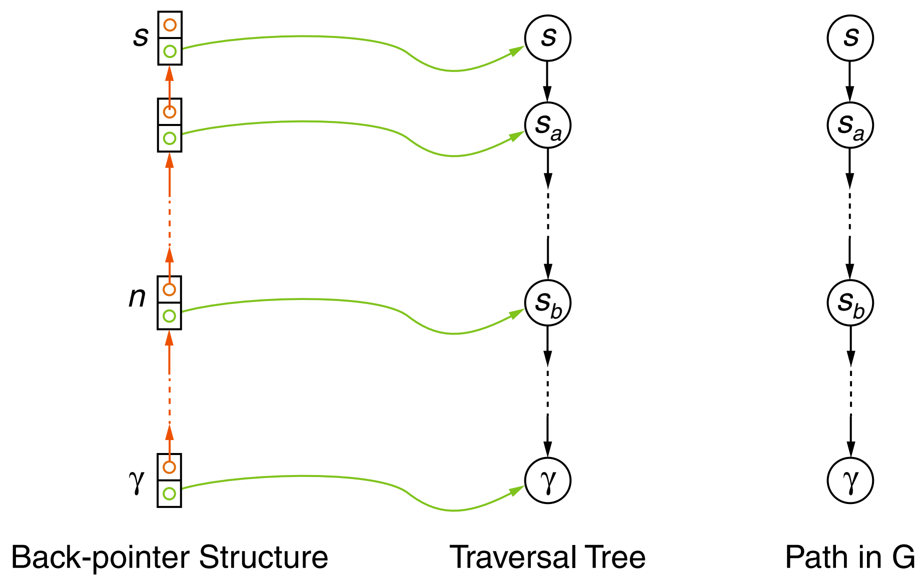
Best-First Search Algorithms

Definition 23 (Optimistic Evaluation Function)

Let G be state-space graph and f an evaluation function for G.

f is called *optimistic*, if for each goal node γ and each predecessor node n in the back-pointer path of γ (n reachable from γ via back-pointers), we have

$$f(n) \leq f(\gamma)$$



Remarks:

- Let G be a state-space graph with non-negative cost values assigned to the edges. Let the evaluation function f be defined by

  $$f_{P_{s_0-s_1}}(s_1) = \text{sum of edge cost value in } P_{s_0-s_1}.$$

  Then f is optimistic.

Best-First Search Algorithms

Advanced Principles for an Algorithmization of Best-First Search for Optimization

Prop$_{BF}(G)$ Required Properties of G for Optimization

1. G has *Prop*$_1(G)$ properties.
2. f is cycle-avers. (Avoiding corrupted backpointer structures.)
3. f is order-preserving. (Avoiding path discarding problems.)

Additional property (kept separate as usual):

- f is optimistic. (Avoiding overestimation problems.)

Task

- Determine an optimum solution path for s in G.

Algorithmization

- The algorithm uses *Delayed Termination*. (Avoiding last step problems.)
- The algorithm uses *Path Discarding*. (Efficiency.)
- The tie breaking strategy for OPEN prefers goal nodes.

Important Properties of Search Algorithms

Definition 24 (Admissibility)

Let $\mathcal{A}$ be an algorithm searching a state-space graph G for a solution path for a given state s.

$\mathcal{A}$ is *admissible* if
$\mathcal{A}$ terminates returning an optimum (with respect to f) solution if a solution exists.

There is no guarantee for the existence of an optimum solution path, even if a solution path exists.

State Space Search

Lemma 25 (Admissibility of BF* for Finite Graphs)

Let G be for finite graphs G with *Prop*$_{BF}(G)$ and let f be an optimistic evaluation function for G. Then BF* is admissible.

Proof (sketch)

1. Since G is finite, the number of cycle-free solution paths starting in s is finite. Hence, a minimum cost solution path $P_{s-\gamma}$ exists in G. (Only cycle-free solution paths have to be considered, since f is cycle-averse and order-preserving.)
2. Assume, BF* terminates returning a non-optimum solution $P_{s-\gamma'}$.
   Hence, $f(\gamma) < f(\gamma')$.
3. At each point in time (whenever BF* is in step 2) before BF* terminates, there is a shallowest node n in $P_{s-\gamma}$ that is in OPEN.
   (Shallowest node in a path is the node nearest to the start node.)
   Hence, BF* cannot terminate with *Fail*.
4. A shallowest OPEN node on an optimum path is optimally reached, i.e., there is no path from s to n with a smaller f-value than that the current back-pointer path.
5. Since f is optimistic, we have $f(n) \leq f(\gamma)$.
6. This contradicts the termination returning $P_{s-\gamma'}$, since goal node γ' was selected from OPEN when also n was available on OPEN.