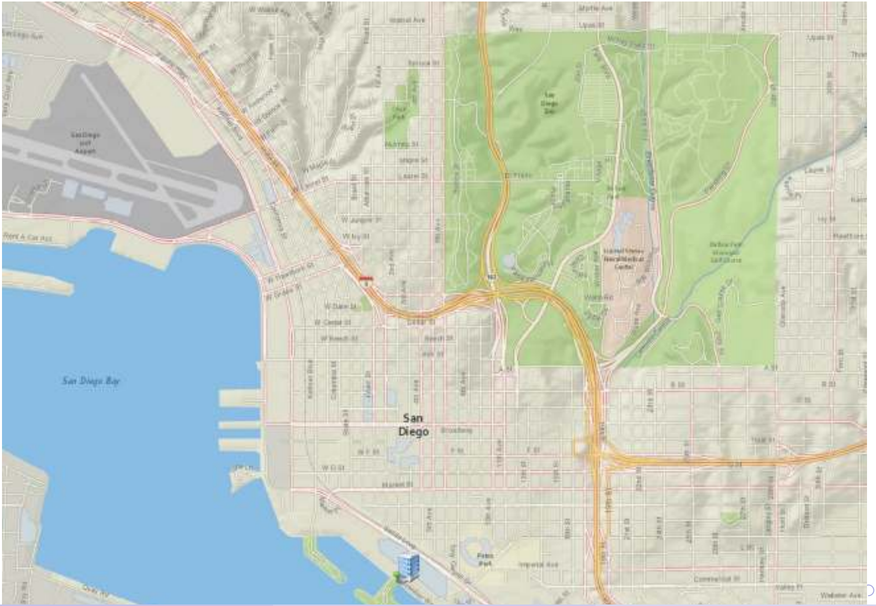
Exact Graph Search Algorithms for Generalized Traveling Salesman Path Problems

Michael N. Rice Vassilis J. Tsotras

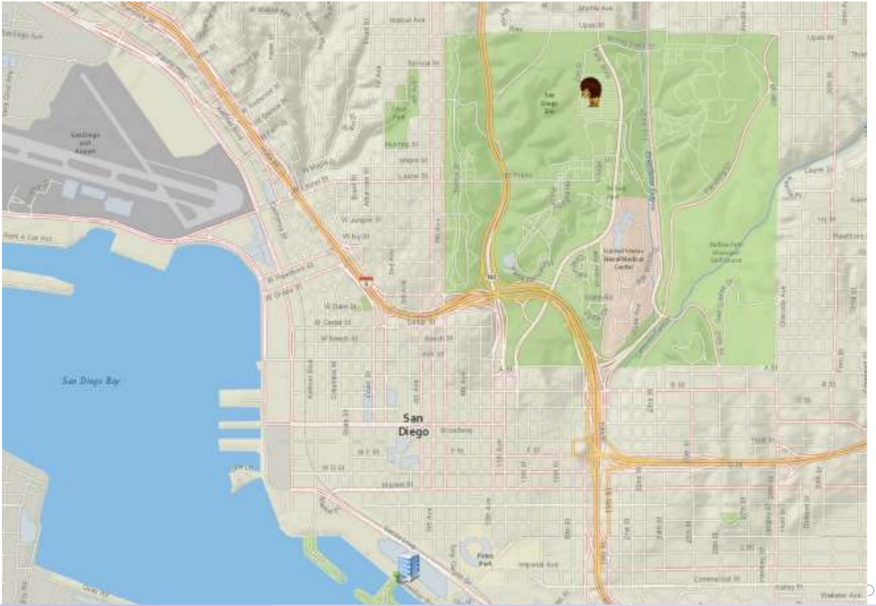
University of California, Riverside (UCR)

11th International Symposium on Experimental Algorithms

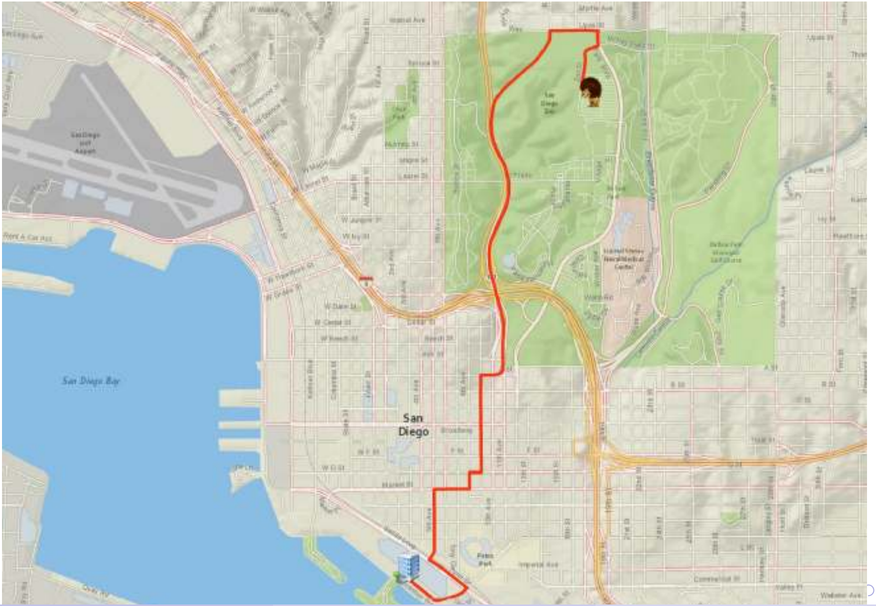
Motivation



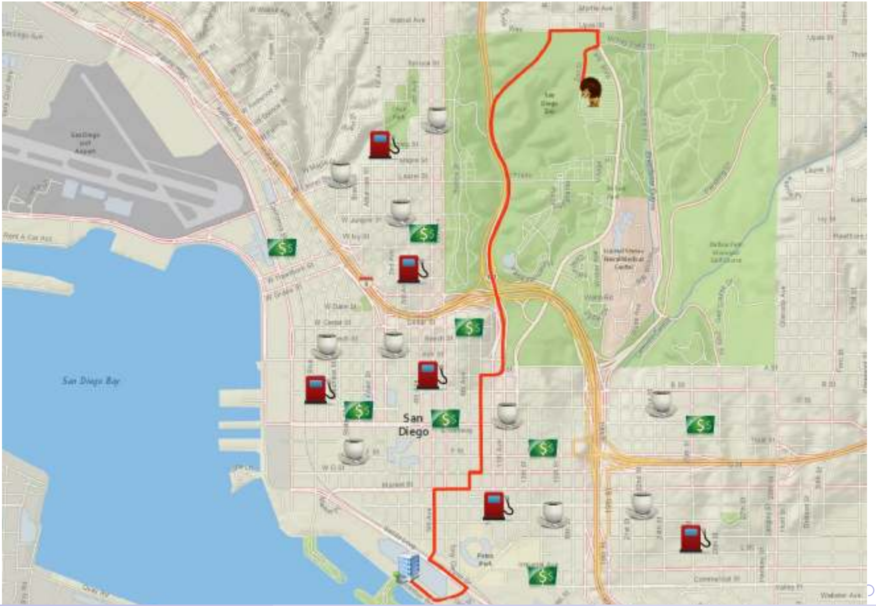
Motivation



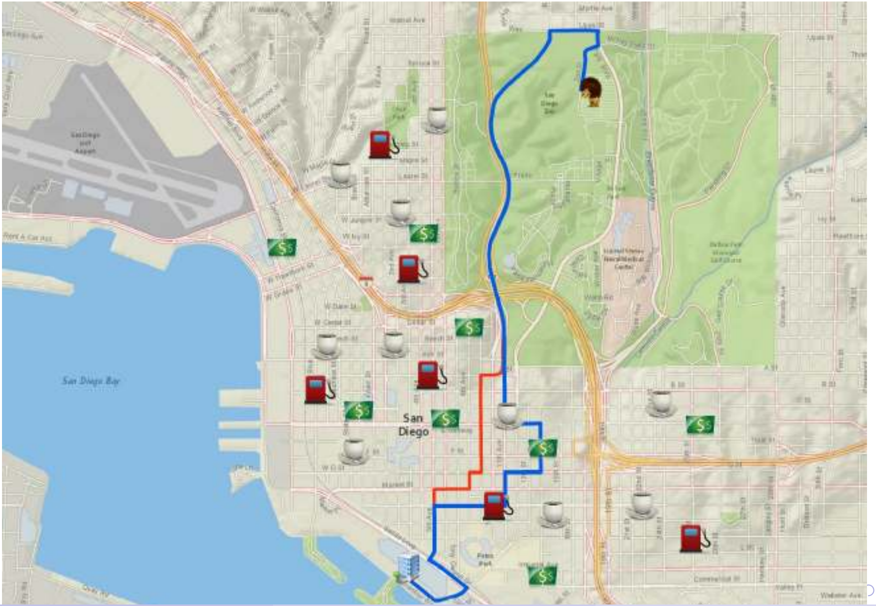
Motivation



Motivation



Motivation



Problem Definition

Problem Definition

Graph

- Weighted, directed graph $G = (V, E)$

Problem Definition

Graph

- Weighted, directed graph $G = (V, E)$

Category Set

- $C = \{C_1, C_2, \ldots, C_k\}$ defined on G
- $C_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,|C_i|}\} \subseteq V$
- Category count $k = |C|$, category density $g = \max_{1 \leq i \leq k} \{|C_i|\}$

Problem Definition

Graph

- Weighted, directed graph $G = (V, E)$

Category Set

- $C = \{C_1, C_2, \ldots, C_k\}$ defined on G
- $C_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,|C_i|}\} \subseteq V$
- Category count $k = |C|$, category density $g = \max_{1 \leq i \leq k} \{|C_i|\}$

Satisfying Path

A path, P, **satisfies** a category set C if, for $1 \leq i \leq k$, $P \cap C_i \neq \emptyset$.

Problem Definition

Graph

- Weighted, directed graph $G = (V, E)$

Category Set

- $C = \{C_1, C_2, \ldots, C_k\}$ defined on G
- $C_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,|C_i|}\} \subseteq V$
- Category count $k = |C|$, category density $g = \max_{1 \leq i \leq k} \{|C_i|\}$

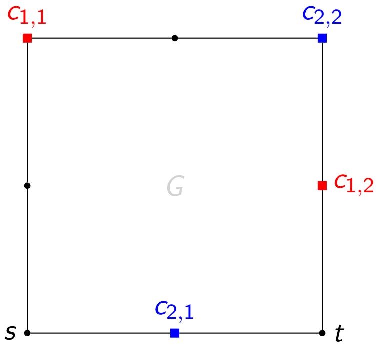
Satisfying Path

A path, P, **satisfies** a category set C if, for $1 \leq i \leq k$, $P \cap C_i \neq \emptyset$.

Generalized Traveling Salesman Path Problem (GTSPP)

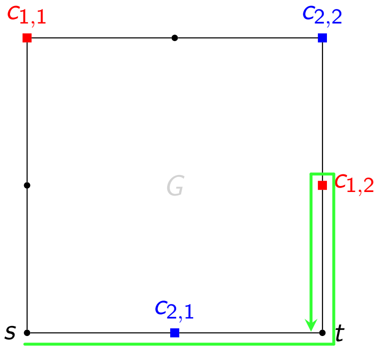
- Instance: $\langle s, t, C \rangle$, for $s, t \in V$ and category set C
- Solution: Minimum-weight satisfying path from s to t

GTSPP Example



- Instance: $\langle s, t, C \rangle$
- $C = \{C_1, C_2\}$
- $C_1 = \{c_{1,1}, c_{1,2}\}$
- $C_2 = \{c_{2,1}, c_{2,2}\}$

GTSPP Example



- Instance: $\langle s, t, C \rangle$
- $C = \{C_1, C_2\}$
- $C_1 = \{c_{1,1}, c_{1,2}\}$
- $C_2 = \{c_{2,1}, c_{2,2}\}$
- Solution: $P_{s,t}$ is optimal

Background and Related Work

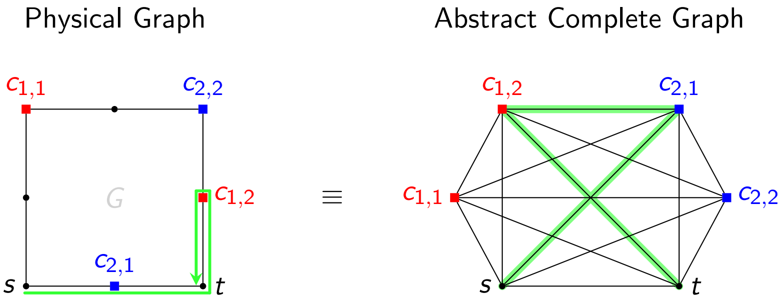
- GTSP introduced in 1960s
- NP-hard generalization of the classical TSP
- Goes by many names...
  - Errand Scheduling
  - Group TSP
  - Set TSP
  - One-of-a-Set TSP
  - Multiple-Choice TSP
  - TSP with Neighborhoods
  - ...
- Many exact, approximate, and heuristic approaches exist, **but**. . .

Limitations of Existing Work

- Existing work relies on complete-graph "abstraction"

Limitations of Existing Work

- Existing work relies on complete-graph "abstraction"



Limitations of Existing Work

- Requires intermediate processing stage during query to compute many-to-many cost matrix

Limitations of Existing Work

- Requires intermediate processing stage during query to compute many-to-many cost matrix
- $O(kg)$ graph searches just to set up the problem for other algorithms

Limitations of Existing Work

- Requires intermediate processing stage during query to compute many-to-many cost matrix
- $O(kg)$ graph searches just to set up the problem for other algorithms
- Our proposed algorithms have running times $O^*(2^k)$

Limitations of Existing Work

- Requires intermediate processing stage during query to compute many-to-many cost matrix
- $O(kg)$ graph searches just to set up the problem for other algorithms
- Our proposed algorithms have running times $O^*(2^k)$
- Advantageous for problems where $k \ll g$

Limitations of Existing Work

- Requires intermediate processing stage during query to compute many-to-many cost matrix
- $O(kg)$ graph searches just to set up the problem for other algorithms
- Our proposed algorithms have running times $O^*(2^k)$
- Advantageous for problems where $k \ll g$
- Most GTSPP problems in personal navigation domain have this characteristic asymmetry:
  - Very few "errands" per trip (i.e., small k)
  - Many choices per "errand" (i.e., large g)

Limitations of Existing Work

- Requires intermediate processing stage during query to compute many-to-many cost matrix
- $O(kg)$ graph searches just to set up the problem for other algorithms
- Our proposed algorithms have running times $O^*(2^k)$
- Advantageous for problems where $k \ll g$
- Most GTSPP problems in personal navigation domain have this characteristic asymmetry:
  - Very few "errands" per trip (i.e., small k)
  - Many choices per "errand" (i.e., large g)

Limitations of Existing Work

- Requires intermediate processing stage during query to compute many-to-many cost matrix
- $O(kg)$ graph searches just to set up the problem for other algorithms
- Our proposed algorithms have running times $O^*(2^k)$
- Advantageous for problems where $k \ll g$
- Most GTSPP problems in personal navigation domain have this characteristic asymmetry:
  - Very few "errands" per trip (i.e., small k)
  - Many choices per "errand" (i.e., large g)

Canonical Example

- $k = 5$, $g = 10,000$
- Constructing complete graph would require ≈ 1 minute preparation
- We solve it optimally in < 2 seconds!

Product Graph Framework

Product Graph Framework

Covering Graph

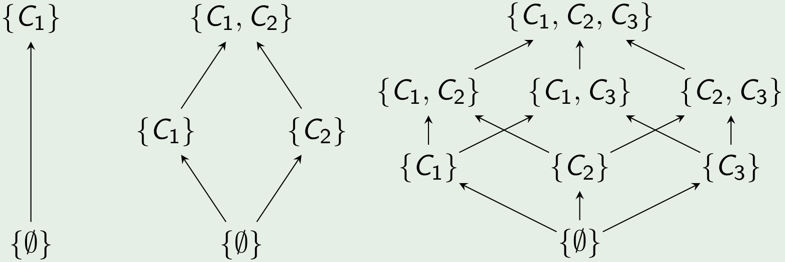
- For $C = \{C_1, C_2, \ldots, C_k\}$, let $G(\mathcal{B}_k) = (\mathcal{P}(C), E(\mathcal{B}_k))$
- $E(\mathcal{B}_k)$ is the minimal set of edges representing inclusion

Product Graph Framework

Covering Graph

- For $C = \{C_1, C_2, \ldots, C_k\}$, let $G(\mathcal{B}_k) = (\mathcal{P}(C), E(\mathcal{B}_k))$
- $E(\mathcal{B}_k)$ is the minimal set of edges representing inclusion

Examples



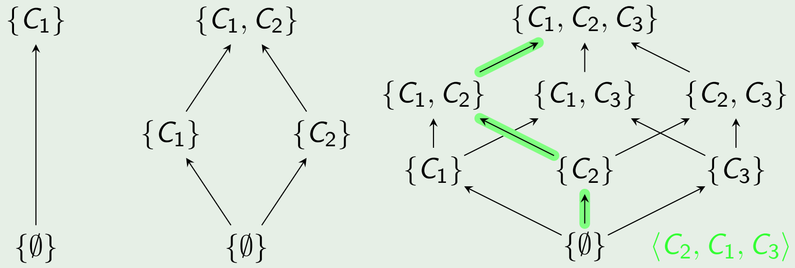
Covering graphs for $k = 1$, $k = 2$, and $k = 3$.

Product Graph Framework

Covering Graph

- For $C = \{C_1, C_2, \ldots, C_k\}$, let $G(\mathcal{B}_k) = (\mathcal{P}(C), E(\mathcal{B}_k))$
- $E(\mathcal{B}_k)$ is the minimal set of edges representing inclusion

Examples



Covering graphs for $k = 1$, $k = 2$, and $k = 3$.

Product Graph Framework

Product Graph Framework

Product Graph

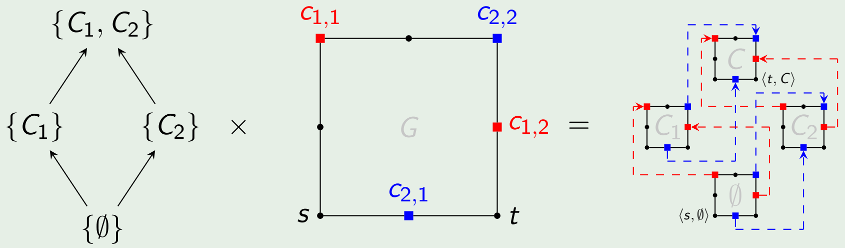
- Let $G_C = G \times G(\mathcal{B}_k) = (V \times \mathcal{P}(C), E_1 \cup E_2)$
- E_1 represents E for every subset in $\mathcal{P}(C)$
- E_2 represents accumulation of a new category via a category node

Product Graph Framework

Product Graph

- Let $G_C = G \times G(\mathcal{B}_k) = (V \times \mathcal{P}(C), E_1 \cup E_2)$
- E_1 represents E for every subset in $\mathcal{P}(C)$
- E_2 represents accumulation of a new category via a category node

Example

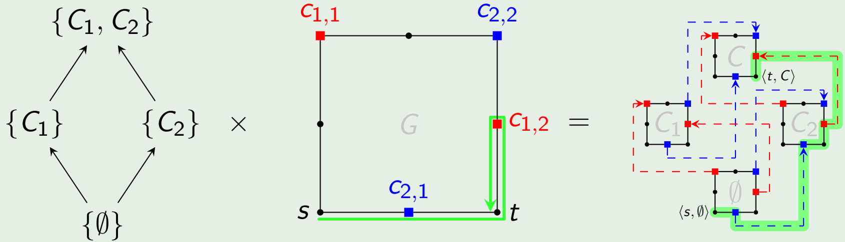


Product Graph Framework

Product Graph

- Let $G_C = G \times G(\mathcal{B}_k) = (V \times \mathcal{P}(C), E_1 \cup E_2)$
- E_1 represents E for every subset in $\mathcal{P}(C)$
- E_2 represents accumulation of a new category via a category node

Example



Product Graph Search Algorithms

Theorem

A shortest path in G_C from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ represents an equivalent-cost, optimal solution for instance $\langle s, t, C \rangle$ in the original graph G.

Product Graph Search Algorithms

Theorem

A shortest path in G_C from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ represents an equivalent-cost, optimal solution for instance $\langle s, t, C \rangle$ in the original graph G.

- Any shortest path search algorithm will work

Product Graph Search Algorithms

Theorem

A shortest path in G_C from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ represents an equivalent-cost, optimal solution for instance $\langle s, t, C \rangle$ in the original graph G.

- Any shortest path search algorithm will work
- E.g., Dijkstra's algorithm is a natural choice

Product Graph Search Algorithms

Theorem

A shortest path in G_C from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ represents an equivalent-cost, optimal solution for instance $\langle s, t, C \rangle$ in the original graph G.

- Any shortest path search algorithm will work
- E.g., Dijkstra's algorithm is a natural choice

Product Graph Search Algorithms

Theorem

A shortest path in G_C from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ represents an equivalent-cost, optimal solution for instance $\langle s, t, C \rangle$ in the original graph G.

- Any shortest path search algorithm will work
- E.g., Dijkstra's algorithm is a natural choice

Theorem

A Dijkstra search in G_C runs in $O(2^k(m + nk + nlogn))$ time.

Product Graph Search Algorithms

Theorem

A shortest path in G_C from $\langle s, \emptyset \rangle$ to $\langle t, C \rangle$ represents an equivalent-cost, optimal solution for instance $\langle s, t, C \rangle$ in the original graph G.

- Any shortest path search algorithm will work
- E.g., Dijkstra's algorithm is a natural choice

Theorem

A Dijkstra search in G_C runs in $O(2^k(m + nk + nlogn))$ time.

Optimization

- Do not explicitly construct the product graph
- Materialize the graph implicitly as needed

Advanced Product Graph Search

- We can do better!

Advanced Product Graph Search

- We can do better!
- We take advantage of two key aspects:
  - Recent progress in speedup techniques for road networks
  - Useful structural properties of the product graph

Advanced Product Graph Search

- We can do better!
- We take advantage of two key aspects:
  - Recent progress in speedup techniques for road networks
  - Useful structural properties of the product graph
- Extend product graph search to incorporate the state-of-the-art Contraction Hierarchies technique

Contraction Hierarchies: Overview

Contraction Hierarchies: Overview

CH Preprocessing

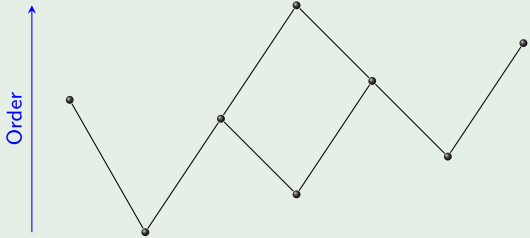
Establish strict total ordering of nodes (i.e., the "hierarchy"), and "contract" nodes in this order.

Contraction Hierarchies: Overview

CH Preprocessing

Establish strict total ordering of nodes (i.e., the "hierarchy"), and "contract" nodes in this order.

Example

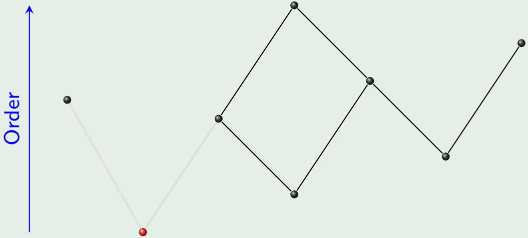


Contraction Hierarchies: Overview

CH Preprocessing

Establish strict total ordering of nodes (i.e., the "hierarchy"), and "contract" nodes in this order.

Example

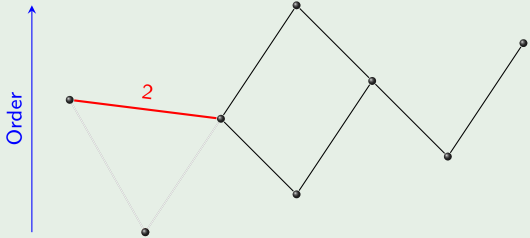


Contraction Hierarchies: Overview

CH Preprocessing

Establish strict total ordering of nodes (i.e., the “hierarchy”), and “contract” nodes in this order.

Example

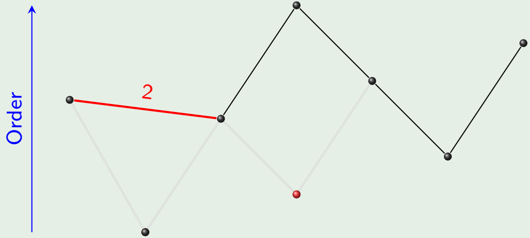


Contraction Hierarchies: Overview

CH Preprocessing

Establish strict total ordering of nodes (i.e., the "hierarchy"), and "contract" nodes in this order.

Example

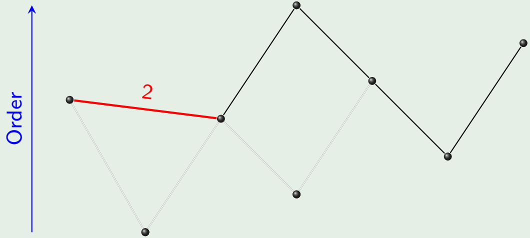


Contraction Hierarchies: Overview

CH Preprocessing

Establish strict total ordering of nodes (i.e., the "hierarchy"), and "contract" nodes in this order.

Example

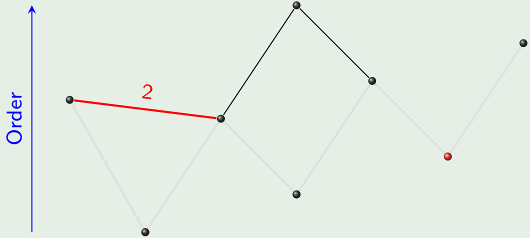


Contraction Hierarchies: Overview

CH Preprocessing

Establish strict total ordering of nodes (i.e., the “hierarchy”), and “contract” nodes in this order.

Example

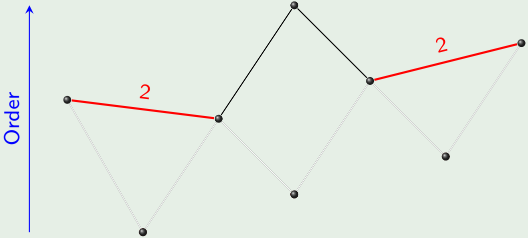


Contraction Hierarchies: Overview

CH Preprocessing

Establish strict total ordering of nodes (i.e., the "hierarchy"), and "contract" nodes in this order.

Example

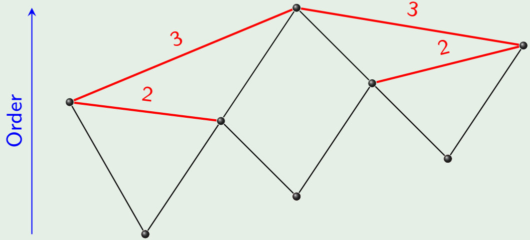


Contraction Hierarchies: Overview

CH Preprocessing

Establish strict total ordering of nodes (i.e., the "hierarchy"), and "contract" nodes in this order.

Example

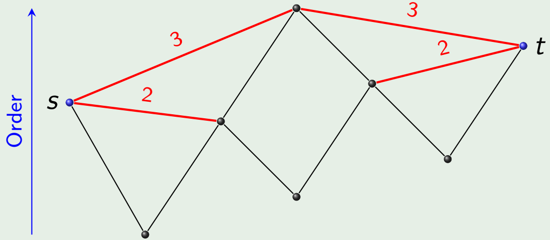


Contraction Hierarchies: Overview

CH Query

Bidirectional-Dijkstra search (forward from s, backward from t), relaxing only "upward-leading" edges.

Example

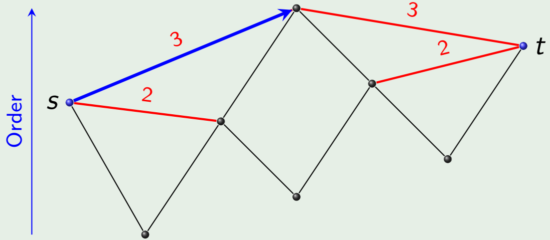


Contraction Hierarchies: Overview

CH Query

Bidirectional-Dijkstra search (forward from s, backward from t), relaxing only "upward-leading" edges.

Example

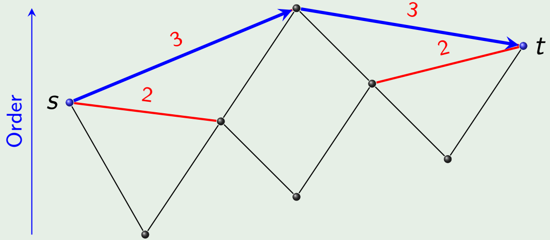


Contraction Hierarchies: Overview

CH Query

Bidirectional-Dijkstra search (forward from s, backward from t), relaxing only "upward-leading" edges.

Example

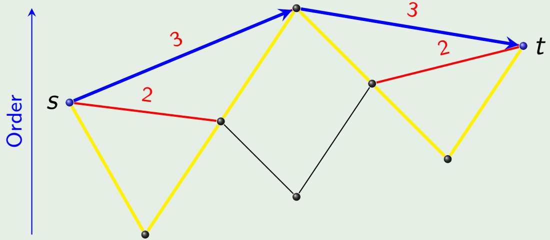


Contraction Hierarchies: Overview

CH Query

Bidirectional-Dijkstra search (forward from s, backward from t), relaxing only "upward-leading" edges.

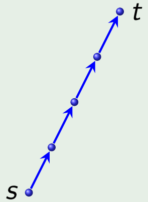
Example



CH Path Types

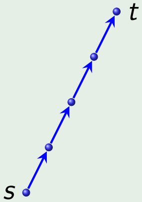
CH Path Types

Path Type #1: Increasing Rank

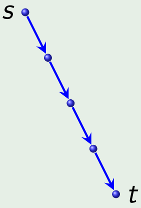


CH Path Types

Path Type #1: Increasing Rank

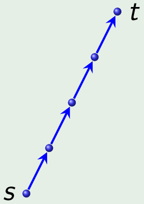


Path Type #3: Decreasing Rank

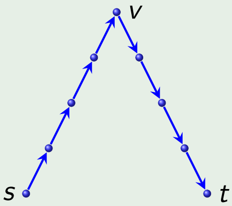


CH Path Types

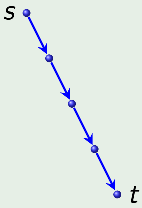
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank

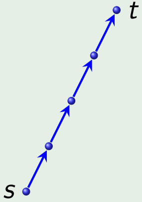


Path Type #3: Decreasing Rank

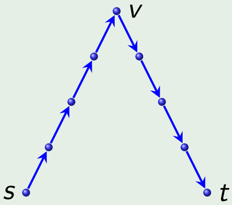


CH Path Types

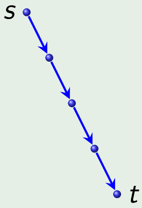
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

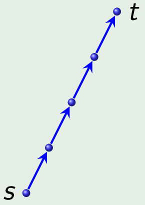


Alternate Algorithm: Sweeping Search

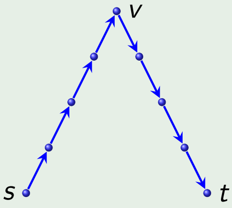
1. Take the union of "upward-reachable" search spaces from s and t

CH Path Types

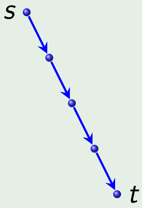
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

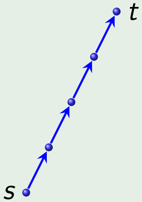


Alternate Algorithm: Sweeping Search

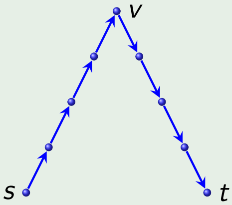
1. Take the union of "upward-reachable" search spaces from s and t
2. Sweep the unioned search space by node rank order

CH Path Types

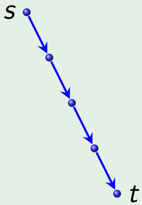
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

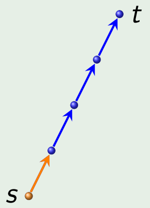


Alternate Algorithm: Sweeping Search

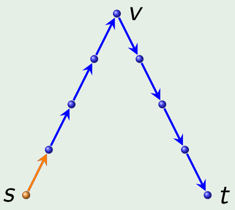
1. Take the union of "upward-reachable" search spaces from s and t
2. Sweep the unioned search space by node rank order
   1. Upsweep: relax outgoing "upward-leading" edges in increasing rank

CH Path Types

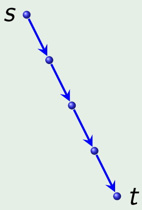
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

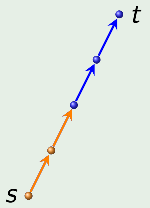


Alternate Algorithm: Sweeping Search

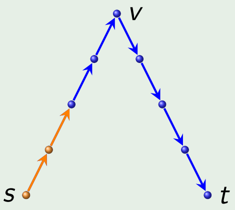
1. Take the union of “upward-reachable” search spaces from s and t
2. Sweep the unioned search space by node rank order
   1. Upsweep: relax outgoing “upward-leading” edges in increasing rank

CH Path Types

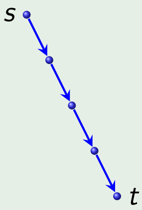
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

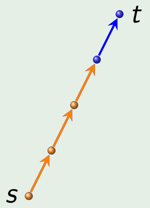


Alternate Algorithm: Sweeping Search

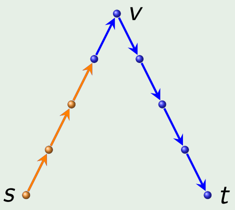
1. Take the union of "upward-reachable" search spaces from s and t
2. Sweep the unioned search space by node rank order
    1. Upsweep: relax outgoing "upward-leading" edges in increasing rank

CH Path Types

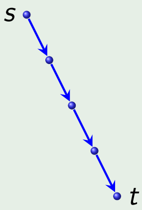
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank



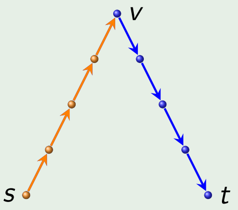
Alternate Algorithm: Sweeping Search

1. Take the union of "upward-reachable" search spaces from s and t
2. Sweep the unioned search space by node rank order
   1. Upsweep: relax outgoing "upward-leading" edges in increasing rank

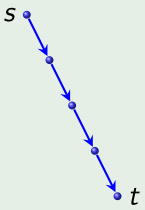
CH Path Types

Path Type #1: Increasing Rank

Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

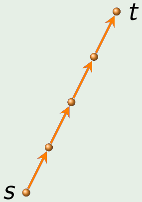


Alternate Algorithm: Sweeping Search

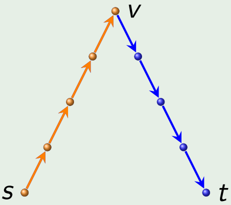
1. Take the union of “upward-reachable” search spaces from s and t
2. Sweep the unioned search space by node rank order
    1. Upsweep: relax outgoing “upward-leading” edges in increasing rank

CH Path Types

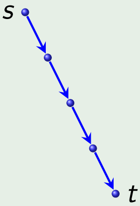
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

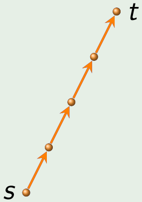


Alternate Algorithm: Sweeping Search

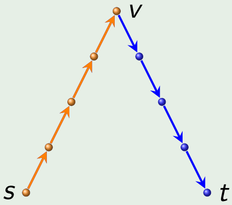
1. Take the union of "upward-reachable" search spaces from s and t
2. Sweep the unioned search space by node rank order
   1. Upsweep: relax outgoing "upward-leading" edges in increasing rank

CH Path Types

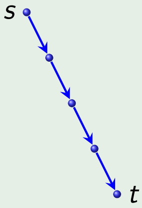
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

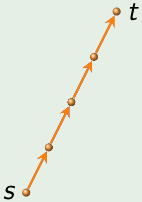


Alternate Algorithm: Sweeping Search

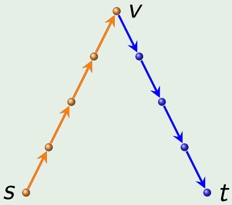
1. Take the union of “upward-reachable” search spaces from s and t
2. Sweep the unioned search space by node rank order
    1. Upsweep: relax outgoing “upward-leading” edges in increasing rank
    2. Downsweep: relax incoming “upward-leading” edges in decreasing rank

CH Path Types

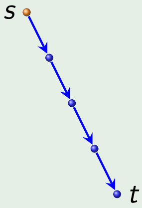
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank



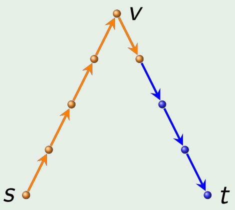
Alternate Algorithm: Sweeping Search

1. Take the union of “upward-reachable” search spaces from s and t
2. Sweep the unioned search space by node rank order
    1. Upsweep: relax outgoing “upward-leading” edges in increasing rank
    2. Downsweep: relax incoming “upward-leading” edges in decreasing rank

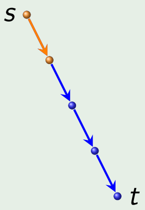
CH Path Types

Path Type #1: Increasing Rank

Path Type #2: Bitonic Rank



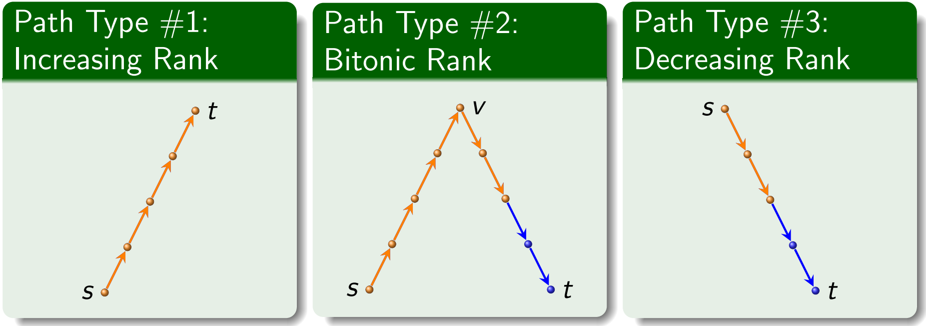
Path Type #3: Decreasing Rank



Alternate Algorithm: Sweeping Search

1. Take the union of “upward-reachable” search spaces from s and t
2. Sweep the unioned search space by node rank order
   1. Upsweep: relax outgoing “upward-leading” edges in increasing rank
   2. Downsweep: relax incoming “upward-leading” edges in decreasing rank

CH Path Types



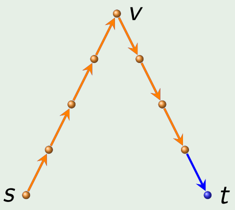
Alternate Algorithm: Sweeping Search

1. Take the union of "upward-reachable" search spaces from s and t
2. Sweep the unioned search space by node rank order
   1. Upsweep: relax outgoing "upward-leading" edges in increasing rank
   2. Downsweep: relax incoming "upward-leading" edges in decreasing rank

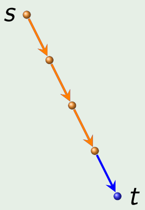
CH Path Types

Path Type #1: Increasing Rank

Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

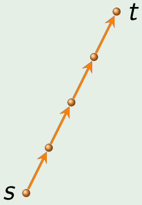


Alternate Algorithm: Sweeping Search

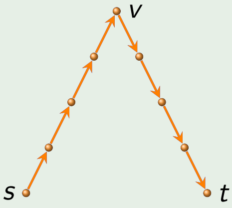
1. Take the union of "upward-reachable" search spaces from s and t
2. Sweep the unioned search space by node rank order
   1. Upsweep: relax outgoing "upward-leading" edges in increasing rank
   2. Downsweep: relax incoming "upward-leading" edges in decreasing rank

CH Path Types

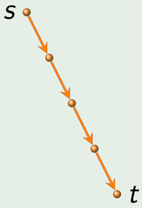
Path Type #1: Increasing Rank



Path Type #2: Bitonic Rank



Path Type #3: Decreasing Rank

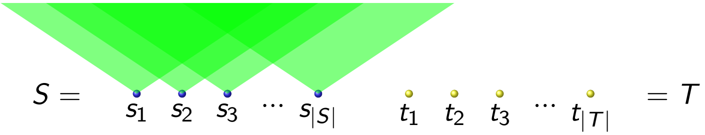


Alternate Algorithm: Sweeping Search

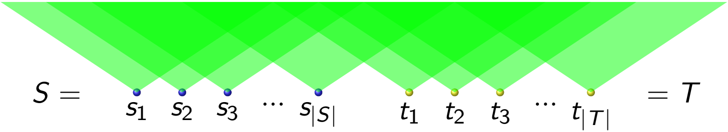
1. Take the union of "upward-reachable" search spaces from s and t
2. Sweep the unioned search space by node rank order
   1. Upsweep: relax outgoing "upward-leading" edges in increasing rank
   2. Downsweep: relax incoming "upward-leading" edges in decreasing rank

Many (S)ources/(T)argets

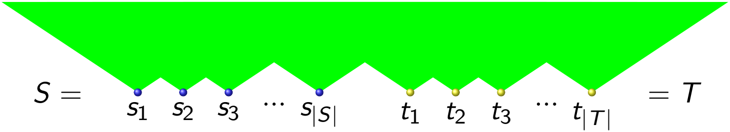
Many (S)ources/(T)argets



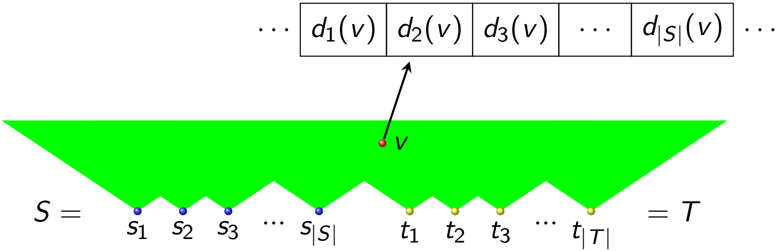
Many (S)ources/(T)argets



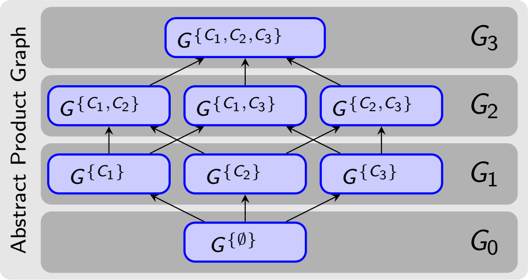
Many (S)ources/(T)argets



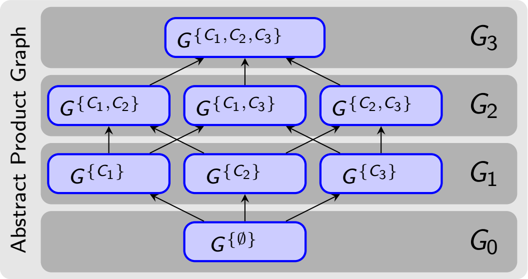
Many (S)ources/(T)argets



Utilizing Structural Properties of the Product Graph

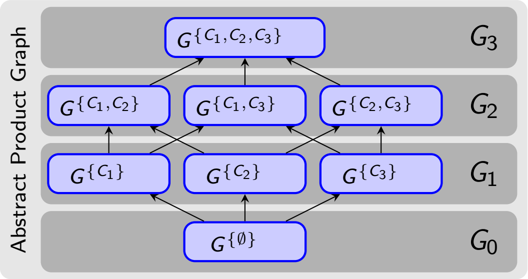


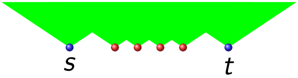
Utilizing Structural Properties of the Product Graph



Level-Sweeping Search (LESS) Algorithm

Utilizing Structural Properties of the Product Graph

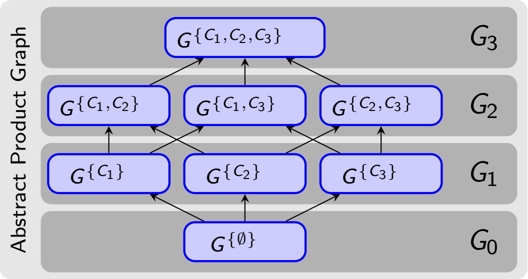


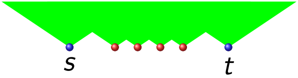


Level-Sweeping Search (LESS) Algorithm

1. Take the union of "upward-reachable" search spaces from s, t, *and C*

Utilizing Structural Properties of the Product Graph

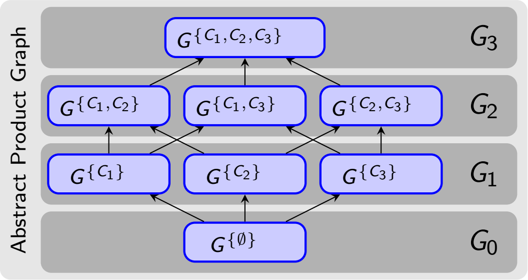


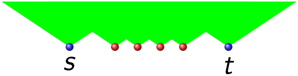


Level-Sweeping Search (LESS) Algorithm

1. Take the union of "upward-reachable" search spaces from s, t, *and C*
2. For $0 \leq i \leq k$, at each level G_i:

Utilizing Structural Properties of the Product Graph

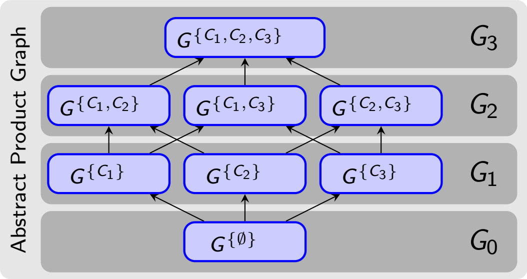


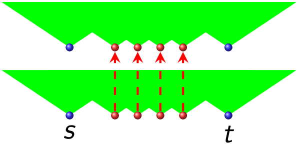


Level-Sweeping Search (LESS) Algorithm

1. Take the union of "upward-reachable" search spaces from s, t, *and C*
2. For $0 \leq i \leq k$, at each level G_i:
   1. Sweep the unioned search space for all $\binom{k}{i}$ subsets per node at level G_i

Utilizing Structural Properties of the Product Graph

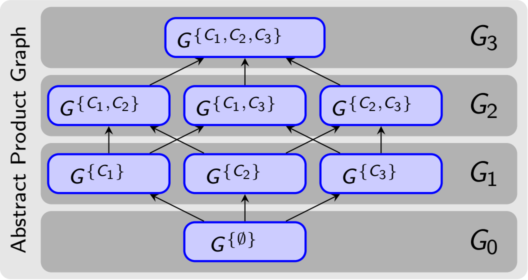


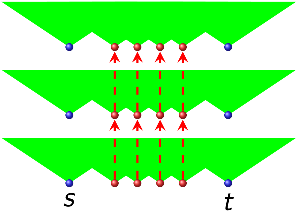


Level-Sweeping Search (LESS) Algorithm

1. Take the union of “upward-reachable” search spaces from s, t, *and C*
2. For $0 \leq i \leq k$, at each level G_i:
   1. Sweep the unioned search space for all $\binom{k}{i}$ subsets per node at level G_i
   2. If $i < k$, transfer costs to G_{i+1} along (zero-cost) E_2 edges

Utilizing Structural Properties of the Product Graph

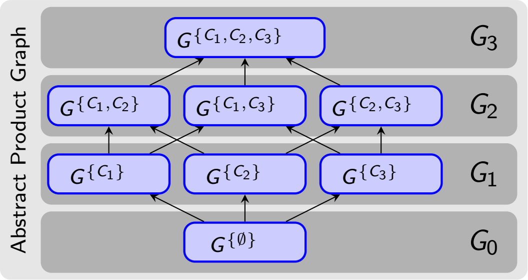


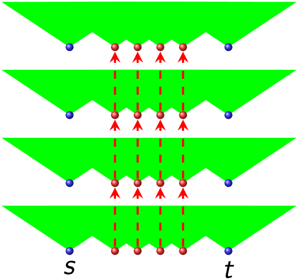


Level-Sweeping Search (LESS) Algorithm

1. Take the union of "upward-reachable" search spaces from s, t, *and C*
2. For $0 \leq i \leq k$, at each level G_i:
   1. Sweep the unioned search space for all $\binom{k}{i}$ subsets per node at level G_i
   2. If $i < k$, transfer costs to G_{i+1} along (zero-cost) E_2 edges

Utilizing Structural Properties of the Product Graph





Level-Sweeping Search (LESS) Algorithm

1. Take the union of "upward-reachable" search spaces from s, t, *and C*
2. For $0 \leq i \leq k$, at each level G_i:
    1. Sweep the unioned search space for all $\binom{k}{i}$ subsets per node at level G_i
    2. If $i < k$, transfer costs to G_{i+1} along (zero-cost) E_2 edges

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ time, where $m' = |E \cup E'|$.

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ time, where $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ *time, where* $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ *time, where* $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ time, where $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ time, where $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning (requires an admissible heuristic function $h : V \times V \to \mathbb{R}_{\geq 0}$)

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ *time, where* $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning (requires an admissible heuristic function $h : V \times V \to \mathbb{R}_{\geq 0}$**)**

1. Establish upper bound on optimal solution:

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ *time, where* $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning (requires an admissible heuristic function $h : V \times V \rightarrow \mathbb{R}_{\geq 0}$**)**

1. Establish upper bound on optimal solution:
   1. Construct satisfying path P' via greedy, nearest-neighbor strategy

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ *time, where* $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning (requires an admissible heuristic function $h : V \times V \to \mathbb{R}_{\geq 0}$)

1. Establish upper bound on optimal solution:
   1. Construct satisfying path P' via greedy, nearest-neighbor strategy
   2. $\mu = w(P')$

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ time, where $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning (requires an admissible heuristic function $h : V \times V \rightarrow \mathbb{R}_{\geq 0}$)

1. Establish upper bound on optimal solution:
   1. Construct satisfying path P' via greedy, nearest-neighbor strategy
   2. $\mu = w(P')$
2. For $1 \leq i \leq k$, for all $c_{i,j} \in C_i$:

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ *time, where* $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning (requires an admissible heuristic function $h : V \times V \to \mathbb{R}_{\geq 0}$)

1. Establish upper bound on optimal solution:
   1. Construct satisfying path P' via greedy, nearest-neighbor strategy
   2. $\mu = w(P')$
2. For $1 \leq i \leq k$, for all $c_{i,j} \in C_i$:
   1. Prune $c_{i,j}$ if $\mu < h(s, c_{i,j}) + h(c_{i,j}, t)$

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ time, where $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning (requires an admissible heuristic function $h : V \times V \to \mathbb{R}_{\geq 0}$)

1. Establish upper bound on optimal solution:
    1. Construct satisfying path P' via greedy, nearest-neighbor strategy
    2. $\mu = w(P')$
2. For $1 \leq i \leq k$, for all $c_{i,j} \in C_i$:
    1. Prune $c_{i,j}$ if $\mu < h(s, c_{i,j}) + h(c_{i,j}, t)$

Pruning

Theorem

LESS runs in $O(2^k(m' + nk))$ time, where $m' = |E \cup E'|$.

- In practice, its runtime is proportional to the size of the unioned search space (influenced by g)
- **Question:** can we reduce the size of the search space?
- If we can identify suboptimal category nodes, we can remove them

Pruning (requires an admissible heuristic function $h : V \times V \to \mathbb{R}_{\geq 0}$)

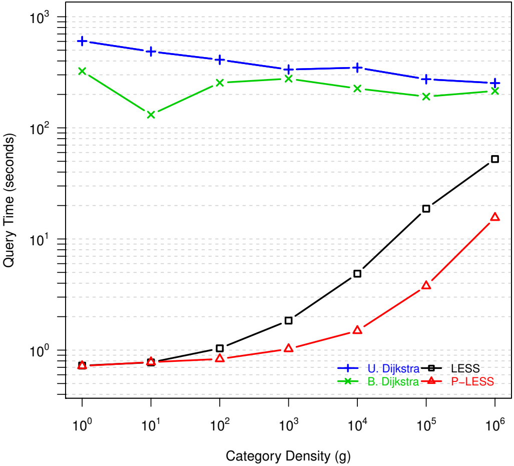
1. Establish upper bound on optimal solution:
   1. Construct satisfying path P' via greedy, nearest-neighbor strategy
   2. $\mu = w(P')$
2. For $1 \leq i \leq k$, for all $c_{i,j} \in C_i$:
   1. Prune $c_{i,j}$ if $\mu < h(s, c_{i,j}) + h(c_{i,j}, t)$

- After pruning, carry out LESS search, as before

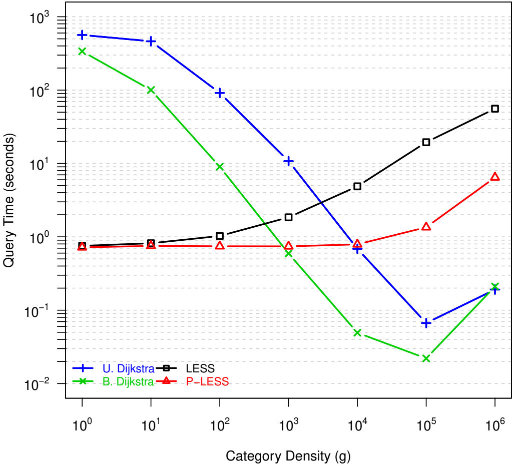
Experiments

- Dataset:
  - Road network of US/Canada with $|V| = 21M$ and $|E| = 52M$
- Environment:
  - Server: 2.53GHz CPU, 18GB RAM
  - Language: C++
- Preprocessing:
  - CH: 18 minutes preprocessing time
  - Pre-Computed Cluster Distances (PCD): 7 minutes (using CH)
- Algorithms:
  - Unidirectional Dijkstra (U. Dijkstra)
  - Bidirectional Dijkstra (B. Dijkstra)
  - Level-Sweeping Search (LESS)
  - LESS + Pruning (P-LESS)
- Queries:
  - Non-Local Queries: cases where $s \neq t$
  - Local Queries: cases where $s = t$

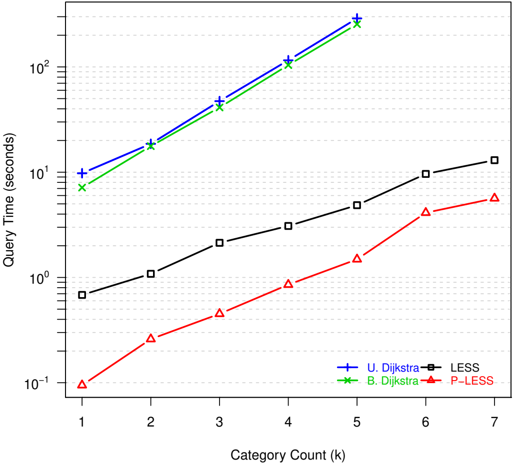
Category Density Experiments: Non-Local Queries ($s \neq t, k = 5$)



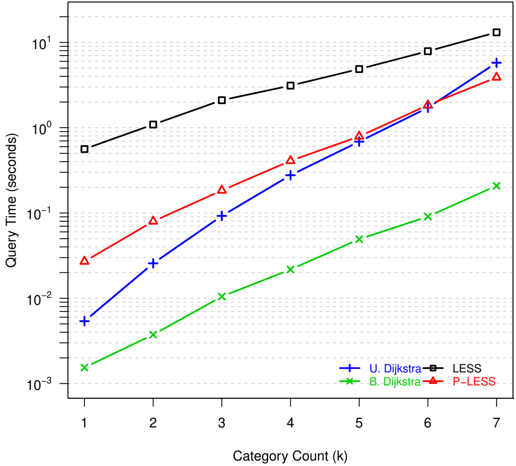
Category Density Experiments: Local Queries ($s = t, k = 5$)



Category Count Experiments: Non-Local Queries ($s \neq t, g = 10,000$)



Category Count Experiments: Local Queries ($s = t, g = 10,000$)



Summary

- New product graph framework for efficient graph search
- Can solve real-world GTSPP instances to optimality in seconds!
- Two competitive algorithms with performance tradeoffs:
  - Dijkstra: good for highly-local, very-dense queries (no pre-processing required)
  - LESS (with pruning): more consistent performance across various sizes and localities

Future Work

- Better space utilization (e.g., reduced memory overhead, better cache locality)
- More aggressive pruning strategies
- Incorporate goal-direction (e.g., A^*)
- Parallelization (exploiting subgraph independence)
- Approximation algorithms

Questions?